Southern Data Science Conference

Apache Spark

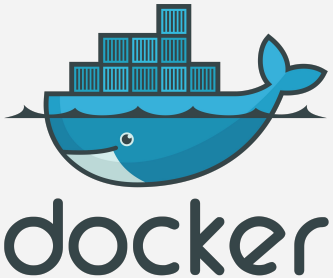
SDSC 2019
Apache Spark Bootcamp

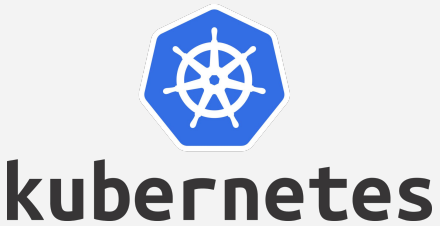
Chris Min
chrism.1202@gmail.com
https://www.linkedin.com/in/chris-min-76256170/

As a Big Data engineer,
I have been building data processing pipelines in various scales.

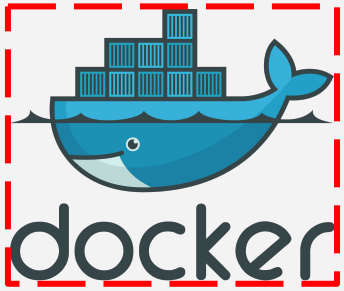
How do I scale these projects?

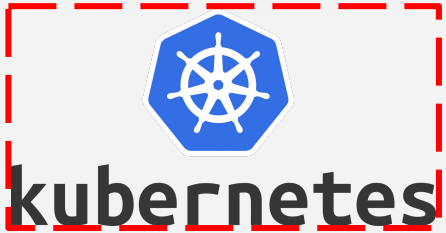
Open source BigData technologies to the rescue!





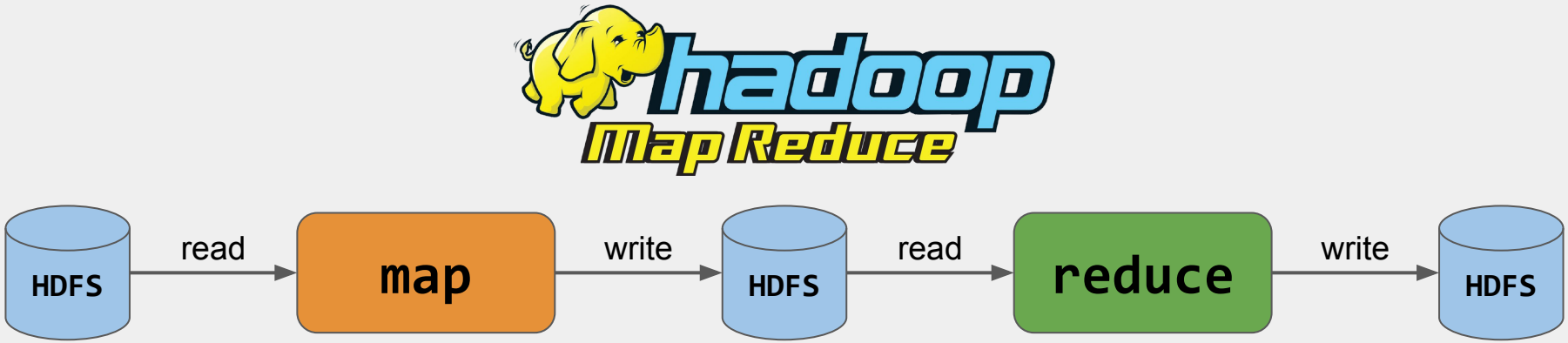
What I primarily use these days…



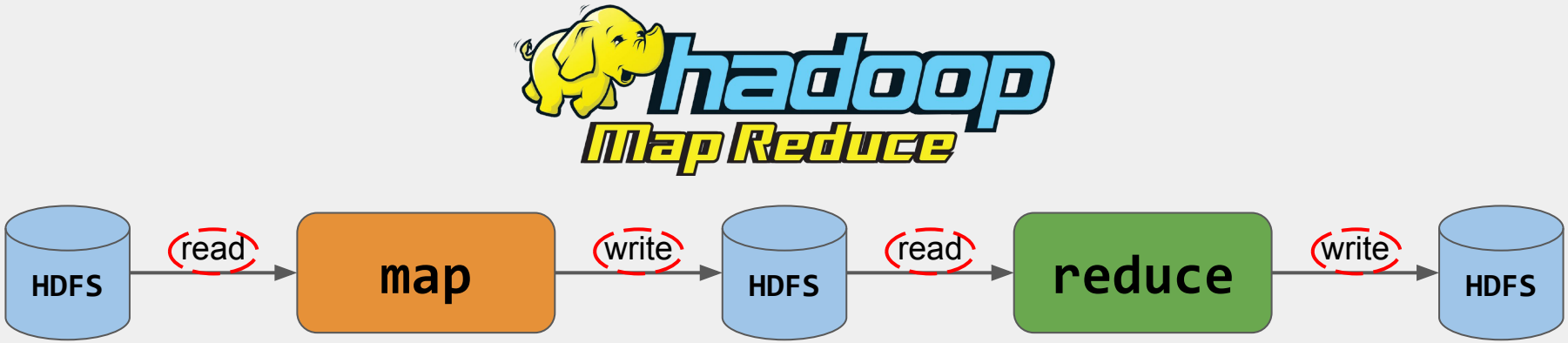



Let's journey back a few years...

