O'REILLY®

Second Edition

High Performance Spark

Best Practices for Scaling and Optimizing Apache Spark

Early Release

RAW & UNEDITED

Holden Karau, Anya Bida, Adi Polak & Rachel Warren

High Performance Spark

2ND EDITION

Best Practices for Scaling and Optimizing Apache Spark

Holden Karau, Anya Bida, Adi Polak, and Rachel Warren

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

High Performance Spark

by Holden Karau, Anya Bida, Adi Polak and Rachel Warren

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Aaron Black
- Development Editor: Shira Evans
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery

- May 2017: First Edition
- June 2025: Second Edition

Revision History for the Early Release

- 2023-05-24: First Release
- 2024-05-17: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098145859 for release details.

978-1-098-14579-8

Brief Table of Contents (*Not Yet Final*)

Ch 14: Debugging, tuning and other things developers like to pretend don't exist (not available)

Chapter 1. Introduction to High Performance Spark

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo is available at *https://github.com/high-performance-spark/high-performance-spark-examples*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

This chapter provides an overview of what we hope you will be able to learn from this book and does its best to convince you to learn to read some Scala and consider writing your Spark jobs in Scala or Python.

Feel free to skip ahead to Chapter 2 if you already know what you're looking for.

What Is Spark and Why Performance Matters

ASF (currently) stands for Apache Software Foundation, although there are calls to rename the foundation. Spark is a high-performance, general-purpose distributed computing system that has become the most active ASF open source project, with more than 1,000 active contributors.[1]

Spark enables us to process large quantities of data, beyond what can fit on a single machine, with a high-level, relatively easy-to-use API. Spark’s design and interface are unique, and it is one of the fastest systems of its kind. Uniquely, Spark allows us to write the logic of data transformations and machine learning algorithms in a parallelizable way, while being relatively system-agnostic.[2]

However, despite its many advantages and the excitement around Spark, the simplest implementation of many common data science routines in Spark can be much slower and less robust than the best version. Since the computations we are concerned with may involve data at a very large scale, the time and resource gains from tuning code for performance are enormous. Performance does not just mean running faster; often, at this scale, it means getting something to run at all. It is possible to construct a Spark query that fails on gigabytes of data but, when refactored and adjusted with an eye toward the structure of the data and the requirements of the cluster, succeeds on the same system with terabytes of data. In our experience writing production Spark code, we have seen the same tasks, run on the same clusters, run 100 times faster using some of the optimizations discussed in this book. In terms of data processing, time is money, and we hope this book pays for itself through a reduction in data infrastructure costs and developer hours.

Not all of these techniques apply to every use case. Especially because Spark is highly configurable and is exposed at a higher level than other computational frameworks of comparable power, we can reap tremendous benefits just by becoming more attuned to the shape and structure of our data. Some techniques can work well on certain data sizes or even certain key distributions, but not all. As a simple example, using `groupByKey` in Spark can very easily cause the dreaded out-of-memory exceptions, but for data with few duplicates, this operation can be just as quick as the alternatives that we will present. Learning to understand your particular use case and system and how Spark will interact with it is a must to solve the most complex data science problems with Spark.

What You Can Expect to Get from This Book

Our hope is that this book will help you take your Spark queries and make them faster, able to handle larger data sizes, and use fewer resources. This book covers a broad range of tools and scenarios. You will likely pick up some techniques that might not apply to the problems you are working with, but that might apply to a problem in the future and may help shape your understanding of Spark more generally. Most of the chapters in this book are written with enough context to allow the book to be used as a reference; however, the structure of this book is intentional and reading the sections in order should give you not only a few scattered tips, but a comprehensive understanding of Spark and how to make it sing.

It's equally important to point out what you will likely not get from this book. This book is not intended to introduce Spark, Scala, or Python; several other books and video series are available to get you started. The authors may be a little biased in this regard,[3] but we think *Learning Spark* 2nd edition by Jules Damji, Brooke Wenig, Tathagata Das, Denny Lee, is an excellent option for Spark beginners. While this book is focused on performance, it is not an operations book, so topics like setting up a cluster and multitenancy are not covered. We assume you already have a way to use Spark in your system, so we won't provide much assistance in making higher-level architecture decisions.

Spark Versions

Spark attempts to follow semantic versioning with the standard [MAJOR].[MINOR].[MAINTENANCE] with API stability for public **non-experimental non-developer** APIs within minor and maintenance releases. Many of these experimental components are some of the more exciting ones from a performance standpoint, including things like custom aggregations, and expressions to accelerate Spark's Datasets evaluation.

Spark aims for binary API compatibility between releases, using MiMa,[4] so if you are using the stable API theoretically, you generally should not need to recompile to run a job against a new version of Spark unless the major version has changed. In practice, we recommend recompiling all Spark jobs against the latest MINOR version as mistakes in binary compatibility have been known to happen.

TIP

This book was created using the Spark 3.3.0 APIs, but much of the code will work in earlier versions of Spark as well. In places where this is not the case, we have attempted to call that out.

Why the focus on Scala and Python?

In this book, we will focus on Spark's Scala and Python APIs with call-outs to other languages where relevant. Part of this decision is simply in the interest of time and space; we trust readers wanting to use Spark in another language can probably read one of (if not both) Scala and Python.

NOTE

In the first edition, we said that we believed most users seeking high performance Spark would have to use some Scala. However improvements in Spark's Python integrations mean this assertion is no longer correct.

While we don't expect you to know how to write Scala and Python code, we expect that you'll be able to **read** Scala or Python.

To Be a Spark Expert You Have to Be Able to Read a Little Scala Anyway

Although Python and Java are more commonly used languages, learning **to read** Scala is a worthwhile investment for anyone interested in delving deep into Spark development. Spark's documentation can be uneven. However, the readability of the codebase is world-class. Perhaps more than with other frameworks, the advantages of cultivating a sophisticated understanding of the Spark codebase is integral to the advanced Spark user. Because Spark is written in Scala, it will be difficult to interact with the Spark source code without the ability, at least, to read Scala code. The methods in the *Resilient Distributed Datasets* (RDD) class closely mimic those in the Scala collections API. RDD functions, such as `map`, `filter`, `flatMap`, `reduce`, and `fold`, have nearly identical specifications to their Scala equivalents.[5] Even folks who primarily use the API will appreciate being able to understand the underlying RDDs.

Fundamentally Spark is a functional framework, relying heavily on concepts like immutability and lambda definition, so using the Spark API may be more intuitive with some knowledge of functional programming. Programmers familiar with functional programming in Python or Scala (map, filter, etc.) will have the easiest time.

The Spark Scala and Python APIs Are Easier to Use Than the Java API

Once you have learned Scala or Python, you will quickly find that writing Spark in Scala or Python is less painful than writing Spark in Java. The Spark shell can be a powerful tool for debugging and development, and is only available in languages with existing REPLs (Scala, Python, and R).

Why Not Scala?

There are several good reasons to develop with Spark in other languages. One of the more important reasons is developer/team preference. Existing code, both internal and in libraries, can also be a strong reason to use a different language. Python is one of the most supported languages today, with some of the best machine-learning tools available.

While writing Java code can be clunky and sometimes lag slightly in terms of API, there is very little performance cost to writing in another JVM language (at most some object conversions). Spark’s Java APIs also tend to be “longer lived” than it’s Scala APIs, by the simple nature of historically Scala not providing binary compatibility between versions.[6]

TIP

While all of the examples in this book are presented in Scala for the final release, we will port many of the examples from Scala to Java and Python where the differences in implementation could be important. These will be available (over time) at our GitHub. If you find yourself wanting a specific example ported, please either email us or create an issue on the GitHub repo.

Spark SQL is not only simpler to understand and write, it also has performance optimizations that one would have to do by hand in RDDs. In some ways, Spark SQL is like the “autopilot” option of Spark, you should still keep an eye on it but it does a pretty good job. Since the code is transpiled behind the scenes, the ability to **read** Scala and Java code can be useful. This transpiling does much to minimize the performance difference when using a non-JVM language. Chapter 8 looks at options to work effectively in Spark with languages outside of the JVM, including Spark’s supported languages of Python and R. This section also offers guidance on how to use Fortran, C, and GPU-specific code to reap additional performance improvements. Even if we are developing most of our Spark applications in Scala, we shouldn’t feel tied to doing everything in Scala, because specialized libraries in other languages can be well worth the overhead of going outside the JVM.

Learning Scala

If after all of this we’ve convinced you to use Scala, there are several excellent options for learning Scala. Spark 3.3 is built against Scala 2.12 and cross-compiled against Scala 2.13.

Depending on how much we’ve convinced you to learn Scala, and what your resources are, there are several different options ranging from books to massive open online courses (MOOCs) to professional training.

For books, *Programming Scala*, 3rd Edition, by Dean Wampler can be great, although much of the actor system references are not relevant while working in Spark. The Scala language website also maintains a list of Scala books.

In addition to books focused on Spark, there are online courses for learning Scala. Functional Programming Principles in Scala, taught by Martin Ordersky, its creator, is on Coursera as well as Introduction to Functional Programming on edX. A number of different companies also offer video-based Scala courses, none of which the authors have personally experienced.

For those who prefer a more interactive approach, professional training is offered by several companies. Be sure to verify the company’s approach to software licensing early in your exploration.

Conclusion

Although you will likely be able to get the most out of Spark performance if you have some understanding of Scala, working in Spark does not require a knowledge of Scala. Special techniques to consider when working with other languages will be covered in Chapter 8. This book is aimed at individuals who already have a grasp of the basics of Spark, and we thank you for choosing *High Performance Spark* to deepen your knowledge of Spark.

The next chapter will introduce some of Spark's general design and evaluation paradigms that are important to understanding how to efficiently utilize Spark.

[1] *http://spark.apache.org/*

[2] Spark runs on the majority of common OSs and architectures as well as many types of cluster managers (YARN, Kubernetes, Mesos, etc.). It even runs on less common systems, like z/OS.

[3] Holden co-wrote the 1st edition of Learning Spark, and while she receives no royalties on it she thinks the second edition is great.

[4] MiMa is the Migration Manager for Scala and tries to catch binary incompatibilities between releases.

[5] Although, as we explore in this book, the performance implications and evaluation semantics are quite different.

[6] Despite the implications in *https://docs.scala-lang.org/overviews/core/binary-compatibility-of-scala-releases.html* releases rarely achieve these goals in meaningful ways see *https://mungingdata.com/scala/maintenance-nightmare-upgrade/* (plus the empirical evidence of cross building Scala versions on maven central).

Chapter 2. Upgrading Spark

When we started writing the second edition of this book, one of the first tasks we had to face was upgrading our examples from Spark 2.2 to Spark 3.3. In our day jobs, we also often face the task of helping people upgrade to new versions of Spark. Upgrading to new versions of Spark is important to be able to take advantage of its many performance improvements; some of these can be as simple as making your code run on the new engine whereas in other cases, you may need to use newer APIs. In this chapter you will learn about how to identify areas of Spark that have changed and where you may need to update your codebase.

Upgrading to newer versions of Spark is not as simple as bumping the version and basking in the joy of a new engine. While Spark officially aims to follow SemVer (semantic versioning), where it maintains API compatibility within the same major version, it frequently falls down in important places as we'll explore in this chapter. Notably, "developer APIs" – which high performance users are most likely to take advantage of – are excluded from Spark's promises around API compatibility. Don't let these obstacles keep you from upgrading, though; let's explore how to find the changes that will impact you, not just in 3.3, but any version of Spark.

Finding What You Need to Change

We like to classify breaking changes into three types, (1) those that break at compile time (Scala/Java/Kotlin only), (2) those that raise exceptions at runtime, and (3) those that produce differing output. These three categories are helpful as the techniques and impact differ between them. We will start with the easiest, compile time changes, but it is important for you to pay attention to runtime output changes as they are the easiest to miss and can have outsized negative impact.