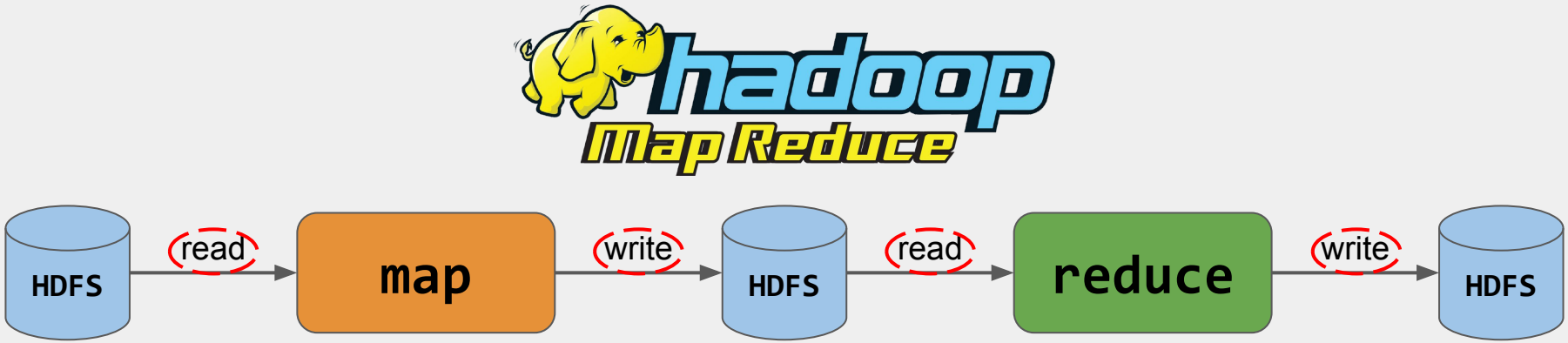
Before 2014


hadoop
Map Reduce
HDFS
read
map
write
HDFS
read
reduce
write
HDFS

Before 2014

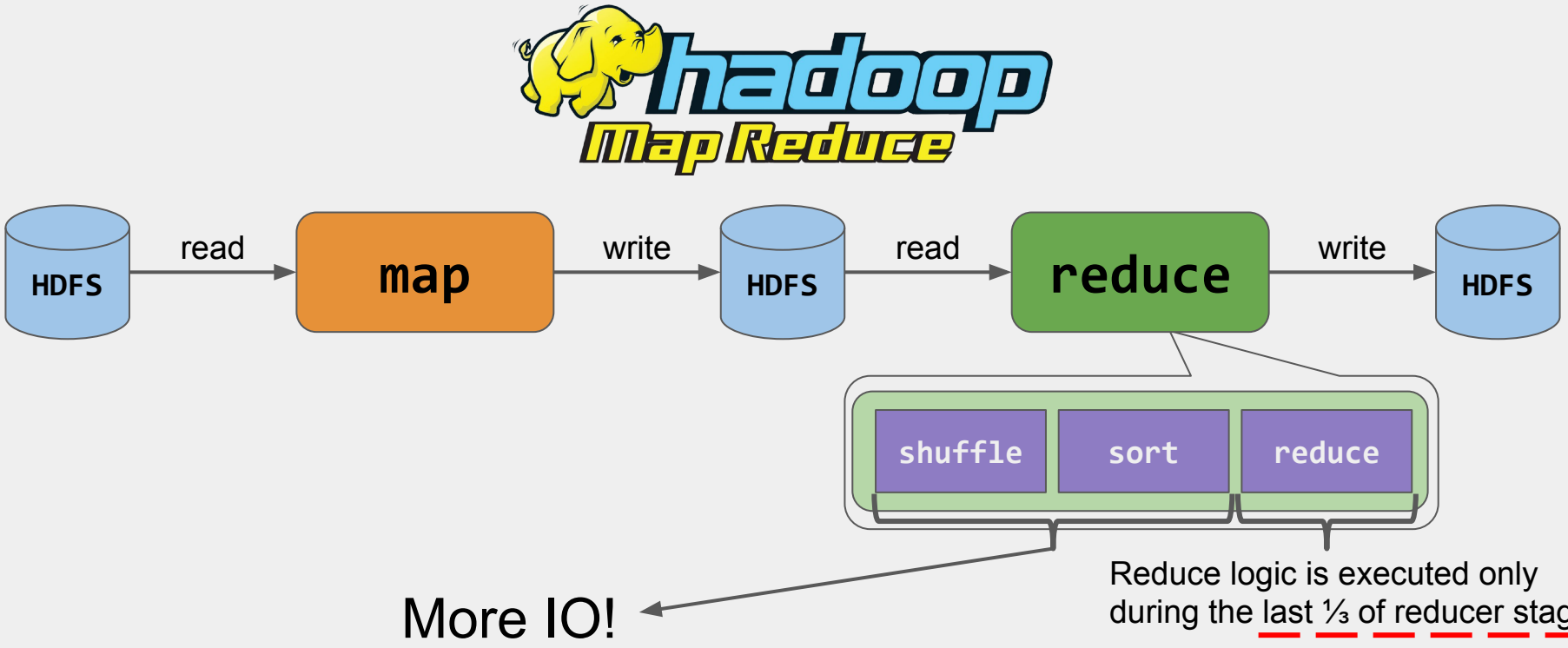

hadoop
Map Reduce
HDFS
read
map
write
HDFS
read
reduce
write
HDFS

Before 2014



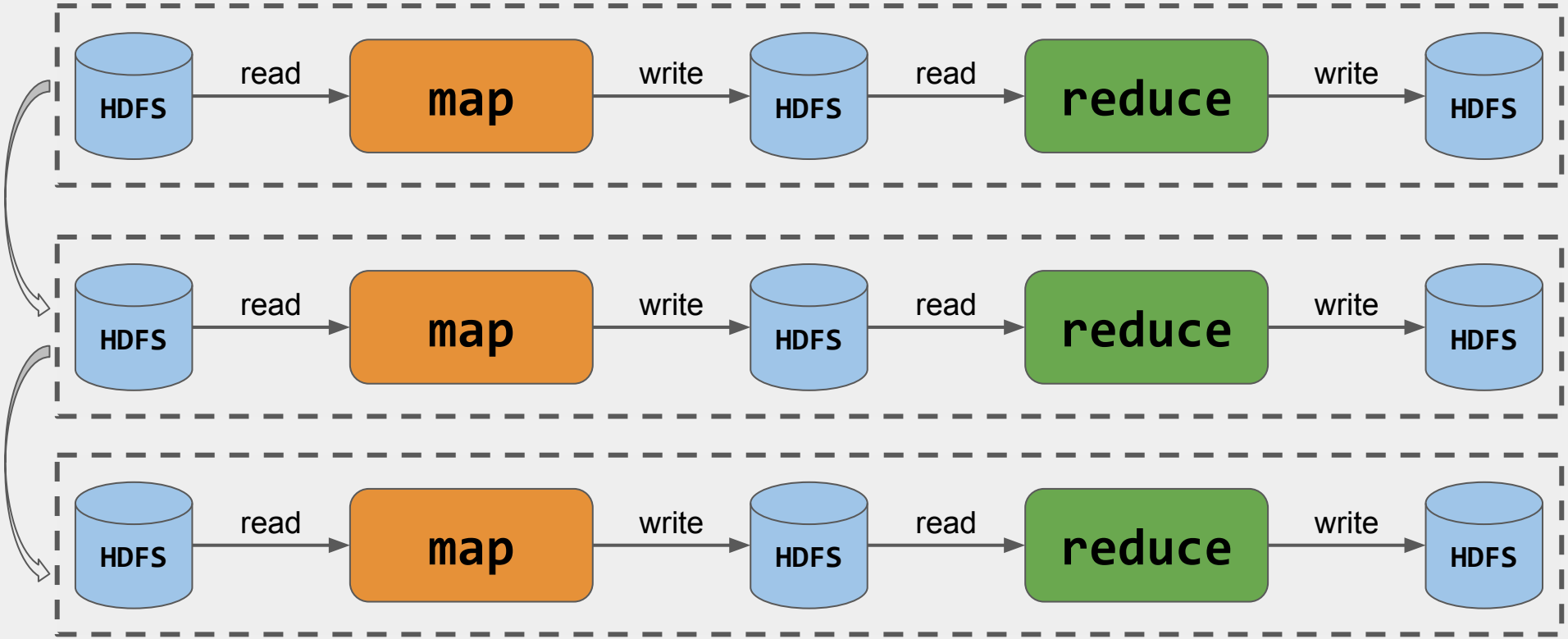
IO

Before 2014

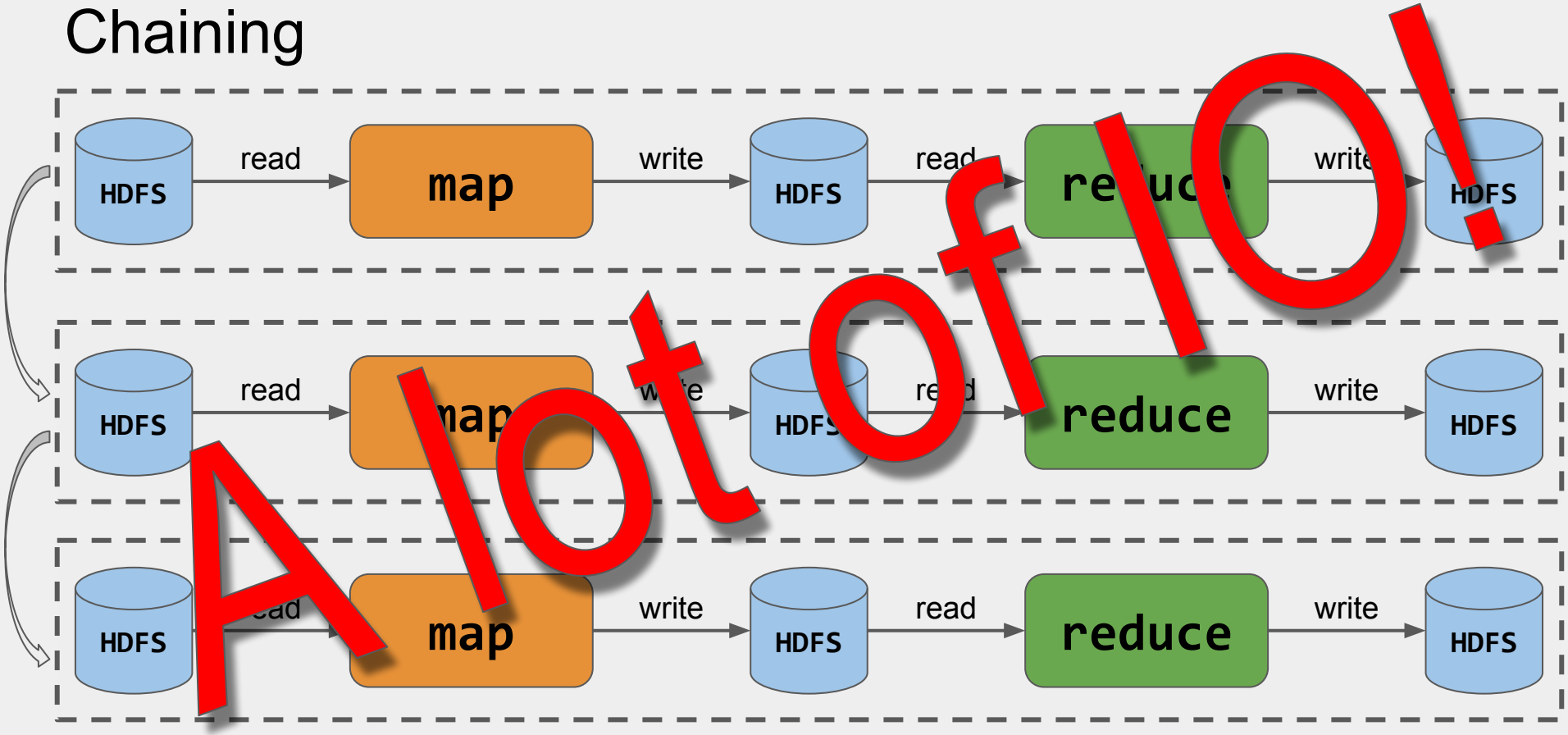

hadoop
Map Reduce
HDFS
read
map
write
HDFS
read
reduce
write
HDFS
shuffle
sort
reduce
More IO!
Reduce logic is executed only
during the last ⅓ of reducer stage

*What if you need to **map** and/or **reduce** multiple times?*

Chaining


HDFS
read
map
write
HDFS
read
reduce
write
HDFS
HDFS
read
map
write
HDFS
read
reduce
write
HDFS
HDFS
read
map
write
HDFS
read
reduce
write
HDFS

Chaining


HDFS
read
map
write
HDFS
read
reduce
write
HDFS
HDFS
read
map
write
HDFS
read
reduce
write
HDFS
HDFS
read
map
write
HDFS
read
reduce
write
HDFS
A lot of IO!