Compile Time Changes

From a documentation point of view, there are a few places you should read. Spark's general migration guide and the release notes are great starting points. While the migration guide is good at calling out breaking changes, the release notes are especially useful for highlighting new APIs you may want to consider using for increased performance. Depending on which components you use, you should also look at the component-specific guide, like DataSet/SQL or ML. Additionally, Spark uses the MiMa build checker, where the Spark developers have to create exceptions when we break Java/Scala APIs. While MiMa only directly checks the JVM APIs, this checker gives you an idea of where the Python APIs might be changing too.

While Spark does generally try to support language versions for relatively long periods of time, major version updates (like Spark 3 v.s. Spark 2) may drop support of old versions of languages. While Spark 3 did drop Python 2 support, this was comparatively late in the game, with many popular libraries no longer supporting Python 2 by that point. Generally speaking, we believe Spark supports dependencies longer than other tools because Spark has such wide adoption.

Another important factor to consider when upgrading Spark is what other libraries you will need to upgrade along with it. For JVM users, Spark 3 increased the *shading* which means more of the libraries are renamed internally. Shading is intended to reduce the number of conflicts you encounter. However, if you're upgrading from Spark 2 to Spark 3, you may find that you have taken a transitive dependency through Spark, that you now need to make explicit. Thankfully most of these will show up at compile time. A good place to check what libraries have changed for Java users is maven central or look at the change log for the main build file. Now let's consider Python users. Spark has less direct Python dependencies, however shading is not an option, and some of the requirements are implicit. For example, some of Spark's performance improvements require that the PyArrow library is installed, but it is not automatically installed. Check the history of the setup file to see what Python libraries have changed.

Exceptions at Runtime

End-to-end test coverage, even if incomplete, can catch most situations where Spark has introduced a new exception at runtime. While you'll sometimes need to evaluate a few values (e.g. empty data tests may not be enough), by having your pipeline run end-to-end you should be aware of the newly prohibited uses. Often these prohibited uses will exhibit undefined behavior, or negative performance characteristics, so fixing them (regardless of upgrade) is often beneficial.

Good testing is an important part of all software development, but we realize the reality that not all your co-workers will either agree with that or work in a business environment which empowers them to spend the time needed to test their data pipelines. We cover testing in more detail in [Link to Come], but for the purpose of migration once you've got your code compiling and running you can use your tests to find areas where your assumptions about Spark may need to be updated. If you find yourself in the situation where you need to migrate an untested pipeline, we encourage you to first add tests and then proceed with the migration.

Differing Results

Differing Results is by far the most challenging part of Spark upgrades because this problem is silent; there's no error thrown. In addition to well-designed tests (the best prevention), we suggest running a side-by-side pipeline comparison. There are several different tools to simplify side-by-side pipeline comparisons, including LakeFS, Nessie, and Iceberg Write-Audit-Publish snapshotting. We've created a proof-of-concept tool that uses each of these to compare the outputs.

NOTE

Not every pipeline is simple to compare side-by-side, for example if your pipeline outputs a machine learning model with some element of randomness involved in selecting the initial weights. In those situations you can try splitting your pipeline into the ETL part which you can more easily make assertions about, and your machine learning code. Not only will this allow you to run (partial) side-by-side comparisons it will likely make it easier to write better tests.

Updating Your Code

While the above can seem like quite a lot, there are tools that can help you with the migrations. It's important to note that these tools, while helpful, are not suitable for fully automated usage as we'll explore below.

Scala

Spark's Scala API is the least stable of the APIs, but its type system makes it the easiest to catch changes and errors.

ScalaFix is a linting and refactoring tool for Scala. With ScalaFix, we create rules to simplify upgrading to new versions of Spark. You can start by creating a *.scalafix.conf* file with the rules (and their configuration) that you want to use. For example, in our automated upgrade project we created the conf file shown in Example 2-1.

Example 2-1. Scala Fix Configuration Example (./.scalafix.conf in the main example repo)

```
UnionRewrite.deprecatedMethod {
  "unionAll" = "union"
}

OrganizeImports {
  blankLines = Auto,
  groups = [
    "re:javax?\\."
    "scala."
```

```
      "org.apache.spark."
      "*"
    ],
    removeUnused = false
  }

  rules = [
    DisableSyntax,
    SparkAutoUpgrade,
    MigrateHiveContext,
    MigrateToSparkSessionBuilder,
    MigrateDeprecatedDataFrameReaderFuns,
    AccumulatorUpgrade,
    onFailureFix,
    ExecutorPluginWarn,
    UnionRewrite,
    GroupByKeyWarn,
    GroupByKeyRewrite,
    MetadataWarnQQ,
    ScalaTestExtendsFix,
    ScalaTestImportChange
  ]
```

sbt users can add the scala fix plugin to their plugins configuration as shown in Example 2-2.

Example 2-2. Adding Scala Fix Plugin (./project/plugins.sbt)

```
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-

resolvers += "sonatype-releases" at "https://oss

resolvers += "sonatype-snapshots" at "https://oss


addSbtPlugin("org.scoverage" % "sbt-scoverage" %

addDependencyTreePlugin

//tag::scalaFix[]
addSbtPlugin("ch.epfl.scala" % "sbt-scalafix" % "
//end::scalaFix[]

//tag::sbtJNIPlugin[]
addSbtPlugin("com.github.sbt" %% "sbt-jni" % "1.5
//end::sbtJNIPlugin[]

//tag::xmlVersionConflict[]
// See https://github.com/scala/bug/issues/12632
ThisBuild / libraryDependencySchemes ++= Seq(
  "org.scala-lang.modules" %% "scala-xml" % Versi
)
//end::xmlVersionConflict[]

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "2
```

To bring in the Spark specific rules you’ll need to add them as special Scala Fix dependencies as shown in Example 2-3.

Example 2-3. Adding Scala Fix Rules (./build.sbt)

```
lazy val root = (project in file("."))
  .aggregate(core, native)


organization := "com.highperformancespark"

//tag::addSparkScalaFix[]
ThisBuild / scalafixDependencies +=
  "com.holdenkarau" %% "spark-scalafix-rules-2.4
ThisBuild / scalafixDependencies +=
  "com.github.liancheng" %% "organize-imports" %
//end::addSparkScalaFix[]

lazy val V = _root_.scalafix.sbt.BuildInfo

scalaVersion := V.scala212
addCompilerPlugin(scalafixSemanticdb)
scalacOptions ++= List(
  "-Yrangepos",
  "-P:semanticdb:synthetics:on"
)
```

```
name := "examples"

publishMavenStyle := true

version := "0.0.1"
resolvers ++= Seq(
  "JBoss Repository" at "https://repository.jboss
  "Cloudera Repository" at "https://repository.cl
  "Apache HBase" at "https://repository.apache.or
  "Twitter Maven Repo" at "https://maven.twttr.co
  "scala-tools" at "https://oss.sonatype.org/cont
  "sonatype-releases" at "https://oss.sonatype.or
  "Typesafe repository" at "https://repo.typesafe
  "Second Typesafe repo" at "https://repo.typesaf
  "Mesosphere Public Repository" at "https://down
  Resolver.sonatypeRepo("public")
)

licenses := Seq("Apache License 2.0" -> url("http

def specialOptions = {
  // We only need these extra props for JRE>17
  if (sys.props("java.specification.version") > "
    Seq(
      "base/java.lang", "base/java.lang.invoke",
      "base/java.util", "base/java.util.concurren

      "base/sun.nio.ch", "base/sun.nio.cs", "base
      "base/sun.util.calendar", "security.jgss/su
    ).map("--add-opens=java." + _ + "=ALL-UNNAMED
  } else {
    Seq()
  }
}


val sparkVersion = settingKey[String]("Spark vers
val sparkTestingVersion = settingKey[String]("Spa


// Core (non-JNI bits)

lazy val core = (project in file("core")) // regu
  .dependsOn(native % Runtime)
  .settings(javah / target := (native / nativeCom
  .settings(sbtJniCoreScope := Compile)
  .settings(
    javacOptions ++= Seq("-source", "1.8", "-targ
    parallelExecution in Test := false,
    fork := true,
    javaOptions ++= Seq("-Xms4048M", "-Xmx4048M",
    Test / javaOptions ++= specialOptions,
    // 2.4.5 is the highest version we have with
    sparkVersion := System.getProperty("sparkVers
    sparkTestingVersion := "1.5.2",

    // additional libraries
    libraryDependencies ++= Seq(
      "org.apache.spark" %% "spark-core"
      "org.apache.spark" %% "spark-streaming"
      "org.apache.spark" %% "spark-sql"
      "org.apache.spark" %% "spark-hive"
      "org.apache.spark" %% "spark-hive-thriftse
      "org.apache.spark" %% "spark-catalyst"
      "org.apache.spark" %% "spark-yarn"
      "org.apache.spark" %% "spark-mllib"
      "com.holdenkarau" %% "spark-testing-base"
      //tag::scalaLogging[]
      "com.typesafe.scala-logging" %% "scala-logg
      //end::scalaLogging[]
      "net.java.dev.jna" % "jna" % "5.12.1"),
    scalacOptions ++= Seq("-deprecation", "-unche
    pomIncludeRepository := { x => false },
  )

// JNI Magic!
lazy val native = (project in file("native")) //
  .settings(nativeCompile / sourceDirectory := so
  .enablePlugins(JniNative) // JniNative needs to

//tag::xmlVersionConflict[]
// See https://github.com/scala/bug/issues/12632
ThisBuild / libraryDependencySchemes ++= Seq(
  "org.scala-lang.modules" %% "scala-xml" % Versi

)
//end::xmlVersionConflict[]
```

```
assemblyMergeStrategy in assembly := {
      case x => MergeStrategy.first
}

assemblyMergeStrategy in native := {
      case x => MergeStrategy.first
}

assemblyMergeStrategy in core := {
      case x => MergeStrategy.first
}
```

These upgrades are semi-automatic, and in some situations introduce nulls and or produce warnings in places where the migration cannot succeed without human intervention. You can run all of the configured rules with `scalaFixAll`.

You can integrate ScalaFix with most of the build tools in the Scala ecosystem, from gradle-scalafx to scalafix-maven-plugin. These all support the same `.scalafix.conf` file used in our example with `sbt` above.

For example, our upgrade to our example source code to Spark 3.3 is shown in this PR.

NOTE

In theory, with Scala 2.13 forward, the problem of incompatible Scala versions should be gone thanks to Dotty, but the majority of Spark code we know of is not yet on Scala 2.13.

You can also use ScalaFix to help smooth language version upgrades. There is a collection of community provided rules listed here as well as the built-in rules. We've used the import re-order tool as part of our upgrade to Spark 3.

Some ScalaFix rules do not play well with others. For example, many of the tools which manipulate imports do not play well together and need to be executed separately. You can execute individual rules with `scalaFixAll [rule]`.

Python

Since the pain of upgrading from Python 2 to Python 3, the Python community has become much more cautious when it comes to introducing breaking changes in their languages. That being said, Spark will continue to drop support for old versions of Python 3, as new features and performance improvements are frequently only available in new versions. In some situations, where Spark does not yet want to drop support for the older version, but does want to take advantage of new functionality, two different code-paths may exist inside of the Spark code. These situations are less likely to be thoroughly tested,[1] so it's worthwhile to run a relatively recent version of Python when possible.

We know part of the appeal for many Python users is its dynamic typing. However, we believe that once you have moved beyond the experimentation phase, it is time to begin to annotate your pipelines with types. These type annotations can catch errors before running long pipelines.

TIP

Real Python has a great Getting Started with Types tutorial for those new to Python typing.

Some of the common PySpark types you will want to take advantage of are `RDD[U` (a templated type) or `DataFrame` (a non-templated type).