Thus was born

*What is **Apache Spark**?*

*What is **Apache Spark**?*

A general-purpose, distributed,
in-memory computing system

A little preview…

- Project Captain Planet — **`Dataset`**
- Project Cougar
- ETL Project

} **`DataFrame`**

A little preview…

Sign up for Databricks Community Edition

Signup: https://databricks.com/try-databricks

Login: https://community.cloud.databricks.com

Examples

- Scala Examples: http://tinyurl.com/sdsc-2019-atl-scala
- `RDD` Examples: http://tinyurl.com/sdsc-2019-atl-rdd
- `DataFrame` Examples: http://tinyurl.com/sdsc-2019-atl-dataframe
- `Dataset` Examples: http://tinyurl.com/sdsc-2019-atl-dataset
- CSV Read Examples: http://tinyurl.com/sdsc-2019-atl-csv-read
- Join Examples: http://tinyurl.com/sdsc-2019-atl-join
- ML Examples: http://tinyurl.com/sdsc-2019-atl-ml
- Spark+TensorFlow Examples*: http://tinyurl.com/sdsc-2019-atl-spark-tensorflow

**The examples fail to run as Databricks notebook does not have enough resources to run them.*

Most examples can be found in my 

https://github.com/chrismin1202/spark-bootcamp/tree/sdsc-2019-atlanta

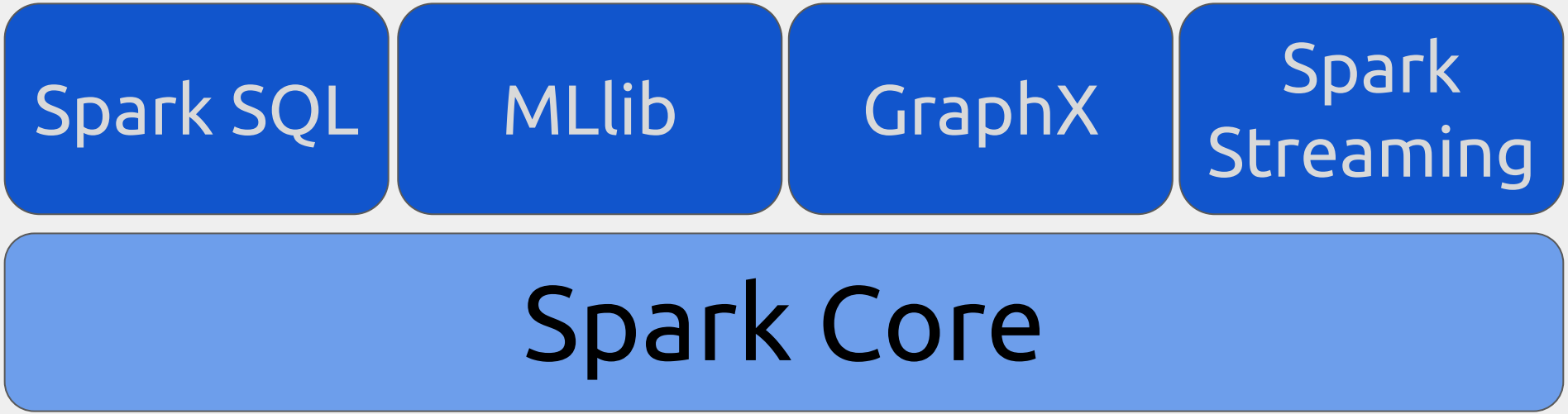
The repository also contains a few more examples that cannot be run in Databricks notebook.

Agenda

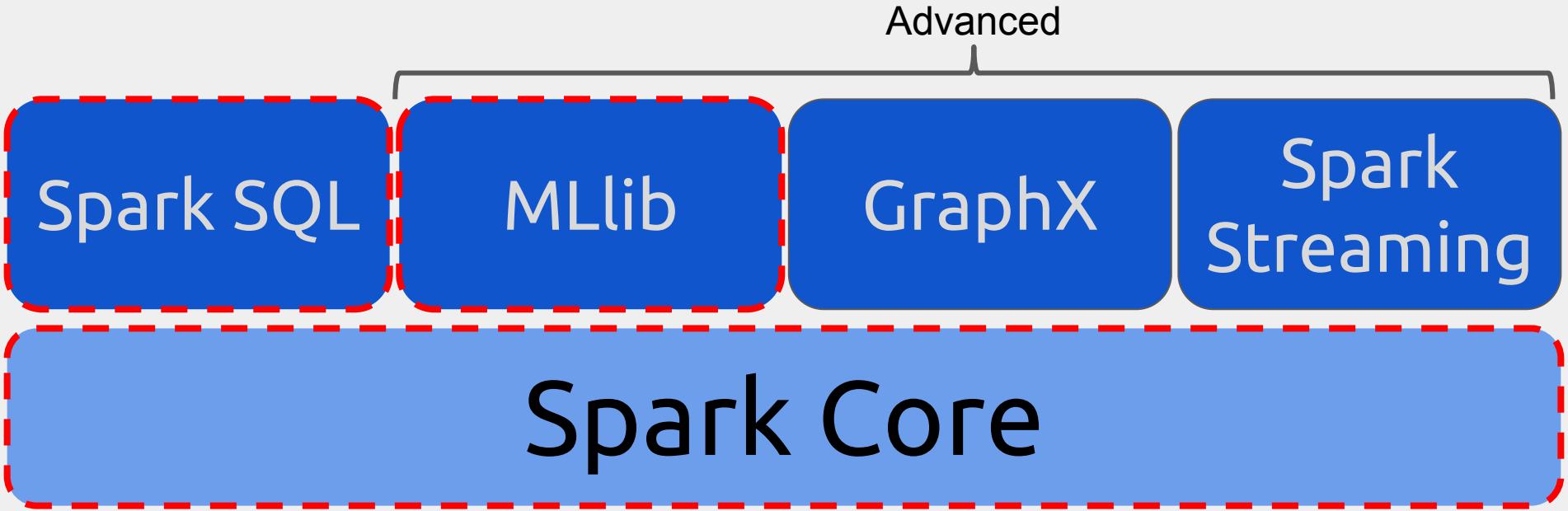
- Spark architecture
- Spark fundamentals
- Standalone Spark vs. Spark on a Resource Manager
- Programming language choices for Spark
- Spark SQL
- Shared variables: Accumulator, Broadcast variables
- Spark ML Pipelines
- `spark.mllib` vs. `spark.ml`
- Spark on Kubernetes
- Tips & Configuration options for performance tuning

Let’s take a deeper dive into Spark’s architecture first.