SQL

Spark SQL has the least amount of API changes, but has the least amount of pre-run checking. Also, unlike Python and Scala, which have been through large breaking migrations (e.g. 2 → 3) and have strong open source tools to help with migrations, SQL migration tools are both more limited and proprietary.[2] SQL also does not have the same testing culture or tools as Python or Scala. While many companies have been through painful SQL migrations it’s more often focused on switching SQL variants (like getting off Oracle). Tools like SQLFluff and Meta’s UPM.[3]

There are a small number of SQL migration rules in the spark-upgrade repo. Both SQLFluff and the Spark SQL migration rules can be installed with pip, e.g. `pip install sqlfluff && python -m pip install 'sqlfluff-plugin-sparksql-upgrade @ git+https://github.com/holdenk/spark-upgrade#subdirectory=sql`. You can see which Spark specific rules are installed with `sqlfluff rules |grep -i spark` and run the entire fix set with `sqlfluff --fix mysqlhere.sql`.

Some good ways to mitigate the risk with SQL migrations is to create small sampled golden set data for side-by-side runs.

On the plus side SQL jobs tend to be well suited to side-by-side runs to verify correctness, which we will discuss more in the next section.

Verifying Correctness and Performance

In addition to the testing, which you hopefully already used early in your migration to suggest what areas you might want to change, a good practice to verify both the correctness and performance of the new pipeline is to run it side-by-side with the old pipeline.

NOTE

The results of your pipeline may not be directly comparable, for example an ML pipeline is not expected to produce the same output on multiple runs. Consider adding an (optional) step which writes out the data before training the model. For pipelines with random numbers (e.g. train/validate splits), having a configurable random seed for testing purposes can help.

The first step in most cases is to parameterize your pipelines so you configure their output tables or use Iceberg branching / staging as in this example to produce different versions (which uses a custom listener to log out the outputs). Then you can kick off two instances of your pipeline (new and old). The final result can then be compared with a combined diff or join as shown in Example 2-4.

Example 2-4. Compare the results of two pipelines (from *https://github.com/holdenk/spark-upgrade*)

```
```

The above example is a bit complicated, since it tries to handle floats and other inexact comparisons.

NOTE

Tooling here is under active development, and hopefully by the time you read this book some parts of the validation will be able to be automated. While we are biased, we encourage you to check out Holden’s “spark-upgrade” & “spark-validator” repos to see the latest.

Conclusion

Keeping your code updated with new versions of Apache Spark is an important part of taking advantage of performance improvements. While parts of these upgrades can be automated, upgrading still requires substantial manual work. The Spark Upgrade tool and ScalaFix can help with upgrading to new versions of Spark and Scala respectively, but they don’t provide any automation around other libraries you will likely need to upgrade as a result. Even for the Spark APIs, if you’re using things which are less common (for example accumulators with custom data types), these tools are unlikely to have a rule to help with that migration.[4] To keep your code “high performance” you will need to manually put in the effort to take advantage of new APIs after you have upgraded, and we’ll look at some of these new APIs in the next chapter.

[1] Tests for new functionality are more likely to be completed for merging initial code than for merging additional language support.

[2] Each SQL vendor has their own special “flavour” of SQL and they generally do their best to maintain backwards compatibility with that flavour avoiding the need for such large migrations. Though migrations are a natural opportunity to consider changing vendors, these proprietary tools do encourage vendor stickiness.

[3] UPMis not currently even available outside of Facebook :((but we hope!) are comparatively early stage.

[4] Community contributions welcomed!

Chapter 3. Going Beyond Scala

Working in Spark doesn’t mean limiting yourself to Scala or even limiting yourself to the JVM, or languages that Spark explicitly supports. There are many ways to use Spark with different languages, including remote procedure call systems like Spark Connect, JVM interop options like Java Native Interface (JNI), Java Native Access (JNA), or Java Native Runtime (JNR), and pre-built wrappers over these (like PySpark or SparklyR). This chapter will discuss the performance considerations of using other languages in Spark, and how to work with existing libraries effectively. You will learn the performance trade-offs of using different languages and how you can use common Spark accelerators to improve your existing pipelines with minimal changes.

Spark’s language interoperability can be thought of in two groups: one is the worker code inside of your transformations (e.g., the lambda’s inside of your maps), and the second is being able to specify the transformations on RDDs/`Datasets` (e.g., the driver program).

Spark has first-party APIs, or built-in support for APIs, to write driver programs and worker code in R, Python, Scala, and Java. The first-party languages share much of the same design, making it easier to reason about their performance. However, even if support is first-party does not mean it will be better. In some cases, third-party bindings have taken interesting work to minimize overhead that has not been implemented in the first-party languages, including JavaScript.[1] Others follow a more traditional implementation, like Julia, C#, and F#. Spark Connect extends this support to a wide variety of languages, albeit with a smaller API scope coverage.

Often, the language you will choose to specify the code inside of your transformations will be the same as the language for writing the driver program, but when working with specialized libraries or tools (such as CUDA[2]) Spark supports a range of languages on the driver, and you can use an even more comprehensive range of languages inside your transformations on the workers. We will discuss the design behind language support and how the performance difference can impact your work.

On the worker side, the Spark worker is always running in the JVM, and if necessary, will start another process for the target and copy the required data and result. This copying can be expensive, but Spark's DAG of dependencies and clever pipelining minimize the amount of data copying that needs to occur. The techniques that the different language APIs use for interfacing their worker code is similar to the same methods you can use to call your custom code regardless of the language of your driver.

There are many ways to go outside the JVM, ranging from Java Native Interface (JNI), Unix pipes, or interfacing with long-running companion servers over sockets. These are the same techniques used inside Spark's internals when interfacing with other languages. For example, JNI is used to call some linear algebra libraries, and Unix pipes are used to interface with Python code on the workers. The most efficient solution often depends on whether multiple transformations will need to be evaluated, the environment and language setup cost, and the computational complexity of the transformations. Regardless of which specific approach you choose to integrate other languages outside the JVM, these all require copying your data from the JVM to the runtime of your target language.

WARNING

Regardless of how you go outside of the JVM, when you introduce additional languages or processes there is more chance for container out-of-memory exceptions as the JVM may have a partial view of the memory used. These container out-of-memory exceptions can be difficult to debug and may only be visible through exit code 137.

New Spark accelerators, which primarily execute outside of the JVM, exist to run faster than Spark’s built-in tools. Some of these accelerators can reduce the “in and out of the JVM” cost by also implementing the read and write stages, so the data is first loaded outside of the JVM or written without even touching the JVM. In most cases, these optimizers will still have to transfer some of the records to and from the JVM. Still, the general performance improvement of using specialized instructions or hardware (like GPUs or ASICS like SpeedData’s) can dwarf this relatively small cost.

Languages with JVM implementations, like Kotlin or Java, can avoid the expensive copy of the data from the Spark worker to the target language. Some languages take a mixed approach, like the Eclair JS project (see [Link to Come]), which executes the worker inside of the JVM but leaves the driver program outside of the JVM. While there is, of course, some overhead in having the driver program outside of the JVM, the amount of data that needs to be passed between the Scala driver and target driver is much smaller compared to the amount of data processed by even just one of the workers.

Beyond Scala within the JVM

This section will look at how to access the Spark APIs from different languages within the JVM and some of the performance considerations of going outside of Scala. Even if you are going outside of the JVM, it is helpful to understand this section since the non-JVM languages often depend on the Java APIs rather than the Scala APIs.

Working in other languages doesn’t always mean moving beyond the JVM and staying within the JVM can have many performance benefits—mostly from not having to copy data. While you don’t necessarily need special bindings or wrappers to access Spark outside of Scala, calling Scala code can be difficult in other languages. Spark supports Java 8 lambdas for use within transformations, and users with older versions of the JDK can implement the corresponding interface from `org.apache.spark.api.java.function`. Even when data doesn’t need to be copied, working in a different language can have small yet important performance considerations.

The difficulty with accessing the Scala APIs is especially true for accessing functions with class tags[3] or using functionality provided through implicit conversions (such as all of the Double and `Tuple-specific` functionality on RDDs). For functionality that depends on implicit conversions, equivalent classes are often provided along with explicit transformations to these concrete classes. For functions that rely on class tags, “fake” class tags (e.g., `AnyRef`) can be supplied. These are normally provided automatically by wrappers for the Java API. Using the concrete class instead of the implicit conversion generally doesn’t add any overhead, but the fake class tags can limit some of the compiler optimizations.

The Java API is kept quite close to the Scala API in terms of features, with only the occasional functionality or Developer API not being available. Support for other JVM languages, like Clojure’s sparkling, is done using the Java APIs instead of calling the Scala APIs directly. Since most of the language bindings, even non-JVM languages like Python and R, go through the Java APIs, it is helpful to understand the Java APIs.

The Java APIs closely resemble the Scala APIs, while avoiding depending on class tags or implicit conversions. The lack of implicit conversions means that rather than automatically converting RDDs containing Tuples or Doubles to special classes with additional functions, explicit function conversions must be used (such as `mapToDouble` and `mapToPair`). These functions are only defined on Java RDDs meaning these special types are simply wrappers of Scala RDDs, which maintains interoperability. These special functions also return different types, such as `JavaDoubleRDD` and `JavaPairRDD`, which have the functionality that is provided by the implicit conversions in Scala.

Let’s revisit the canonical word count example using the Java APIs shown in Example 3-1. Since calling Scala APIs from Java can sometimes be convoluted, Spark’s Java APIs are primarily implemented in Scala while hiding the non-Java compatible components (namely class tags and Scala specific types). This allows the Java wrappers to be a very thin layer, consisting of only a few lines on average, with very little reimplementation required.

Example 3-1. Java Word count example

```
import scala.Tuple2;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext

import java.util.regex.Pattern;
import java.util.Arrays;

public final class WordCount {
  private static final Pattern pattern = Pattern

  public static void main(String[] args) throws E
    JavaSparkContext jsc = new JavaSparkContext()
    JavaRDD<String> lines = jsc.textFile(args[0])
    JavaRDD<String> words = lines.flatMap(e -> Ar
                                              patte
```

```
      JavaPairRDD<String, Integer> wordsIntial = wo
        e -> new Tuple2<String, Integer>(e, 1));
    }
  }
```

Sometimes, you may want to convert your Java RDDs to Scala RDDs or vice versa. Most commonly, this is for libraries that require or return Scala RDDs. Still, sometimes core Spark functionality may not yet be available in the Java API, and converting your RDD to a Scala RDD is an easy way to access the new functionality.

If you have a Java RDD you want to pass to a Scala library expecting a regular Spark RDD, you can access the underlying Scala RDD with `rdd()`. Most often, this is sufficient to pass the resulting RDD to whichever Scala library you need to call. (Some notable exceptions are Scala libraries that depend on implicit conversions of the contents of the RDD or class tag information.) In this case, writing a small wrapper in Scala can be the simplest way to access the implicit conversions. If a Scala shim is out of the question, explicitly calling the corresponding function on the JavaConverters object, construct a fake class tag.

To construct a fake class tag you can use `scala.reflect.ClassTag$.MODULE$.AnyRef()` or get the actual class tag with `scala.reflect.ClassTag$.MODULE$.apply(CLASS)` as illustrated in Examples Example 3-2 and Example 3-3.

Moving from a Scala RDD to a Java RDD often requires more class tag information than most Spark libraries. This is because, while the different `JavaRDDs` expose public constructors that take Scala RDDs as arguments, these are intended to be called from within Scala and, therefore, expect class tag information.

If you are in a mixed language project or library, consider constructing the Java RDD in the Scala, where the class tag information is more easily available.

Fake class tags are most commonly used in generic or templated code where you don't know the exact types at compile time. Using fake class tags often works, although some specialization may be lost in the Scala side; very occasionally, the Scala code depends on correct class tag information. In this case, you *must* use an accurate class tag. In most cases, using a real class tag is not substantially more effort and can offer performance advantages, so use them when possible.

Example 3-2. Java/Scala RDD interoperability with fake class tag

```
    public static JavaPairRDD wrapPairRDDFakeCt(
      RDD<Tuple2<String, Object>> rdd) {
      // Construct the class tags by casting AnyRef
      // with generic or templated code where we ca
      // class tag as using fake class tags may res
      ClassTag<Object> fake = ClassTag$.MODULE$.Any
      return new JavaPairRDD(rdd, fake, fake);
    }
```

Example 3-3. Java/Scala RDD interoperability

```
    public static JavaPairRDD wrapPairRDD(
      RDD<Tuple2<String, Object>> rdd) {
      // Construct the class tags
      ClassTag<String> strCt = ClassTag$.MODULE$.ap
      ClassTag<Long> longCt = ClassTag$.MODULE$.app
      return new JavaPairRDD(rdd, strCt, longCt);
    }
```

Both the Spark SQL and the ML pipeline APIs are mostly unified between Scala and Java. There are still Java-specific helper functions in which the equivalent Scala function is complex to call. Some examples of this are the various numeric functions, like `plus`, `minus`, etc., on `Column` as the overloaded `Scala` equivalents (&plus, -) cannot be easily accessed. Rather than having `JavaDataFrame` and a `JavaSQLContext`, the methods required for Java access are available on the regular `DataFrame` and `SQLContext`. This can be somewhat confusing, as some of the methods that will appear in the JavaDoc may not be usable from Java, but in those cases, similarly named functions will be provided to be called from Java.

Java UDFs, and by extension most other non-Scala languages, require specifying the return type of your function as it can’t be inferred in the same way it is done in Scala. A simple Java UDF is shown below in Example 3-4.

Example 3-4. Sample Java UDF

```
    sqlContext.udf()
      .register("strlen",
                (String s) -> s.length(), DataTy
```

While the types required by the Scala and Java APIs are different, for the most part, the Java collection types can be wrapped without requiring an extra copy. For iterators, the wrap conversion can be done lazily as the elements are accessed, allowing Spark to spill the data as needed (as discussed in [Link to Come]). This is especially important since, for many simple operations, the cost of copying the data can quickly dominate the actual computation required.

Custom Code Beyond Scala, and Beyond the JVM

Going beyond the JVM greatly opens up the scope of different languages you can work in. However, in its current architecture, going outside of the JVM in Spark—especially on the workers—can involve a substantial performance cost of copying data on worker nodes between the JVM and the target language. For complex operations, the cost of copying the data is relatively low, but for simpler operations, the cost of copying the data can easily double the computation cost. The first non-JVM language to be directly supported inside of Spark is Python, and its API and interface have become a model that other non-JVM languages have based their implementations on.

How PySpark Works

PySpark connects to JVM Spark using a mixture of pipes on the workers and Py4J, a specialized library for Python/Java interoperability, on the driver. This relatively simple architecture, shown in Figure 3-1, hides many complexities involved in making PySpark work. One of the bigger challenges is that even once the data has been copied from the Python worker to the JVM, it isn't in a form the JVM can easily parse. This requires special handling on both the Python worker and Java to ensure sufficient information for things like partitioning is available in the JVM.

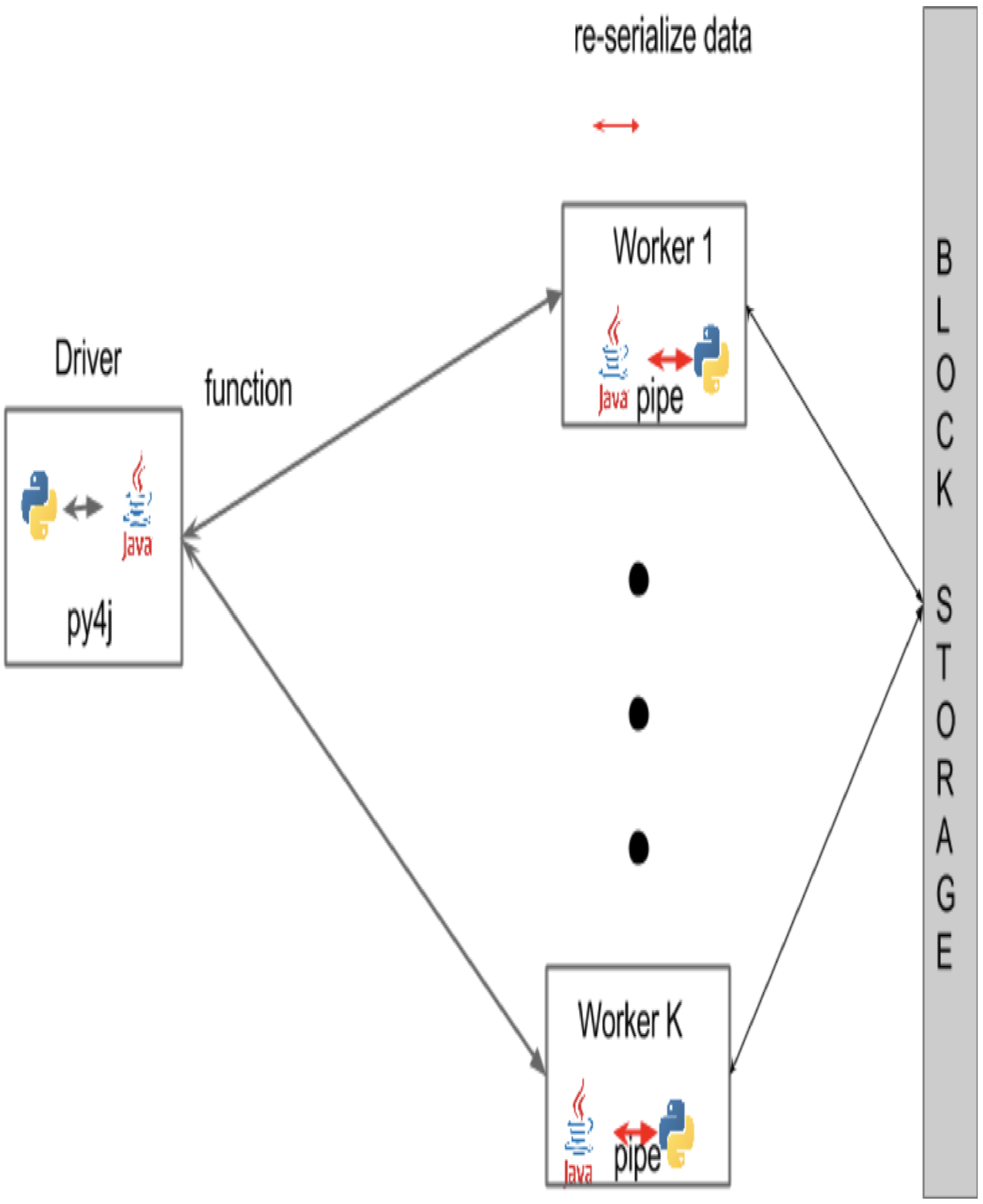

Figure 3-1. PySpark diagram

After the initial reading from persistent storage (like HDFs or S3) and between any shuffle, the data on the workers needs to be passed between the JVM and Python.

PySpark RDDs

Transferring the data to and from the JVM and starting the Python executor has significant overhead. Using the `DataFrame/Dataset` API avoids many of the performance challenges with the PySpark RDD API by keeping the data inside the JVM for as long as possible.

Copying the data from the JVM to Python is done using sockets and pickled or arrow encoded objects. A more general version of this, for talking to programs in other languages, is available through the `PipedRDD` interface illustrated in “Using Pipe and Friends”.

Since piping the data back and forth for each transformation would be expensive, PySpark pipelines Python transformations inside of the Python interpreter when possible, so a `filter` then a `map` will be chained together on the iterator of Python objects using a specialized `PipelinedRDD`. Even when the data has to be shuffled and PySpark is unable to chain our transformations inside of a single worker VM, the Python interpreter is capable of being reused so the interpreter startup overhead doesn’t further slow us down.

This is only part of the puzzle. Normal `PipedRDDs` work on `Strings`, which can’t easily be shuffled since there is no inherent key. The approach taken in PySpark, and mirrored in many other language bindings, is a special `PairwiseRDD` in which the key must be a long and is only deserialized with custom Scala code to parse the Python value. This deserialization is not overly expensive, but does serve to illustrate that for the most part, Spark Scala treats the results of Python as opaque bytes arrays.

DataFrames and Datasets generally use some similar variation of ArrowPythonRunner, which as the name implies, uses Arrow to move the data back and forth.

Since there is some overhead associated with serialization and deserialization, PySpark uses a batch serializer, and this can occasionally result in unexpected effects (like when repartitioning PySpark will not split up things in the same batch).

For all its simplicity this approach to integrating works surprisingly well, with the majority of operations on Scala RDDs available in Python. Some of the more difficult places are interacting with libraries, such as MLlib, and loading and saving from different sources.

Interacting with different formats is another restriction, as much of Spark’s load/save code is based on Hadoop’s Java interfaces. This means that any data loaded is initially loaded into the JVM and then transferred to Python.

For interacting with MLlib, generally two approaches have been taken: either a specialized data type is used in PySpark with equivalent Scala decoders, or the algorithm is reimplemented in Python. These problems are avoided with Spark ML, which uses the `DataFrame/Dataset` interface that generally keeps the data stored in the JVM.

PySpark DataFrames and Datasets

`DataFrames` and `Datasets` avoid many of the performance downsides of the Python RDD API by keeping the data inside the JVM for as long as possible. The same benchmark we did to illustrate `DataFrame`'s general improvement over RDDs ([Link to Come]) shows a greater difference when rerun in Python (Figure 3-2).

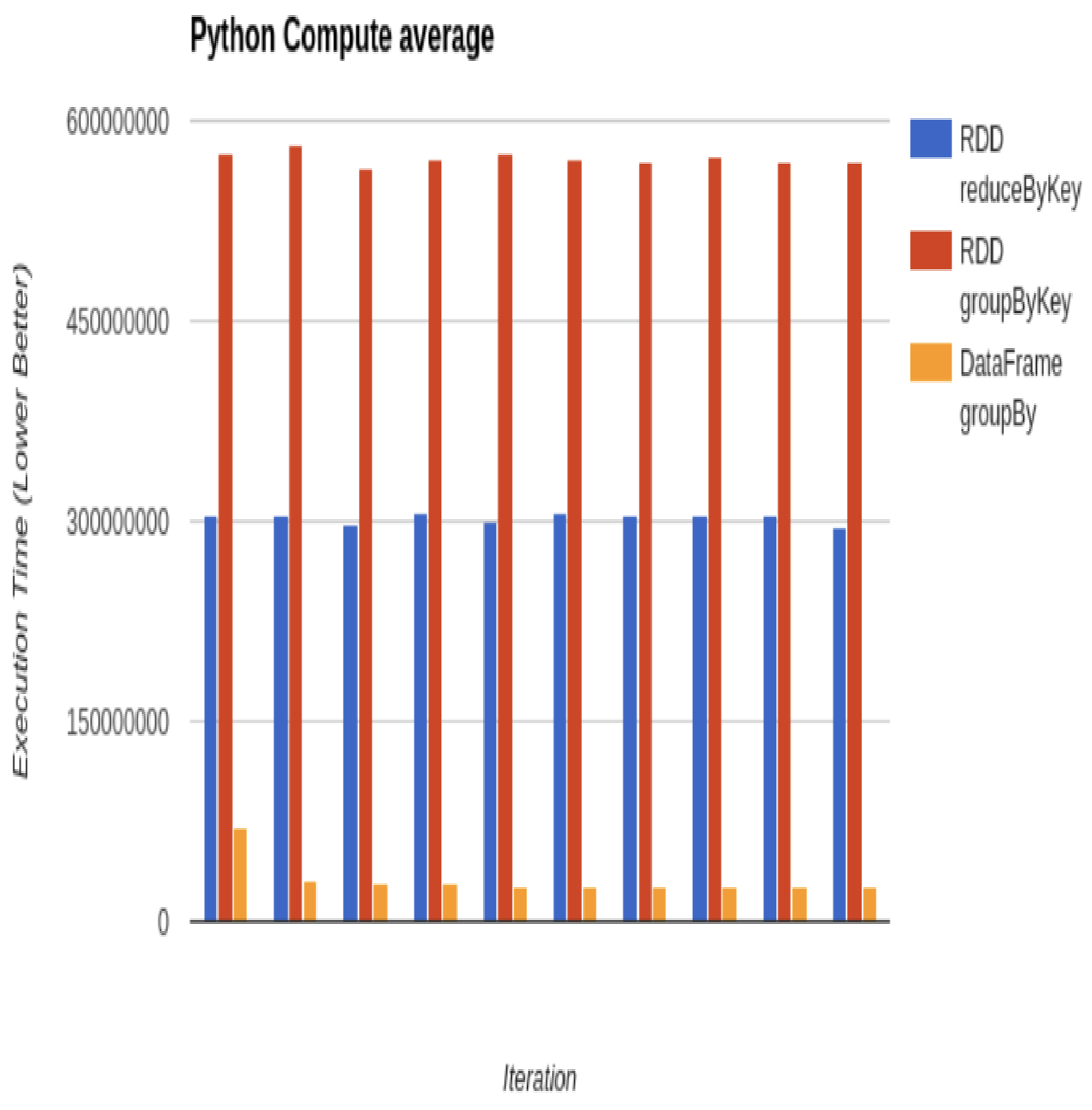

Figure 3-2. Spark SQL performance in Python

For many operations on `DataFrame`s and `Dataset`s, the data may never actually need to leave the JVM, although using Python UDFs, UDAFs, or lambdas naturally requires transferring some of the data to the

JVM. This results in a simplified architecture diagram for many operations, which instead of Figure 3-1, looks like Figure 3-3.