Spark Architecture

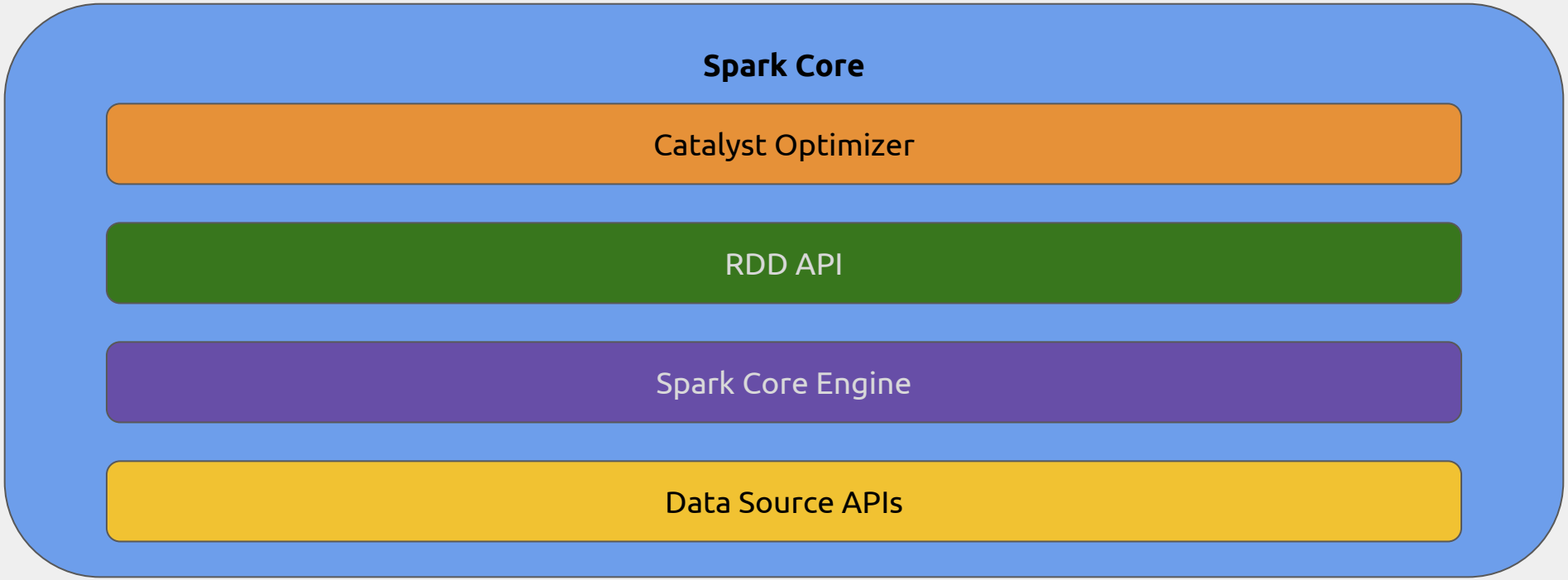


Spark Architecture

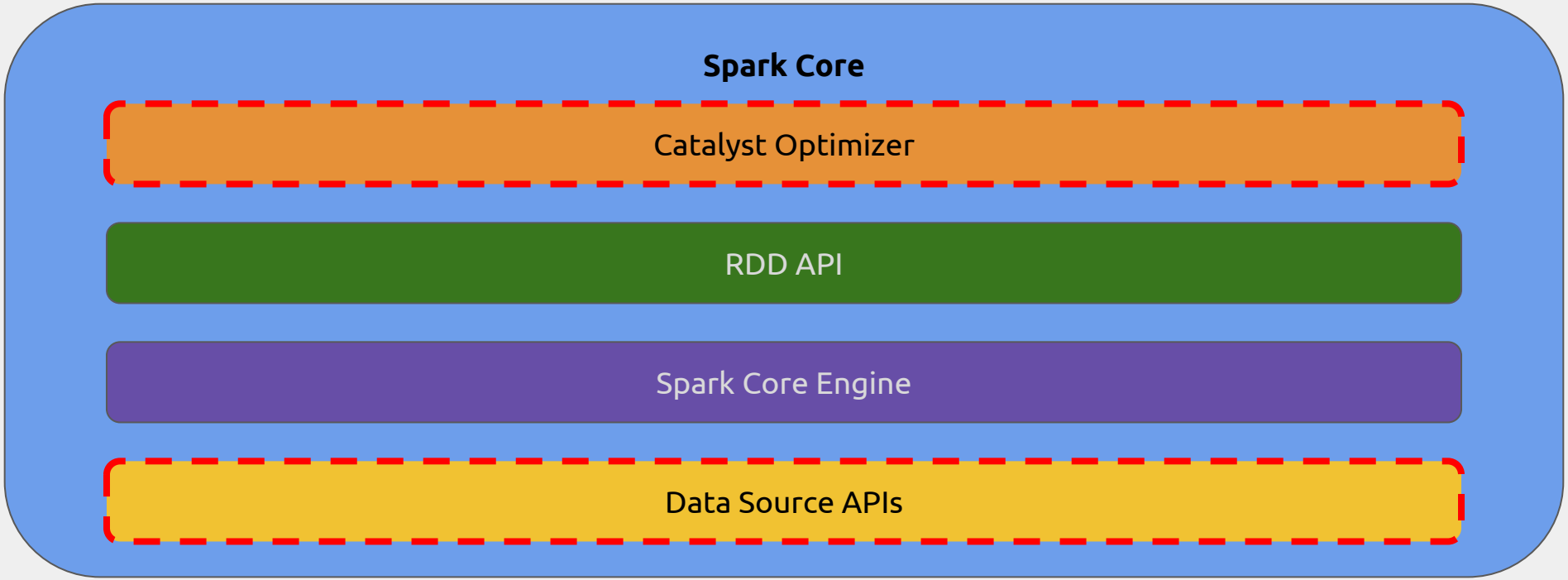


Spark Architecture

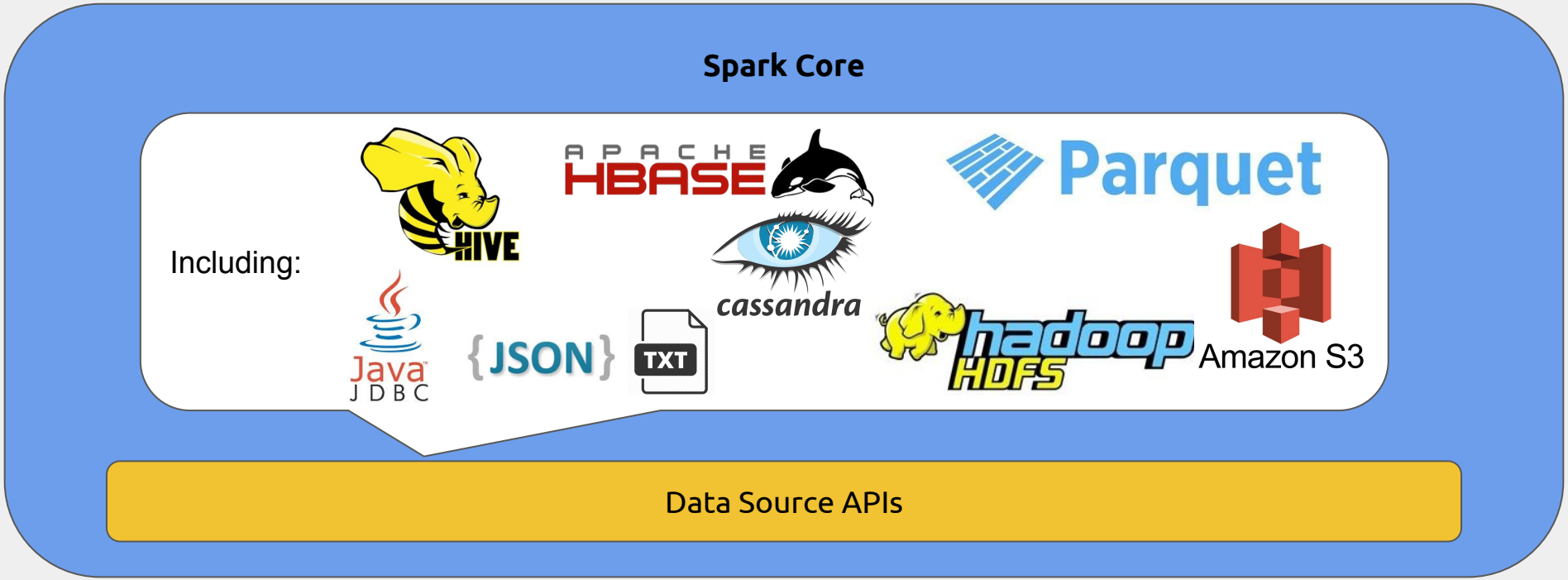
Spark Core Architecture



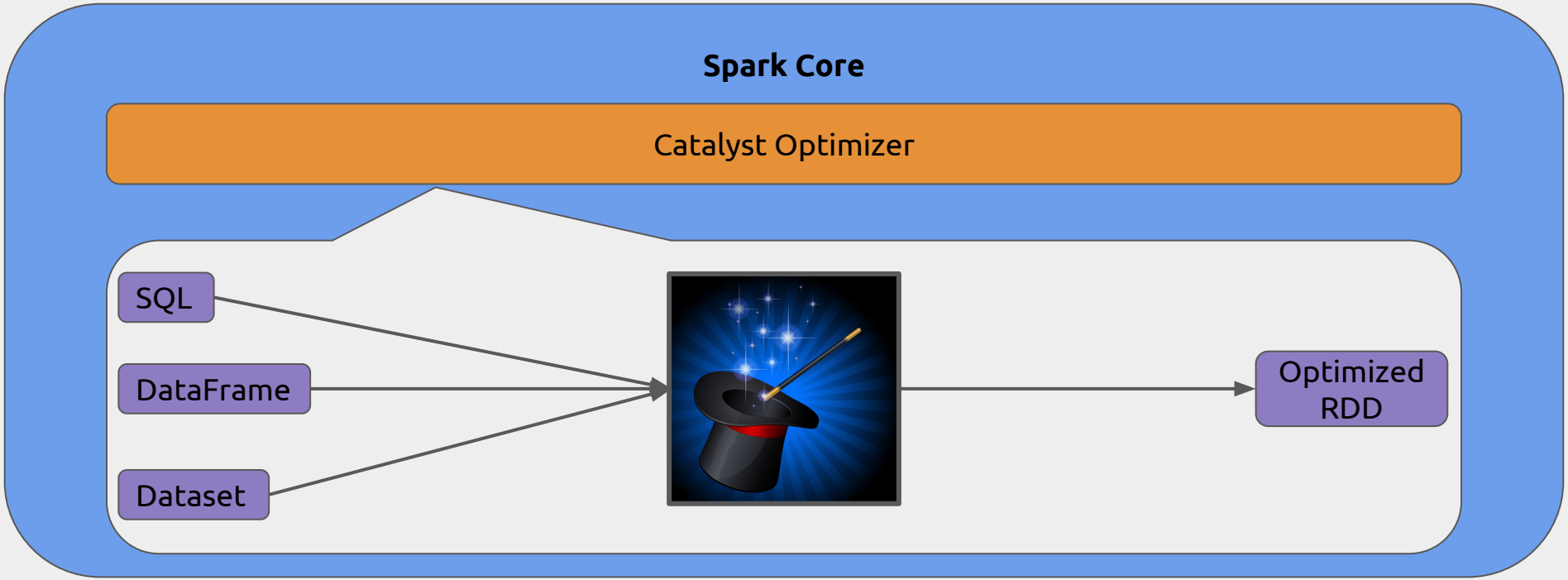
Spark Core Architecture