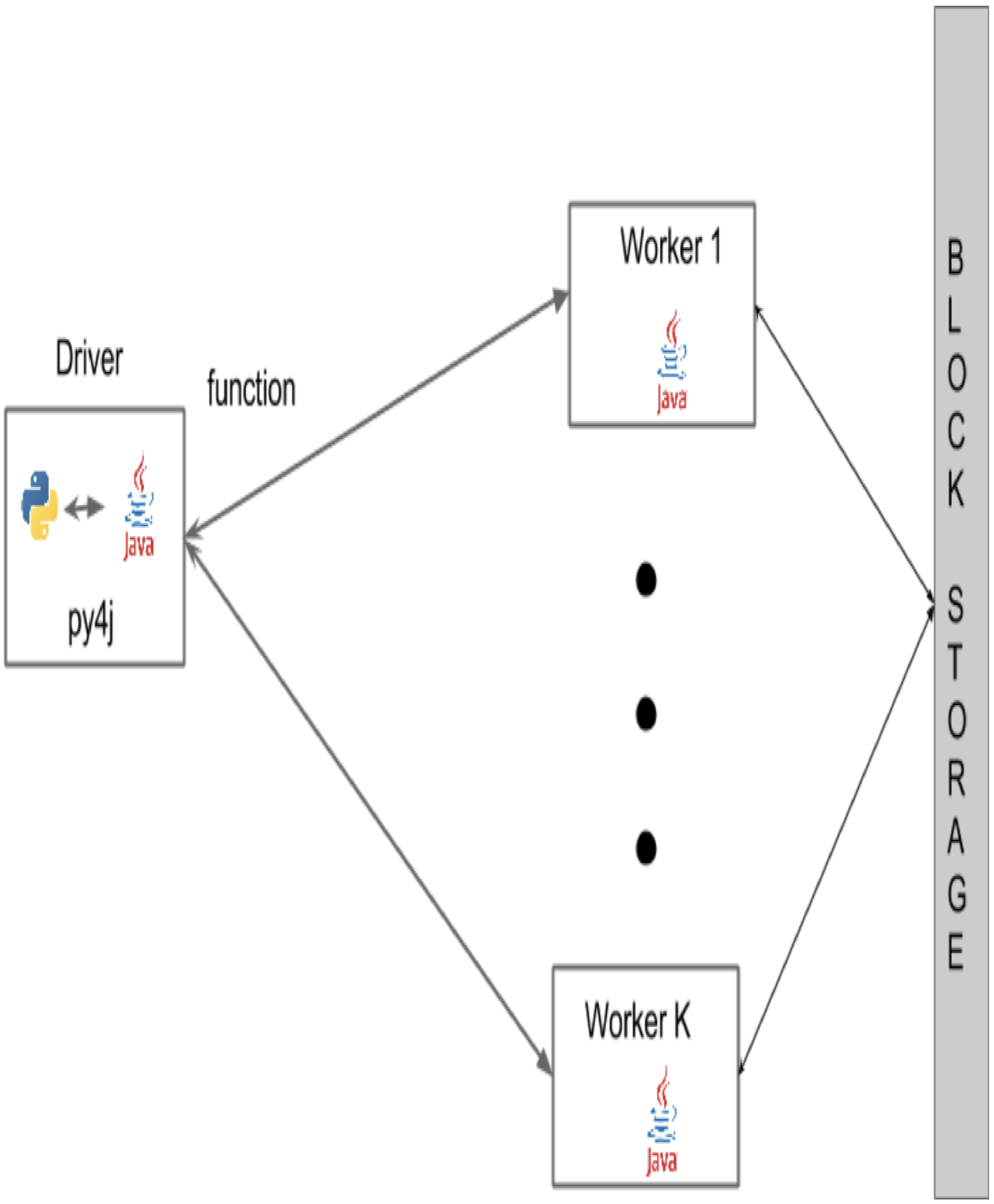

Figure 3-3. PySpark SQL diagram

PySpark doesn’t use Jython as most Python users need access to libraries like numpy, scipy, and pandas, which do not work well in Jython.[4]

Pandas API on Spark

If you are more familiar with the Pandas API or need to port existing pandas code to Spark, you can consider using Spark’s implementation of the Pandas APIs. Under the hood Pandas on Spark is implemented in Spark, so everything you’ve learned about Spark performance will generally apply to Pandas on Spark.

Not all of Panda’s APIs are supported in the Spark version. Some of these are intentional, like not implementing iter functions which would require serial processing, and others are missing because Panda’s is also under active development. Additionally your intuition around which functions are available on local collections and not on distributed collections can serve as a guide for what functions might be intentionally left out of Pandas API on Spark.

User Defined Functions in PySpark with and without Arrow

You can still use custom Python code with DataFrames by creating user-defined functions. Arrow is generally faster for serialization and deserialization in both Java and Python. Spark’s Arrow accelerated UDFs also offer the ability to perform vectorized operations instead of row-by-row.

The classic “row-by-row” UDF is shown below:

Example 3-5. Simple Python UDF

```
@udf("long")
def classic_add1(e: int) -> int:
    return e + 1
```

When possible you should use the Arrow accelerated UDFs as the serialization and deserialization cost is much lower. To take the most advantage of arrow-accelerated UDFs you switch from working on individual rows/elements to working on batches, which can be Pandas Series, DataFrames.

Most regular UDFS (applied as part of select, filter, etc.) take and return Pandas Series.

Example 3-6. Arrow Python UDF

```
@pandas_udf("long")
def pandas_add1(s: pd.Series) -> pd.Series:

    # Vectorized operation on all of the elems i
    return s + 1
```

On the other hand if your UDF takes or returns nested structures, that column (or return value) will be a pandas data frame.

Example 3-7. Python UDF with nested structures

```
@pandas_udf("long")
def pandas_nested_add1(d: pd.DataFrame) -> pd.Se
    # Takes a struct and returns the age elem + 
    # to update (e.g. return struct) we could up
    return d["age"] + 1
```

UDAFS (user defined aggregate functions), which are those used with windows, groupBy, and so on take series and return scalar values (think long/string).

Example 3-8. Python user defined aggregate function

```
@pandas_udf("long")
def pandas_sum(s: pd.Series) -> int:
    return s.sum()
```

It’s important to note that these batches that your UDF will be passed may be smaller than the entire partition. This means that even your batched your UDF may be invoked multiple times per partition. If your UDF has some state which needs to be initialized (think DB connection, table, etc.) you can also work on “batches of batches” by taking iterators of the underlying type.

Example 3-9. Batches of Batches Arrow Python UDF

```
@pandas_udf("long")
def pandas_batches_of_batches(t: Iterator[pd.Ser
    my_db_connection = None  # Expensive setup lo
    for s in t:
        # Do something with your setup logic
        if my_db_connection is None:
            # Vectorized operation on all of the
            yield s + 1
```

These UDFs allow you to use your custom Python code (and any libraries you need) in Spark.

It’s also technically possible to call Python UDFS from Scala or even JDBC by launching PySpark and then having PySpark start your Scala’s main function post registration. This is kind of hacky but potentially useful for accessing unique Python libraries. The next section will dig into how to more broadly call into Java & Scala from Python.

Accessing the backing Java objects and mixing Scala code

An important implication of the PySpark architecture is that many of Spark's Python classes simply exist as wrappers to translate your Python calls to the JVM.

If you work with Scala/Java developers and you wish to collaborate, preexisting wrappers won't exist to call your own code—but you can register Java/Scala UDFs and then use them from Python. You can do this through the `registerJavaFunction` utility on the `sqlContext`.

Sometimes these wrappers don't do everything you need, and since Python doesn't have strong protections around accessing private methods, you can jump directly into the JVM. The same techniques can be used to call your own JVM code, and with a bit of work translate the results into Python objects.

While the Py4J API is accessible, these techniques depend on implementation details of PySpark, and these implementation details may change between releases.

Thinking back to [Link to Come], we suggested that it was important to use the JVM version of `DataFrame`s and RDDs to cut the query plan. This is a workaround for when a query plan becomes too large for the Spark SQL optimizer to process, by putting an RDD in the middle the SQL optimizer can't see back past the point where the data is in an RDD. While you could accomplish the same thing using public Python APIs, you would lose much of the advantage of `DataFrame` s as the entire data would need to be round-tripped through the Python workers. Instead, by using some of the internal APIs, you can cut the lineage from Python while keeping the data in the JVM (as shown in [Link to Come]).

Example 3-10. Cut large DataFrame query plan with Python

```
def cutLineage(df):
    """
    Cut the lineage of a DataFrame - used for ite

    .. Note: This uses internal members and may b
    >>> df = rdd.toDF()
    >>> cutDf = cutLineage(df)
    >>> cutDf.count()
    3
    """
    jRDD = df._jdf.toJavaRDD()
    jSchema = df._jdf.schema()
    jRDD.cache()
    session = df.sparkSession
    javaSparkSession = session._jsparkSession
    newJavaDF = javaSparkSession.createDataFrame(
    newDF = DataFrame(newJavaDF, session)
    return newDF
```

In general, the convention for most python objects is _j[shortname] to access the underlying Java version. So, for example, the `SparkContext` has `_jsc` to get at the underlying Java `SparkContext`. This is only available on the driver program, so if any PySpark objects are sent to the workers you won't be able to access the underlying Java component and large parts of the API will not work.

The Python APIs generally wrap Java versions of the API rather than directly wrapping the Scala versions.

If you want to access a JVM Spark class that does not already have a Python wrapper, you can directly use the Py4J gateway on the driver. The SparkContext contains a reference to the gateway in `_gateway`. Arbitrary Java objects can be accessed with `sc._gateway.jvm.[fulljvmclassname]`.

Py4J depends heavily on reflection to determine which methods to call. This is normally not a problem, but can become confusing with numeric types. Attempting to call a Scala function expecting a Long with an `Integer` will result in an error message about not being able to find the method, even though in Python the distinction normally would not matter.

The same technique works for your own Scala classes provided they are on the class path. You can add JARs to the class path with `spark-submit` with `--jars` or by setting the `spark.driver.extraClassPath` configuration property. Example 3-11, which we used to generate Figure 3-2, is intentionally structured to use the existing Scala code to generate the performance testing data.

Example 3-11. Calling non-Spark JVM classes with Py4J

```
    sc = sqlCtx._sc
    javaSparkSession = sqlCtx._jsparkSession
    jsc = sc._jsc
    scalasc = jsc.sc()
    gateway = sc._gateway
    # Call a java method that gives us back an RI
    # While Python RDDs are wrapped Java RDDs (ev
    # different, so we can't directly wrap this.
    # This returns a Java RDD of Rows - normally
    # return a DataFrame directly, but for illust
    # with an RDD of Rows.
    java_rdd = gateway.jvm.com.highperformancespa
        scalasc, rows, numCols
    )
    # Schemas are serialized to JSON and sent bac
    # Construct a Python Schema and turn it into
    schema = StructType(
        [StructField("zip", IntegerType()), Struc
    )
    jschema = javaSparkSession.parseDataType(sche

    # Convert the Java RDD to Java DataFrame
    java_dataframe = javaSparkSession.createDataF
```

```
        # Wrap the Java DataFrame into a Python DataF
        python_dataframe = DataFrame(java_dataframe, 
        # Convert the Python DataFrame into an RDD
        pairRDD = python_dataframe.rdd.map(lambda row
        return (python_dataframe, pairRDD)
```

Attempting to use the Py4J bridge inside of your transformations will fail at runtime.

While many of the Python classes are simply wrappers of Java objects, not all Java objects can directly be wrapped into Python objects and then used in Spark. For example, objects in PySpark RDDs are represented as pickled strings, which can only be easily parsed in Python. Thankfully, `DataFrame`s are standardized between the languages, so provided you can convert your data into a `DataFrame`, you can then wrap it in Python and use it directly as a Python `DataFrame` or convert the Python `DataFrame` to a Python RDD.

Scala UDFs and UDAFs can be used from Python without having to go through the Py4J API.

TIP

With the increased popularity of machine learning and Spark, special logic has been added to more quickly get data into Tensorflow, this will be covered more in the next chapter on using Spark for machine learning.

PySpark dependency management

Often a large part of the reason one wants to use a language other than Scala is for the libraries that are available with that language. In addition to language-specific libraries, you may need to include libraries for Spark itself to use, especially when working with different data formats. There are a few different options for using both Spark-specific and language-specific libraries in PySpark.

Spark Packages is a system that allows us to easily include JVM dependencies with Spark. A common reason for wanting additional JVM libraries in PySpark is support for additional data formats.

If you are working in the Scala shell you can use the `--packages` command-line argument to specify the Maven coordinates of a package you want in the shell. If you are building a Scala package you can also add any requirements to your assembly *.jar*.

For Python, you can create a Java or Scala project with your JVM dependencies and add the *.jar* with `--jar`. If you're working in the PySpark shell, command-line arguments aren't allowed, so you can instead specify the `spark.jars.packages` configuration variable.

When using Spark Packages the dependencies are automatically fetched from Maven and distributed to the cluster. If your JVM dependency is not available in Maven, you can use the same technique we discuss next for adding local Python dependencies.

Adding local dependencies with PySpark can be done at both job submission time and dynamically using the `SparkContext`. Local dependencies can be *.jar* files, for JVM requirements, or *.zip* and *.egg* for Python dependencies, which are automatically added to the `PYTHONPATH`.

Work is currently underway to allow Python Spark programs to specify required pip packages and have them auto-installed, but the proposal has not yet been accepted. See the pull request and SPARK-5929 for the status of this proposal.

For individuals working with a CDH cluster, it is now possible to easily add packages with Anaconda. Cloudera's post Making Python on Apache Hadoop Easier details how to install the packages on your cluster. To make the resulting packages accessible to Apache Spark, all you need to do is set the shell environment variable `PYSPARK_PYTHON` to `/opt/cloudera/parcels/Anaconda/bin/python` either with export in your shell profile or in your *spark-env.sh* file.

For those using Kubernetes, another option can be to create a docker container with your desired Python libraries installed. This is a great improvement over the Hadoop ecosystem days as it allows for additional customization including native code that your Python packages may depend on.

If none of the above work for your cluster configuration there are a few remaining options, but these are less ideal. The simplest, but very hacky, approach is to have your transformations explicitly import the package and on failure, perform a pip installation. Similar approaches can be done with broadcast variables or a setup map at the start of the program. Failing that, you can ask your cluster administrator to install the package system wide.

Installing PySpark

First-party languages for Spark don't require any separate installation, but as mentioned for Python packages, Python has its own mechanisms for dealing with package management.

PySpark is now published on PyPi, so you can install it with a simple `pip install pyspark==3.5.0`. Alternatively, you can also `pip install` from a the Python directory of a regular Spark distribution, which is preferable for folks who work with a customized Spark distribution. Once you have PySpark pip installed you can then start your favorite Python interpreter and import `pyspark` like any other package or start the PySpark shell with `pyspark`.

It's important to note that pip installing Spark is optional. If you wish you can run PySpark from a regular Spark setup without pip installation (although then you must use `spark-submit` or `pyspark` from the Spark bin directory).

How SparkR Works

SparkR takes a similar approach to PySpark and your learnings should transfer.

Of the directly supported languages, SparkR is the furthest away from Scala Spark in terms of feature completeness. The API documentation will give you an idea if what you are looking for is already available.

To give you an idea of what the SparkR interface looks like, the standard word count example has been rewritten in R in Example 3-12.

Example 3-12. SparkR word count

```
library(SparkR)

# Setup SparkContext & SQLContext
sc <- sparkR.init(appName="high-performance-spar

# Initialize SQLContext
sqlContext <- sparkRSQL.init(sc)


# Load some simple data
```

```
df <- read.text(fileName)

# Split the words
words <- selectExpr(df, "split(value, \" \") as w

# Compute the count
explodedWords <- select(words, alias(explode(word
wc <- agg(groupBy(explodedWords, "words"), "words


# Attempting to push an array back fails
# resultingSchema <- structType(structField("word
# words <- dapply(df, function(line) {
#   y <- list()
#   y[[1]] <- strsplit(line[[1]], " ")
# }, resultingSchema)
# Also attempting even the identity transformatio
# in Spark 2.0-preview (although works fine on ot

# Display the result
showDF(wc)
```

To execute your own custom R code you can use the `dapply` method on DataFrames as illustrated in Example 3-13. SparkR’s custom code execution support has a long way to go, as illustrated by the difficulty of attempting to perform a word count with `dapply` in Example 3-12.

Example 3-13. SparkR arbitrary code with DataFrames

```
library(SparkR)

# Setup SparkContext & SQLContext
sc <- sparkR.init(appName="high-performance-spar

# Initialize SQLContext
sqlContext <- sparkRSQL.init(sc)



# Count the number of characters - note this fai
df <- createDataFrame (sqlContext,
  list(list(1L, 1, "1"),
  list(2L, 2, "22"),
  list(3L, 3, "333")),
  c("a", "b", "c"))
resultingSchema <- structType(structField("lengt
result <- dapply(df, function(row) {
  y <- list()
  y <- cbind(y, nchar(row[[3]]))
}, resultingSchema)
showDF(result)
```

Internally `dapply` is implemented in a similar way to Python’s UDF support. As with PySpark, arbitrary non-JVM code execution with SparkR is slower than traditional Scala Spark code.

SparkR isn’t the only interface for running Spark and R together. Sparklyr is a 3rd party library, from R Studio, which is also quite popular. From a performance point of view, it shares the same underlying mechanisms as SparkR in interfacing with the JVM.

Spark on the Common Language Runtime (CLR)—C# and Friends

Microsoft’s *https://github.com/dotnet/spark*.NET for Apache Spark] provides C# bindings for working with Apache Spark. The general design is similar to that of PySpark, with the internals of `PythonRDD` instead communicating with the CLR. As with PySpark, RDD transformations involve copying the data from the JVM, and `DataFrame` transformations that don’t use UDFs in C# don’t require copying the data on the workers (or even launching the CLR). If you are curious about using CLR languages with Apache Spark you can check out the design documents and examples.

Calling Other Languages from Spark

In addition to using other languages to call Spark, we can call other languages from Spark.

Using Pipe and Friends

If there aren't existing wrappers for the language you are working with, one of the simplest options is using Spark's `pipe` interface. To use the `pipe` interface you start by converting your RDDs into a format in which they can be sent over a Unix pipe. Often simple formats like JSON or CSV are used for communicating, as lightweight libraries exist for generating and parsing these records in many languages.

Let's return to the Goldilocks example from [Link to Come]. Suppose that in addition to optimal panda porridge temperature, you also wanted to find out which pandas had been commenting on Spark PRs;[5] you might create/cook up a quick little Perl script, as in Example 3-14. Later on, if you want to use this script in Spark you can use the `pipe` command to call your Perl script from the workers. Since `pipe` only works with strings, you will need to format your inputs as a string and parse the result string back into the correct data type, as in Example 3-15.

Example 3-14. Perl script to be called from pipe

```perl
#!/usr/bin/perl
use strict;
use warnings;

use Pithub;
use Data::Dumper;


# Find all of the commentors on an issue
```

```
my $user = $ENV{'user'};
my $repo = $ENV{'repo'};
my $p = Pithub->new(user => $user, repo => $repo
while (my $id = <>) {
    chomp ($id);
    my $issue_comments = $p->issues->comments->l
    print $id;
    while (my $comment = $issue_comments->next) 
        print " ".$comment->{"user"}->{"login"};
    }
    print "\n";
}
```

Then you can call your external program using the pipe interface as shown below:

Example 3-15. Using pipe (from Scala Spark) to talk to a Perl program on the workers

```
  def lookupUserPRS(sc: SparkContext, input: RDD
    // Copy our script to the worker nodes with 
    // Add file requires absolute paths
    val distScriptName = "ghinfo.pl"
    val userDir = System.getProperty("user.dir")

    val localScript = s"${userDir}/src/main/perl
    val addedFile = sc.addFile(localScript)
    // Pass enviroment variables to our worker
    val enviromentVars = Map("user" -> "apache",
    val result = input.map(x => x.toString)
      .pipe(SparkFiles.get(distScriptName), envi
    // Parse the results
    result.map{record =>
      val elems: Array[String] = record.split(" "
      (elems(0).toInt, elems.slice(1, elems.size
    }
  }
```

TIP

Spark will not automatically copy your script to the worker machines, so if you are calling a custom program you can use the `sc.addFile` interface as in Example 3-15. Otherwise (e.g., if you are calling a systemwide program), just skip that part.

PySpark and SparkR both use specialized versions of the Piped RDDs for communication with the workers.

It is essential that you write your code to handle empty partitions since your program will be called even for empty partitions.

JNI

The Java Native Interface (JNI) is another option for interfacing with other languages. JNI can work well for calling certain C/C++ libraries, as well as other statically compiled languages like FORTRAN. While JNI doesn't exactly suffer from double serialization in the same way calling PySpark or using `pipe` does, you still need to copy your data out of the JVM and back.

This is why some libraries, such as JBLAS, implement some components inside of the JVM, since once copy cost is added, the performance benefit of native code can go away.

To illustrate how to use JNI with Spark, consider calling a very simple C function that sums all of the nonzero inputs. Its function signature is shown in Example 3-16.

Example 3-16. Simple C header

```
#ifndef _SUM_H
#define _SUM_H

int sum(int input[], int num_elem);

#endif /* _SUM_H */
```

You can write the JNI specification to call this in either Java (Example 3-17) or Scala (Example 3-18). Although the tooling for Java can be a bit simpler, there is no significant difference between them.

Example 3-17. Simple Java JNI

```
class SumJNIJava {
  public static native Integer sum(Integer[] arra
}
```

Example 3-18. Simple Scala JNI

```
class SumJNI {
  @native def sum(n: Array[Int]): Int
}
```

Manually writing wrappers takes effort. Check out SWIG to automatically generate parts of your bindings.

Once you have your C function and your JNI class specification, you need to generate your class files and from them generate the binder heading (see Example 3-19). The `javah` command will take the class files and generate headers that is then used to create a C-side wrapper.

Example 3-19. Generate header with the command-line interface

```
javah -classpath ./target/examples-0.0.1.jar \
com.highperformancespark.examples.ffi.SumJNI
```

For those of you building with SBT, Jakob Odersky’s `sbt-jni` package makes it easy to integrate your native code with your Scala project. `sbt-jni` is published as an SBT plug-in like `spark-packages-sbt`, and is included by adding an entry to *project/plugins.sbt* as shown in Example 3-20.

Example 3-20. Add sbt-jni plug-in to project/plugins.sbt

```
addSbtPlugin("com.github.sbt" %% "sbt-jni" % "1.5
```

`sbt-jni` simplifies generating the header file by adding the `javah` target to sbt, which will generate the header files and place them in *./target/native/include/*.