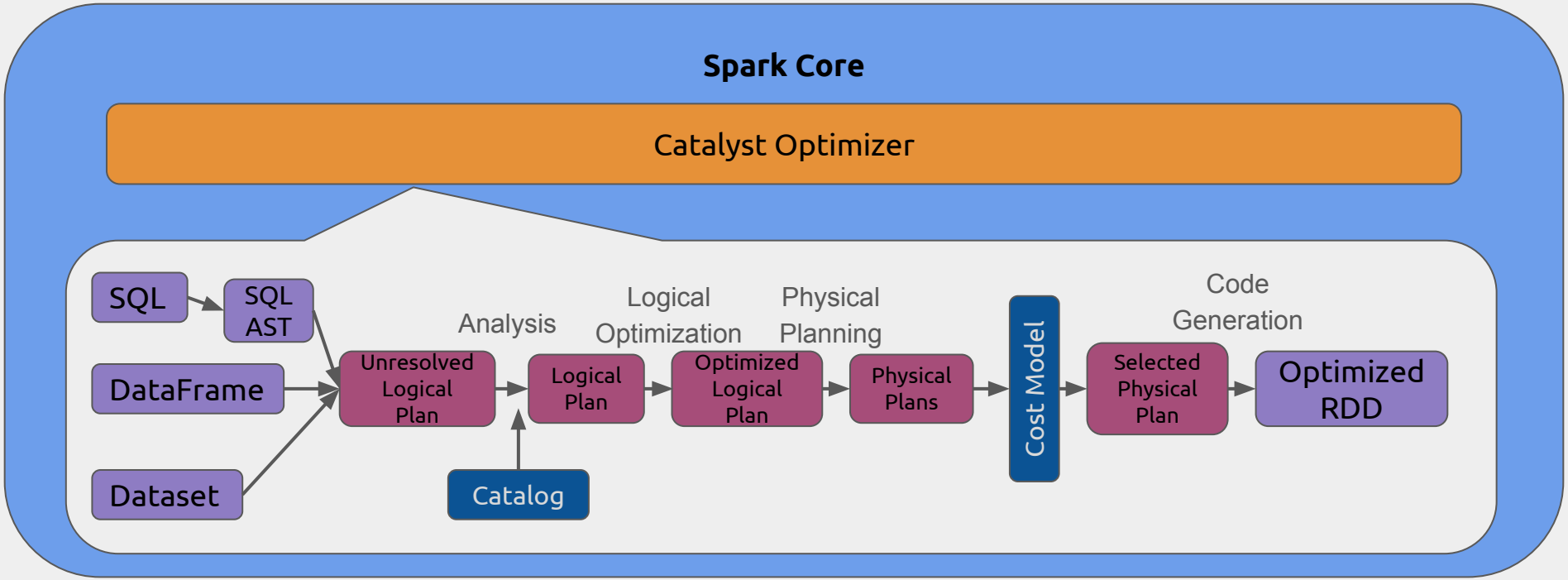
Spark Core: Data Source APIs



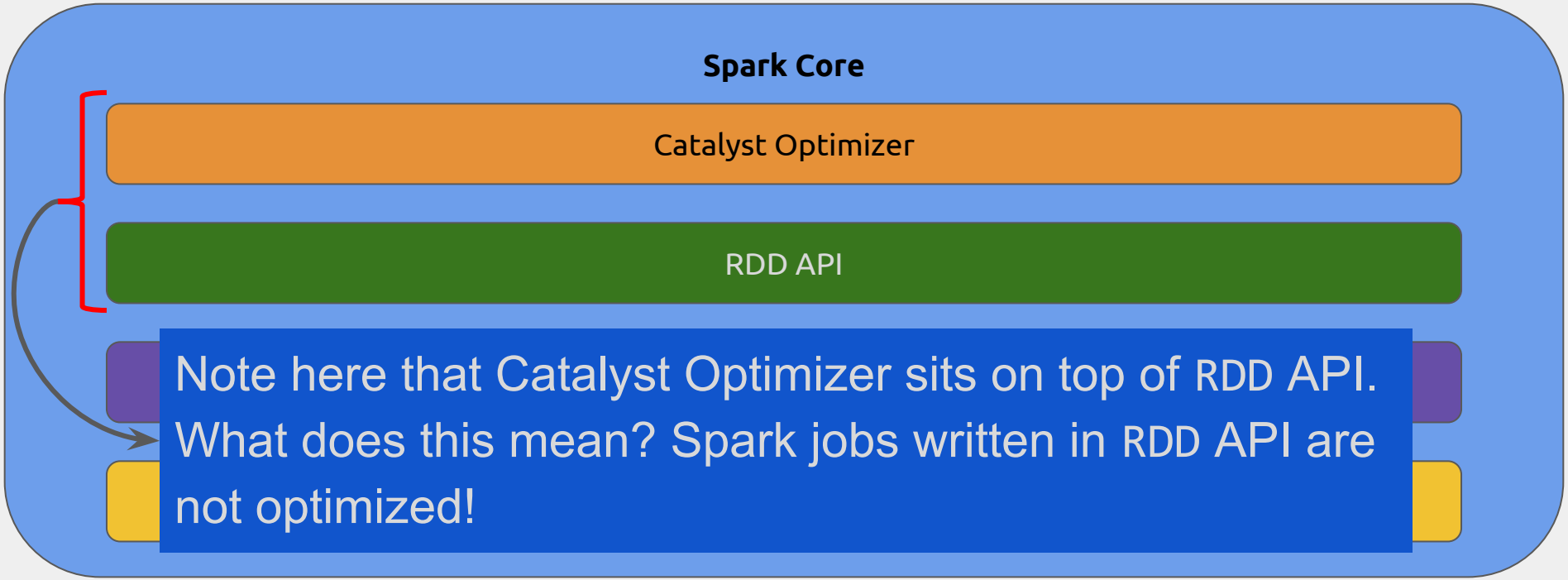
Spark Core: Catalyst Optimizer



Spark Core: Catalyst Optimizer

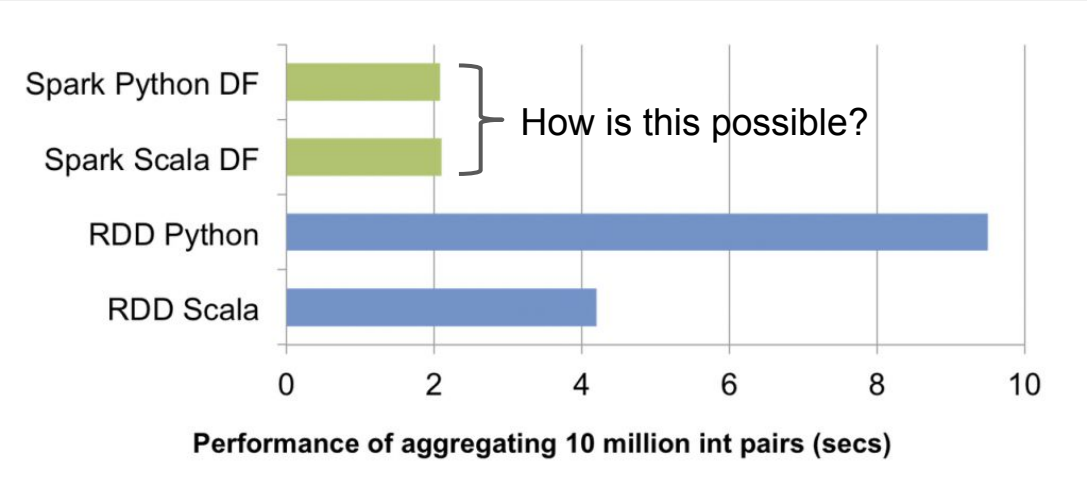


Spark Core Architecture



There is yet another reason especially if you are writing in Python!