Once we have our header file we need to write a wrapper in C. The generated header file shouldn’t be modified, but rather imported into our shim as shown in Example 3-21.

Example 3-21. JNI C shim

```
#include "sum.h"
```

```
#include "include/com_highperformancespark_examp
#include <ctype.h>
#include <jni.h>

/*
 * Class:     com_highperformancespark_examples_
 * Method:    sum
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_com_highperformances
(JNIEnv *env, jobject obj, jintArray ja) {
  jsize size = (*env)->GetArrayLength(env, ja);
  jint *a = (*env)->GetIntArrayElements(env, ja,
  return sum(a, size);
}
```

`sbt-jni` also simplifies building and packaging native code, adding `nativeCompile`, `javah`, and `packageBin` to allow you to easily build an assembly JAR with both your native files and Java artifacts. For `sbit-jni` to build your native code (in addition to the JVM code) as well, you need to provide a Makefile. If you are starting with a new project, `nativeInit CMake` target will generate a skeleton *CMakeLists.txt* file you can use as a basis for your native build.

In our example project, we've built the native code along with the Scala code. Alternatively, especially if you plan to support multiple architectures, you may wish to create a separate package for your native code.

If your artifact is built with `sbt-jni` you can use the `nativeLoader` decorator from `ch.jodersky.jni.nativeLoader` to automatically load your native code as needed. In the example we've been working on, our library is called `libhigh-performance-spark0` so we can have it automatically loaded by adding the decorator to our SumJNI class, as in Example 3-22.

Example 3-22. Native Loader decorator

```
@nativeLoader("high-performance-spark0")
```

If you are working in Java, or just want more control, you can use `System.loadLibrary`, which takes a library name and searches `java.library.path` or `System.load` with an absolute path.

TIP

Leave off the “lib” prefix, which `loadLibrary` (and `sbt-jni`) automatically append, or you will get confusing runtime linking errors.

TIP

The Oracle JNI specification can be a useful reference.

If your native library likely isn’t packaged in your JAR, you need to make sure the JVM running the Spark worker is able to call it. If your library is already installed on the workers you can add `-Djava.library.path=...` to your `spark.executor.extraJavaOptions`.

The JVM is expanding it’s options for accessing non-JVM code beyond the traditional JNI interface. These new options can be faster, but require modern versions of the JVM (like 22 and higher) and while many of the APIs have been finalized a few remain subject to change. If you need to access non-JVM code it may be worth looking into what the latest options are.

Java Native Access (JNA)

Java Native Access (JNA) is a community-driven alternative to JNI to allow calling of native code, ideally without all of the boilerplate required by JNI. Although JNA is a community package this does not mean it is low quality; it is used by a variety of mature projects and has been used by Spark application developers. We can use JNA to call our previous example in both Scala (Example 3-23) and Java.

Example 3-23. Scala simple JNA

```
import com.sun.jna._
```

```
  object SumJNA {
    Native.register("high-performance-spark0")
    @native def sum(n: Array[Int], size: Int): Int
  }
```

It’s important to note that these JNA examples skip the requirement for writing the JNI wrapper (as in Example 3-21) and instead directly call the C function for us. While SWIG can do a good job of generating much of the JNI wrappers, for some this is a compelling reason to use JNA over JNI.

When using JNA, `jna.boot.library.path` allows you to add libraries to the search path before the system library path.

Project Panama

Comparatively new in the JDK, Project Panama offers automatic wrapper generation with jextract, as well as columnar type interfaces and shared memory, which is ideal for large datasets. Project Panama is still incubating, and its APIs may change through out the JDK Enhancement Process so while it can bring substantial benefits it’s more likely to require code changes than using “old-fashioned” JNI.

Underneath Everything Is FORTRAN

A surprising number of numeric computing libraries still have FORTRAN implementations. Thankfully many of these libraries already have Java or Python wrappers, which greatly simplify our access. These libraries often can make intelligent decisions about what operations are worth the overhead of copying our data into FORTRAN and what operations make more sense to be implemented in the host language. Not all FORTRAN code already has wrappers, and you may find yourself in a place with which you want to interface. The general process is to first create a C/C++ wrapper that exposes the FORTRAN code for Java to call, and then link the C/C++ code together with the FORTRAN code. Continuing the sum example in FORTRAN (Example 3-24), you would create a C wrapper like Example 3-25, and then follow the existing steps for calling a C library in [Link to Come].

Example 3-24. FORTRAN sum function

```
       INTEGER FUNCTION SUMF(N,A) BIND(C, NAME='
       INTEGER A(N)
       SUMF=SUM(A)
       END
```

Example 3-25. C wrapper for FORTRAN sum function

```
// Fortran routine
```

```
extern int sumf(int *, int[]);

// Call the fortran code which expects by referen
int wrap_sum(int input[], int size) {
  return sumf(&size, input);
}
```

TIP

If you like `sbt-jni` you can extend the generated CMake file to also compile your FORTRAN code.

These wrappers can also be automatically generated with programs like fortrwrap, or skipped entirely with JNA. Calling the FORTRAN function with JNA is very similar to calling the C function, as shown in Example 3-26.

Example 3-26. FORTRAN SUMF through JNA

```
import com.sun.jna._
import com.sun.jna.ptr._
object SumFJNA {
  Native.register("high-performance-spark0")
  @native def sumf(n: IntByReference, a: Array[In
  def easySum(size: Int, a: Array[Int]): Int = {
    val ns = new IntByReference(size)
    sumf(ns, a)
    }
  }
```

Calling FORTRAN code from the JVM is more difficult than calling C code. If available, it’s often better to use existing wrappers as they can make intelligent decisions about which components to execute in FORTRAN rather than in the JVM.

Getting to the GPU

GPUs are another great way of working with parallel, numeric computing problems. They are particularly effective at certain machine learning problems. One of the best options for getting to the GPU comes from using NVIDIA’s Spark RAPIDs accelerator, which has the ability to leave your data on the GPU for calling to other libraries.

Alternatively, when working with RDDs you can have Spark assign GPUs through resource profiles, and then use your GPU compatible libraries. Some people, including people at DeepMind, have also used aparapi to automate the compilation of Java code to OpenCL.

When working with Dataframes, or if you’re working with GPUs consistently throughout your job, you can configure Spark to assign GPU resources with `spark.task.resource.gpu.amount` with any decimal value, for example `0.25` meaning each task uses a quarter of a GPU.

Going Beyond the JVM with Spark Accelerators

As Spark has grown in popularity, both open-source and closed-source accelerators have been developed to allow for improving the running speed of your existing Spark code without rewriting it. Parts of these accelerators often require sticking within a limited range of supported operations, mostly Datasets & SQL without user-defined functions. The most common accelerator is Databrick's Photon, followed by spark-rapids for (GPU acceleration) and gluten, and blaze for CPU based acceleration.There is also a relatively new project, Apache Arrow's Comet DataFusion, which has some important and unique benefits.

WARNING

Accelerators often involve some slight constant increase in overhead and are therefore not well suited to frequent small queries.

At their core, all of the accelerators work by replacing how Spark's executors process data using code outside of the Java Virtual Machine. The accelerators generally either use Apache Arrow or a similar format, suited for vectorized operations that apply the same logic to many elements at once instead of one at a time. Each accelerator works by inserting an optimizer step to Spark SQL, which replaces those components of the SQL plan the optimizer can handle with a node to trigger the executor side accelerator (commonly through JNI). Each of the three leading accelerators uses a custom shuffle implementation and data readers/writers, allowing them to avoid copying data into and out of the JVM when executing queries entirely supported by the optimizer. For queries where some parts do not have a native implementation, they also have code to copy data to/from the JVM with some additional overhead involved.

WARNING

As with all of the techniques in this chapter, there is a penalty to be paid whenever you have to move data into or out of the JVM. These new optimizers, mostly, transparently fall back to the JVM when an operation (or data source/sink) is not implemented inside of the optimizer – but this also means that changing data source formats, or even just using a new not yet fully supported function, can result in huge performance degradation.

You can get a better understanding of both how much work the optimizer is able to handle, and where the data copies are occurring by looking at the Spark query plan. Each optimizer has internal alternatives for the components of the Spark plan they handle, as well as names for the inserted stages that copy data to/from the JVM.

Databrick’s Photon and Apache Arrow Comet Datafusion, are the only two accelerators that do not require static allocation between the JVM and accelerator. This is especially important since many Spark users have to spend a substantial amount of time with out of memory exceptions.

Databricks Photon

While Databricks Photon is the most popular of Spark accelerators, it is also closed source so it is more difficult to ascertain how it currently functions, but there is a sigmod paper on how Databricks implemented it. In general, Photon works by converting parts of your Spark SQL (or Dataset/DataFrame) operation into native code with vectorized operations.

TIP

As closed source, if Databricks makes changes we may not know so be careful about depending on any of the details discussed here.

Every time you see a Transition node in your Spark SQL plan with Photon enabled it means that the data is having to be copied to/from the JVM. While the optimizer does a good job organizing transformations to minimize these, it can be worth investigating if you can manually move any of them together to reduce the number of copies (for example, automatic re-ordering between join conditions is limited automatically but can often be done safely by humans).

Databricks Photon has some important memory management features that help it stand apart from the other optimizers. Photon does not require statically allocated off-heap memory; instead, it shares memory with Spark’s existing memory manager. This greatly reduces the change of container out-of-memory exceptions. In addition, Photon is integrated with Spark’s existing “spill-to-disk” infrastructure making processing partitions too large to fit in memory possible (provided that each columnar batch can fit in memory). Given how difficult memory problems can be to debug in Spark, as well as their frequency, these benefits are difficult to overstate.

An important drawback of Photon over the other accelerators is that it is impossible to add your own functions.[6] This means if your work is largely executed inside of complex non-built-in operations, Photon is unlikely to be able to offer much performance improvement.

As a commercial offering, it’s difficult to determine the project’s “health,” but it appears to continue to be a core element of Databrick’s offering, and being created by many of the same people developing Spark means it is likely to continue to be updated to new versions of Spark.

Apache Arrow Comet Datafusion

Apache Arrow Comet Datafusion (which we’ll comet from here on out), was initially contributed to the Apache Arrow project by Apple in early 2024. At the time of the writing, the early contribution lacked a number of important features and could only accelerate a few operations – but it had some of the most promising roadmaps of any of the open-source projects. Comet's unified memory management with Apache Spark makes it, while perhaps not the choice of today, a likely top contender in the future for open-source Spark acceleration.

As of the writing Comet has not yet had a release, so to try out Comet you'll need to build it from source. In practice, since most of the accelerators are not statically linked on modern system, you'd have to do this regardless.

Example 3-27. Building Comet

```
# If we don't have fusion checked out do it
if [ ! -d arrow-datafusion-comet ]; then
  git clone https://github.com/apache/arrow-dataf
fi

# Build JAR if not present
if [ -z "$(ls arrow-datafusion-comet/spark/target
  cd arrow-datafusion-comet
  make clean release PROFILES="-Pspark-${SPARK_MA
  cd ..
fi


COMET_JAR="$(pwd)/$(ls arrow-datafusion-comet/spa
export COMET_JAR
```

Once you’ve got the JARs you can then enable Comet with Spark with the following command line options:

Example 3-28. Command Line Flags to run with Comet

```
--jars ${COMET_JAR} \
--driver-class-path ${COMET_JAR} \
--conf spark.comet.enabled=true \
--conf spark.comet.exec.enabled=true \
--conf spark.comet.exec.all.enabled=true \
--conf spark.shuffle.manager=org.apache.spark.sq
--conf spark.comet.exec.shuffle.enabled=true \
--conf spark.comet.columnar.shuffle.enabled=true
--conf spark.sql.extensions=org.apache.comet.Come
```

The above example enables all of comet’s functionality, but you can use Comet without the shuffle manager. Doing this means that the data will have to flow back and forth between the JVM more frequently, but can be useful for debugging or when you are working in an environment with it’s own shuffle manager.

NOTE

If you get a class loader error with Comet shuffle manager double check you’ve specified –driver-class-path as well as –jars, or you can copy the Comet jar into your Spark distributions regular class path.

Comet does not yet have support for custom user defined functions, although it is part of the roadmap.

As of the writing of this chapter (2024), Comet+Spark is in it’s early stages it’s unique approach to memory management is very promising. Since it is built on top of the Datafusion library and based on an existing in production accelerator at Apple it will hopefully develop quickly.

Project Gluten

Project Gluten uses Apache Arrow for data representation and supports multiple native backends of execution. The two open-source backends it supports out of the box are Facebook’s Velox and Kyligence’s Clickhouse. In our experience, as of the writing of this book, Project Gluten feels rougher around the edges than the other optimizers, with more fiddling required for install. That being said it also can also be run on commodity hardware without any special license and supports a wide array of operations.

WARNING

It is very important to make sure the native build target you choose matches your setup as errors (or even corruption) may occur with native library mismatches.

Getting Started

There are pre-built JARs for Gluten, but they are only built against select versions of Ubuntu and Spark. You can check out the releases page to see if there is a pre-built JAR for your configuration, otherwise, you'll need to build either the Velox or Clickhouse enabled JAR you want.[7] Since our examples in this book run on more than just Ubuntu 18.04, we build Gluten+Velox as part of the book's CI testing and our build is as follows:

Example 3-29. Gluten + Velox Build

```
sudo apt-get install -y locales wget tar tzdata g
     llvm-dev clang libiberty-dev libdwarf-dev li
     libcurl4-openssl-dev maven rapidjson-dev lib
     libsodium-dev libsnappy-dev nasm
sudo apt install -y libunwind-dev
sudo apt-get install -y libgoogle-glog-dev
sudo apt-get -y install docker-compose
sudo apt-get install -y libre2-9 || sudo apt-get
  if [ ! -d incubator-gluten ]; then

    git clone https://github.com/apache/incubator
  fi
```

```
cd incubator-gluten
sudo ./dev/builddeps-veloxbe.sh --enable_s3=ON
mvn clean package -Pbackends-velox -Pspark-3.4
GLUTEN_JAR_PATH="$(pwd)/package/target/gluten-
```

WARNING

As of the writing, building gluten is a flaky process of trial and error outside of the pre-supported base systems.

Here we will focus on Velox, but the general principles are the same between Velox and Clickhouse.

Velox + Gluten

Enabling Gluten involves making the JAR available to all of the executors and drivers and some minor configuration. A sample set of Spark configs enabling Gluten is shown bellow Example 3-30.

Example 3-30.

```
spark.plugins=io.glutenproject.GlutenPlugin
spark.memory.offHeap.enabled=true
spark.shuffle.manager=org.apache.spark.shuffle.s

# This static allocation is one of the hardest p
spark.memory.offHeap.size=20g
```

The trickiest part is figuring out what the correct value is for `spark.memory.offHeap.size` since if you over provision the value your “wasting” memory and if you under provision you may get a container OOM error.

UDFs

You can also make your user-defined functions in C++ (and from there call other ABI-compatible languages), although this process is much more involved than writing “regular” Spark UDFs. The exact details of building Gluten + Velox UDFs are beyond the scope of this book, but if you’re familiar with C++ can find the documentation on creating UDFs as part of the Gluten project here.

NOTE

You must ensure that any native UDFS you build are suitable for deployment on your cluster. If you have a different local system than your deployment cluster, like an ARM Mac and x86 Ubuntu cluster, you’ll need to cross-compile (or remotely build) your library. Cross-compiling native libraries is beyond the scope of this book, but thankfully there are a great many resources, including Docker’s faster cross platform builds, on how to set this up.

Clickhouse + Gluten

Clickhouse is an alternative backend for Gluten, however they do not currently distribute pre-built Clickhouse + Gluten JARs which may indicate a reduced amount of testing. Clickhouse and Gluten also do not, as present, have a built in story for handling user-defined functions, so any UDFs will result in a copy to/from the JVM.

Gluten’s Future

Gluten seems to be relatively actively developed, but lags substantially behind upstream. Even a several months after Spark 3.5’s Scala 2.13 release the latest version supported was Spark 3.3 with Scala 2.12 on Ubuntu 18.04 which was already past the standard support end of life. Getting fixes, even small ones, upstreamed into Gluten appears challenging for a mixture of reasons from an incomplete local development experience to challenges with reviewer engagement. While these challenges are not unique to Gluten, many projects lag behind the latest and can be difficult to get fixes into, it’s important to keep in mind when considering a new dependency.

Gluten was recently added to the ASF as an incubating project, which is a good sign for future openness to collaboration.

Gluten also has some important performance improvements on its road map. All optimizers perform best when they can handle the initial reading and writing stages, as this avoids large data copies. At present, Gluten delegates this to Spark for many of the most popular meta stores, including

Iceberg and Delta Lake. While Gluten could be used with GPUs, we do not see any current open-source implementations. The next accelerator we'll dive into uses GPUs for its speed-up.[8]

Spark RAPIDs

Spark RAPIDs is different from the other accelerators as it requires special hardware to achieve it's speed up. However, the rest remains similar, with select tasks being run outside of the JVM, native file & memory representations, as well as a focus on columnar as opposed to row-based operations. As with Gluten and Photon, Spark RAPIDs has it's own native implementations of file and table formats and like the rest can fall back to Spark's JVM implementations when reading or writing an unsupported format.

Setting Up & Configuring Spark RAPIDs

Spark RAPIDS is released as a JAR with broad support for different Spark versions[9] which you can download from their page here. From there launching Spark w/RAPIDs is incredibly simple and you can, at it's simplest, launch with `--jars ./accelerators/rapids-4-spark_2.12-24.02.0.jar --conf spark.plugins=com.nvidia.spark.SQLPlugin`.

While you can use our general guidance with optimizers, most Spark built-ins can benefit from acceleration; the RAPIDs library also gives you a handy way to check by setting the `spark.rapids.sql.mode` property to explainOnly. In explain mode, the driver logs will log which parts of your query plan can be executed by rapids.

TIP

If you end up using RAPIDs extensively you can also include it on your global Hadoop class path or Docker base containers for faster worker startup time.

Once you've verified you can benefit from RAPIDs, you need to ensure there are GPU resources available on your Spark cluster. Each resource manager and cloud is a little different, and the cloud APIs are likely to change, so take a look at[10] the correct guide for your cluster: Yarn GPUs, NVIDIA's GPU discovery on Kubernetes or NVIDIAs cloud guide. In addition to having the GPU resources present and labeled,remember Spark can request executors of different sizes and types so not every node needs to have GPUs.

TIP

While RAPIDs needs to be loaded at start you can enable and disable it by setting spark.rapids.sql.mode to true or false. You can combine this with resource profile to selectively use GPU resources for relevant parts of your workloads.

RAPIDs has some extra configuration parameters you may wish to fine tune and they publish their own fine tuning guide.

UDFS

Spark RAPIDs is unique in it’s approach to UDFS; instead of not supporting them or requiring custom code, it has built-in transpiling for select UDFS. It supports select pandas UDFs and even some Scala and Java code. In practice, the Scala/Java UDF compilation is largely limited to operations that can be expressed without a UDF anyway, but it can make the code easier to read for those of us who like lambdas.

WARNING

The UDF transpiling is experimental and may produce slightly different results than non-transpiled UDFs. Always validate accelerated jobs.

Going Beyond RAPIDs on the GPU

One of the understated benefits of using Spark RAPIDs is that your data is already on the GPU for machine learning. The next chapter will look more at how to use Spark for machine learning, and Spark RAPIDs is one of the ways you can ensure your data is both in the correct format and location (GPU).

Project Health

Overall, Spark RAPIDs appears to be an important offering from NVIDIA, and with the core accelerators (CUDA kernels) shared between many projects it's likely to continue to be updated. That being said, given that the majority of the development comes from NVIDIA we are unlikely to see features like ROCm support added.

Application-Specific Integrated Circuits (ASICS)

While they are not production ready at the time of the writing, new ASICS have the potential for greatly accelerating Spark Dataset/SQL workflows. One of the primary vendors, speeddata, claims a "100x faster & 90% cost saving) and while you will likely not experience the idealized results (unless you're all pure SQL) they are sufficiently fast that it may be worth investigating especially if you control your hardware deployments.

Managing Memory Outside of the JVM

The more languages involved, the more chance there is for something to go wrong and accidentally exceed memory limits resulting in an out-of-memory exception (or OOM kill with exit code 137). To an extent, Python memory is managed with , although since Python often depends on native libraries that may do their own memory allocations this is not complete. You'll need to carefully manage the amount of "overhead" assigned to non-JVM tasks in all cases except Arrow DataFusion Comet & Photon.

The future (from ~2024)

Apache Arrow is supported by a growing number of languages, and it has the potential to serve as a semi-standardized bridge between languages. In the future, we hope that the accelerators and Spark are able to standardize on shared formats as well as interfaces. Some variant of Apache Arrow has the potential to be this format. While standardizing here would benefit the users, it would also reduce the “moat” for each of the accelerators, so it may not happen for business reasons.

Conclusion

Writing high-performance Spark code need not be limited to Scala, let alone the JVM (although it can certainly make things easier). Spark has a wide variety of language bindings, both built-in and third party, and can interface with even more languages using JNI, JNA, pipes, or sockets. For some operations, the cost of copying the data outside of the JVM and back can be more expensive than just doing the operation in the JVM—even with specialized libraries—so it is important to consider the complexity of your transformations before going outside of the JVM. In the next chapter you’ll see how even more of how Spark can work beyond the JVM with AI libraries.

[1] The demand for JavaScript big data did not emerge as hoped, but the EclairJS engine demonstrates an interesting and unique approach using different JS evaluation allowing for faster execution along the ideas of using Py4J which were not successful due to native libraries. It’s a fascinating design, albeit not a project that is current maintained.

[2] CUDA is a specialized language for parallel GPU programming from NVIDIA. ROCm is a similar option from AMD.

[3] Used to store type information that is erased or not known at compile time. The most common example for usage of class tags is constructing Arrays.

[4] Some early work is being investigated to see if Jython can be used to accelerate Python UDFs, which don’t depend on C extensions, but has been shelved. See SPARK-15369 for details.

[5] This is somewhat of a stretch as far as the relationship to Goldilocks goes, but you know.

[6] Outside of getting a job at Databricks.

[7] In our experience the build instructions are generally accurate, but may be missing various libraries. If you see a CMake library error during the build you likely need to install that package on your build machine

(googling the error message tends to tell you the package names on different Linux systems).

[8] Given many of their similarities, we’d love it if Spark Rapids and Project Gluten combined but we don’t see that on either of their roadmaps.

[9] Doing this inside of a single JAR is extremely difficult, so thank you to the NVIDIA team.

[10] Or have your cluster administrator take a look at.

About the Authors

Holden Karau is transgender Canadian, and an active open source contributor. When not in San Francisco working as a software development engineer at IBM’s Spark Technology Center, Holden talks internationally on Apache Spark and holds office hours at coffee shops at home and abroad. She is a Spark committer with frequent contributions, specializing in PySpark and Machine Learning. Prior to IBM she worked on a variety of distributed, search, and classification problems at Alpine, Databricks, Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a Bachelor of Mathematics in Computer Science. Outside of software she enjoys playing with fire, welding, scooters, poutine, and dancing.

Anya Bida is a cisgender American who loves unblocking individuals and teams in their data adventures. In her Developer Relations role at Prophecy.io, Anya guides data engineers and organizations to implement best practices in their Spark ecosystems. Anya previously owned Spark Operations for Salesforce Einstein as Lead SRE and held customer-facing roles at Faros.ai and Alpine Data.

Rachel Warren is a data scientist and software engineer at Alpine Data Labs, where she uses Spark to address real-world data processing challenges. She has experience working as an analyst both in industry and academia. She graduated with a degree in Computer Science from Wesleyan University in Connecticut.