If you are writing Spark application in Python, you should definitely stick to `DataFrame` because...



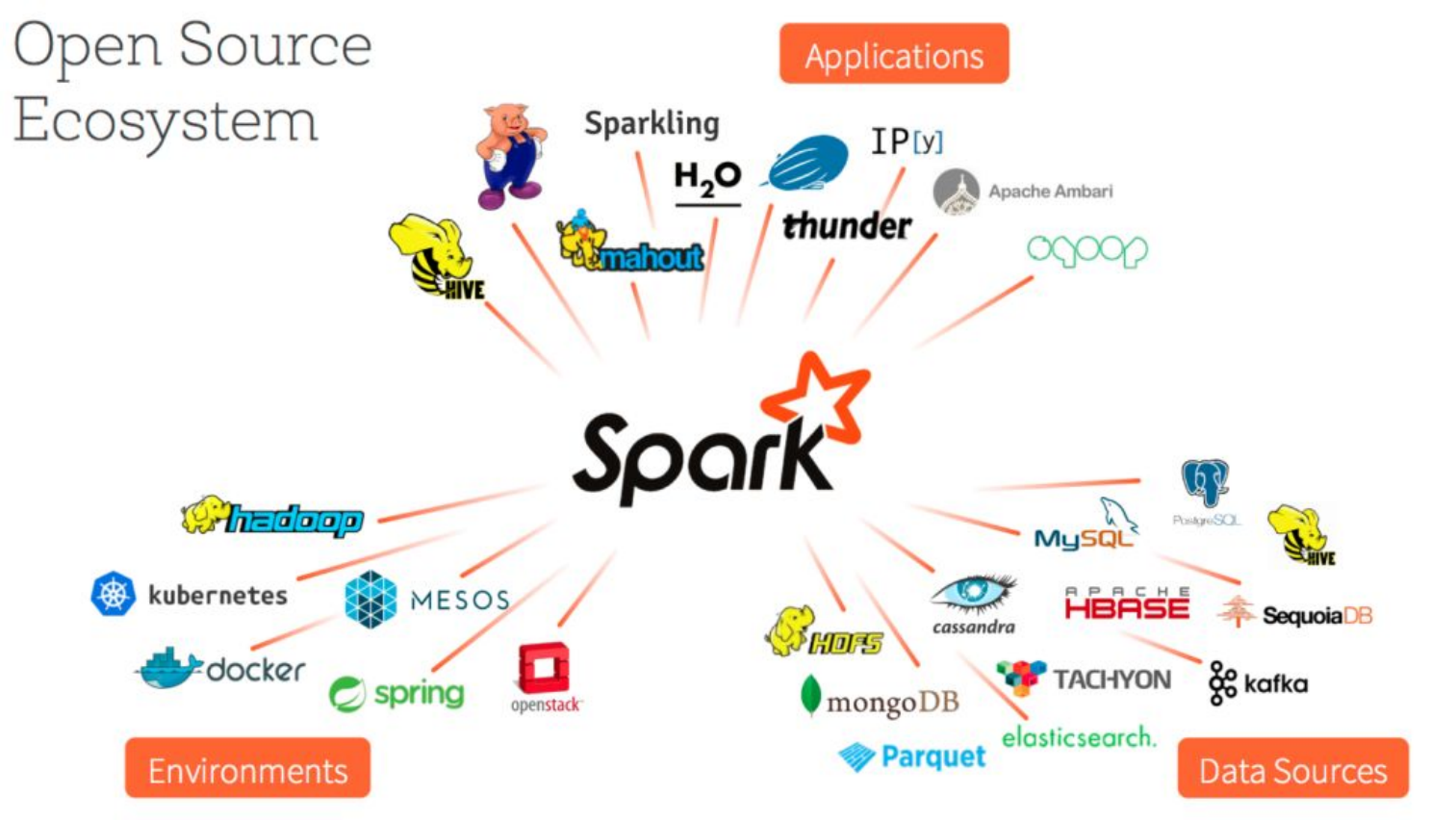
https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html

Catalyst Optimizer!

Catalyst Optimizer!

Both Scala `DataFrame` code and PySpark `DataFrame` code get compiled to more or less the same highly optimized `RDD` code.

Spark Ecosystem

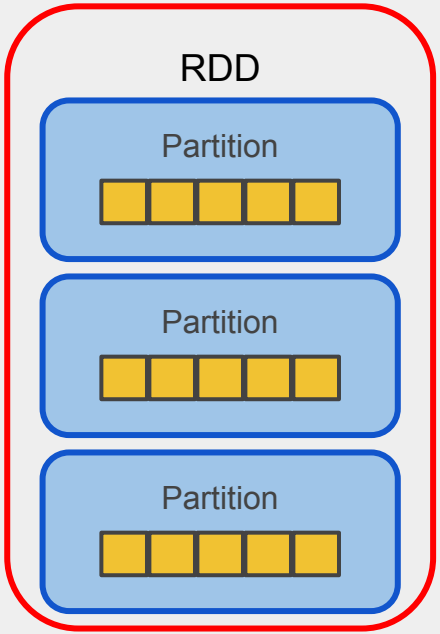


https://databricks.com/blog/2016/01/05/apache-spark-2015-year-in-review.html

Key concepts of **Spark**

RDD: The Building blocks of Spark

- Stands for **R**esilient **D**istributed **D**ataset



Can be distributed to multiple nodes

Fault-tolerant

Can be partially recomputed when there is a node failure

Transformations vs. Actions

- **Transformation**s are operations that “transform” RDD[T] to RDD[U] *lazily*.
- Two categories of transformation:
    a. Narrow transformation
    b. Wide transformation

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

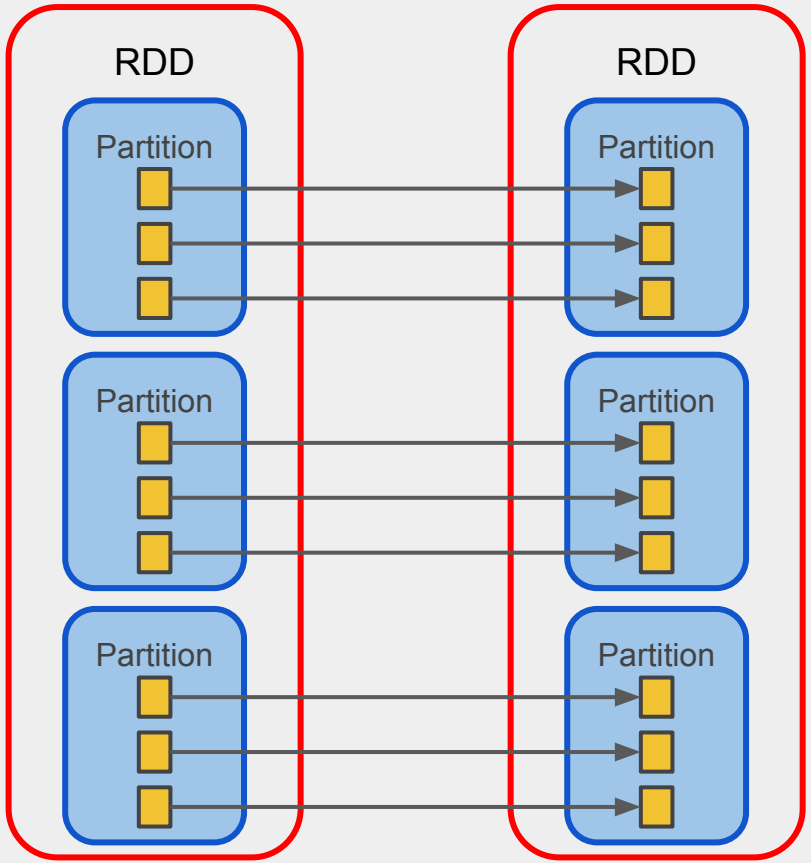
- **Action**s are operations that trigger computation
    a. The computed values are either sent to the driver or saved to a storage.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Narrow Transformations

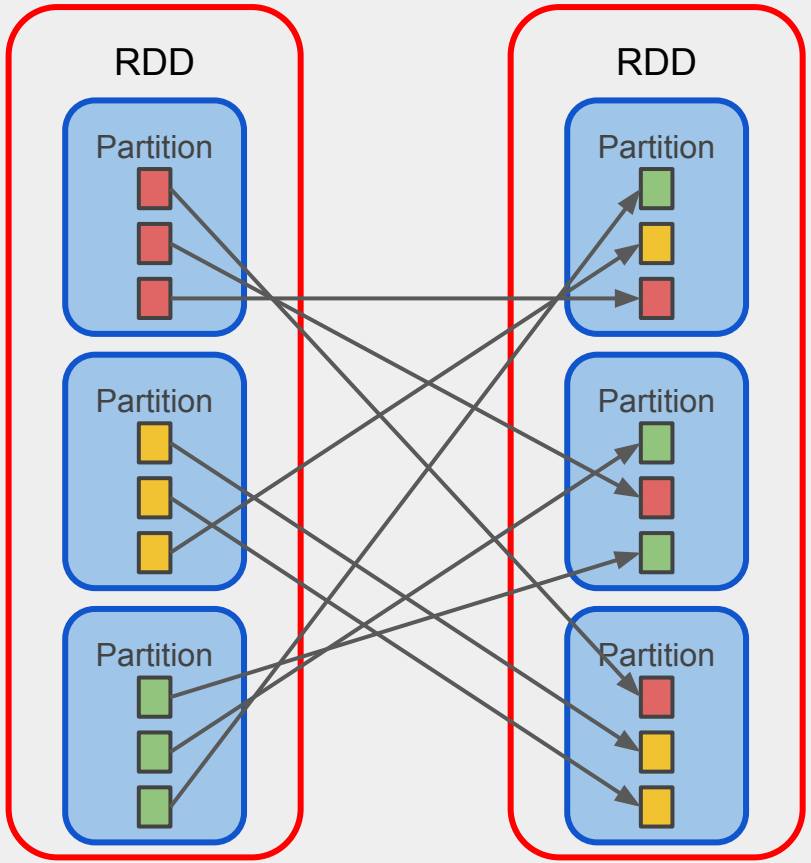
Operations that do not require **shuffle**

- `map`
- `flatMap`
- `filter`
- etc.



Wide Transformations

- Operations that require **shuffle**
- “Widening” causes records in a single partition to spread across multiple partitions
- Caused by **`combineByKey`** and the transformations derived from it



Actions

- `reduce(func)`
- `collect()`
- `count()`
- `first()`
- `take(n)`
- `saveAsTextFile(path)`
- `foreach(func)`
- etc.

Persistence

- RDDs can be persisted and reused for multiple actions.
- Two ways to persist:
  - `rdd.persist(StorageLevel)`
    - Persists at the specified storage level
  - `rdd.cache()` same as `rdd.persist()`
    - Persists at the default storage level (`MEMORY_ONLY`)

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

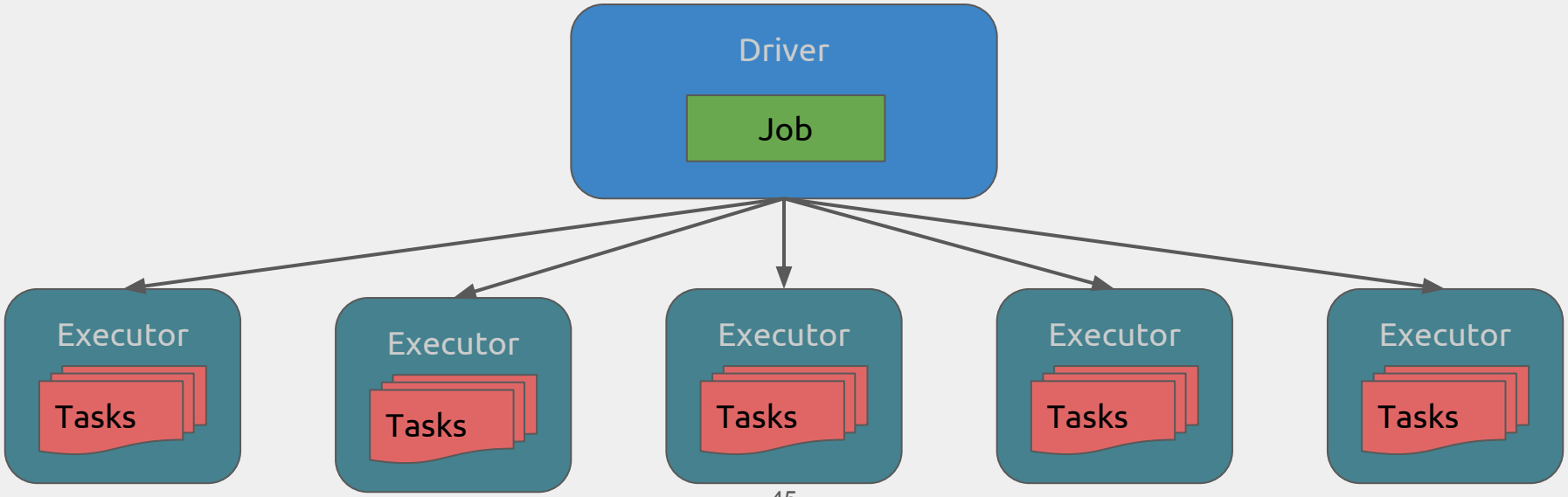
Persistence

Storage Level	
`MEMORY_ONLY`	Persists deserialized objects in memory; some partitions are not cached when not fit in memory
`MEMORY_AND_DISK`	Persists deserialized objects and spills to disk when not fit in memory
`MEMORY_ONLY_SER` (Java and Scala)	Same as MEMORY_ONLY, but persists serialized objects; more memory-efficient, but read is CPU-intensive due to deserialization
`MEMORY_AND_DISK_SER` (Java and Scala)	Same as MEMORY_AND_DISK, but persists serialized objects
`DISK_ONLY`	Persists only to disk
`*_2` for the levels above, e.g., `MEMORY_ONLY_2`	Same as above, but replicated to 2 nodes
`OFF_HEAP` (experimental)	Off-heap memory needs to be enabled; similar to `MEMORY_ONLY_SER`, but the RDD is persisted in off-heap memory

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

Driver vs. Executor

- Driver
  - An instance that schedules tasks and coordinates the executors
- Executor
  - An instance that executes the task scheduled by the driver



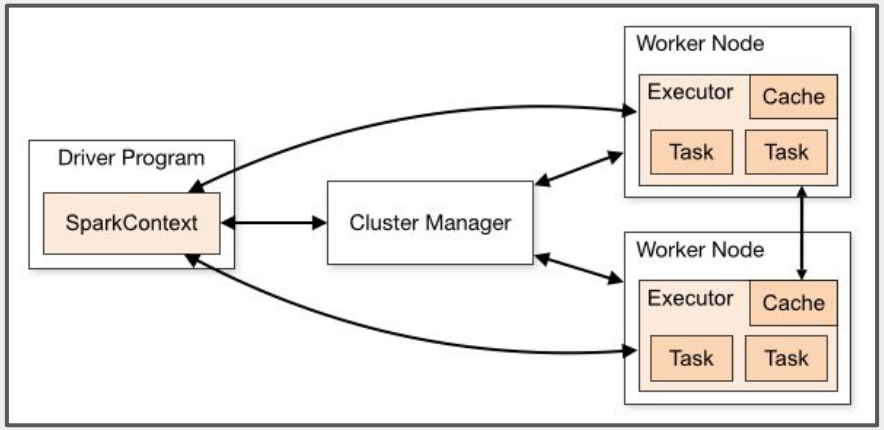
Client Mode vs. Cluster Mode

- Client Mode
  - The driver runs on the machine from which the job is submitted
- Cluster Mode
  - The driver runs from a random node in the cluster
  - Cluster Managers
    - Standalone: You do all the work
    - Apache Hadoop YARN: YARN stands for **Y**et **A**nother **R**esource **N**egotiator
    - Apache Mesos
    - Kubernetes 🔥🔥🔥

https://spark.apache.org/docs/latest/cluster-overview.html

Deeper Dive into Cluster Manager (YARN)

Enterprise-level applications typically run on a cloud or from a data center and somebody needs to manage resources. This is where cluster managers like YARN comes into play.



https://spark.apache.org/docs/latest/cluster-overview.html

Deeper Dive into Cluster Manager (YARN)

- YARN is capable of allocating resources to any application from a centralized pool of resources, e.g., memory, CPU
  - Not just Spark but MapReduce, Pig, Impala, etc.
- YARN can control the number of nodes for each Spark application
  - Note that in Standalone mode, an executor needs to run on every node
- YARN supports security
  - If the Hadoop cluster is “Kerberized,” i.e., if Kerberos, which is the authentication protocol used in Hadoop, is enabled, it applies secure authentication between processes.
- YARN has a scheduler
  - Allows you to prioritize and preempt tasks

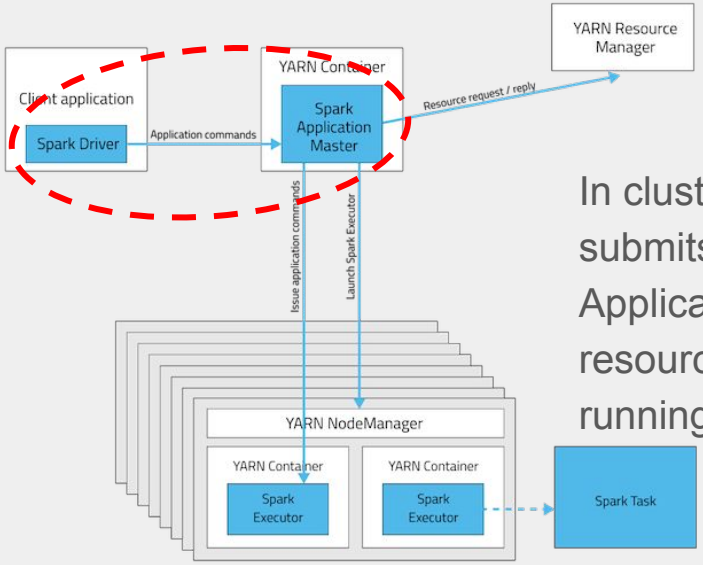
https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Deeper Dive into Cluster Manager (YARN)

- YARN supports 2 modes
  - **client mode**
    - The driver runs in the machine from which the job is submitted
    - Run in this mode if
      - interactively tuning or debugging the application
      - the application needs to support command line arguments and it needs to print the output to the console, e.g., `--help` option typically prints usage to the console.
    - Caution: When multiple client-mode jobs are submitted simultaneously, the machine can go down!
  - **cluster mode**
    - The driver runs on a random node just like executors

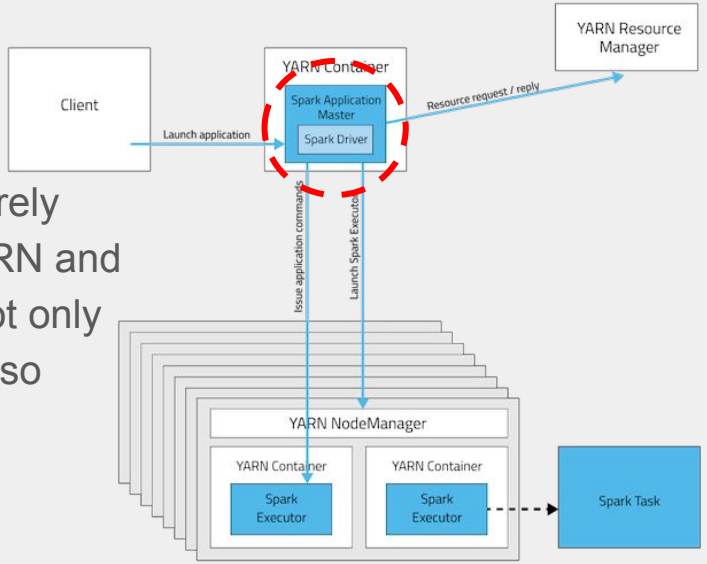
https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Deeper Dive into Cluster Manager (YARN)



client-mode

In cluster-mode, the client merely submits the application to YARN and Application Master handles not only resource managements but also running the driver.

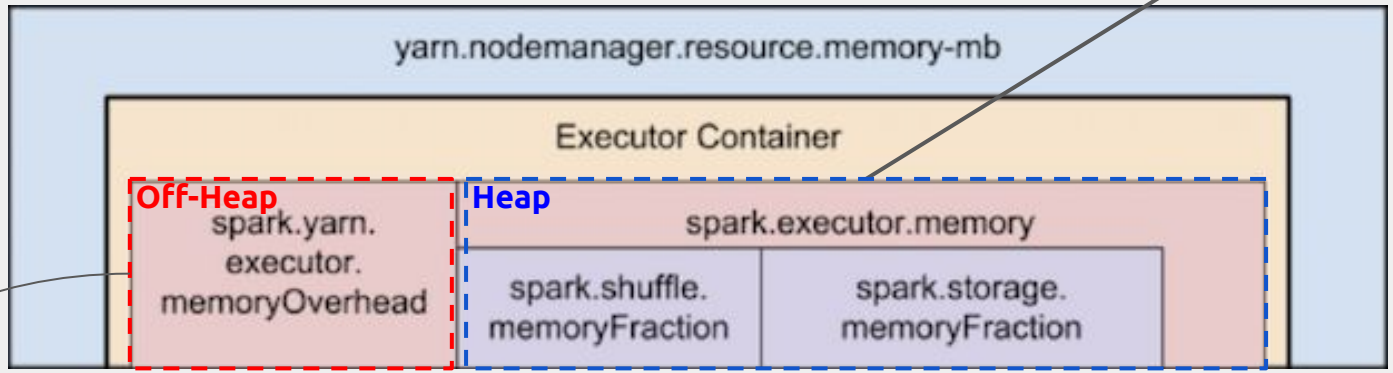


cluster-mode

https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

Memory Allocation in YARN

Can be specified in either MB or GB, e.g., 2048M, 3G
`OutOfMemoryError` is a typical symptom.



default: `max(384, .07 * spark.executor.memory)` Often not enough
The executor that ran out of off-heap memory are killed by YARN
Typical error message: “Consider boosting spark.yarn.executor.memoryOverhead”
Solution: resubmit the application with `--conf spark.yarn.executor.memoryOverhead [memory in MB]`
ex) `--conf spark.yarn.executor.memoryOverhead 1024`

https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

More on Memory Management

Project Tungsten

- Closer to the metal
- Explicit memory management off-heap to avoid GC pause
  - Spark applications started requiring more off-heap memory (`spark.yarn.executor.memoryOverhead`) as a result.
- Efficient code generation for Spark SQL
  - This is yet another reason why one should not use `RDD` API directly anymore!

https://databricks.com/glossary/tungsten
https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

Programming Languages for Spark

Programming Languages for Spark

Chris' Pick

Why Scala?

- Spark is implemented in Scala for the most part.
- Spark takes advantage of Scala's language features and constructs.
- vs. Java
  - Scala's syntactic elegant allows developers to write less code
  - Scala gives you expressive power: functional, object-oriented, pattern-matching, etc.
- vs. Python
  - Scala's syntax is as simple as that of Python
  - Scala is strongly-typed whereas Python isn't
  - Python lacks `Dataset` API
- vs. R
  - SparkR has many limitations

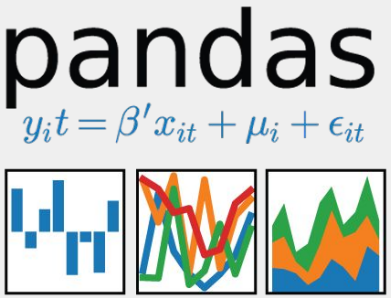
Programming Languages for Spark

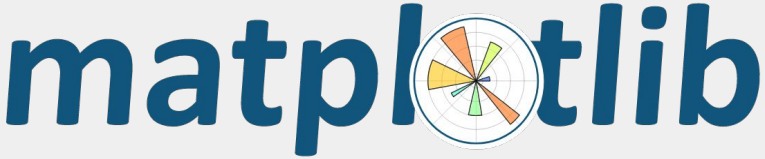
Data Science

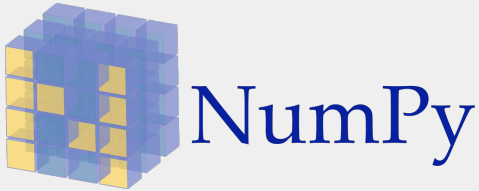
Why Python?

- The language is easier to learn.

- Many widely used data science libraries are written in Python.





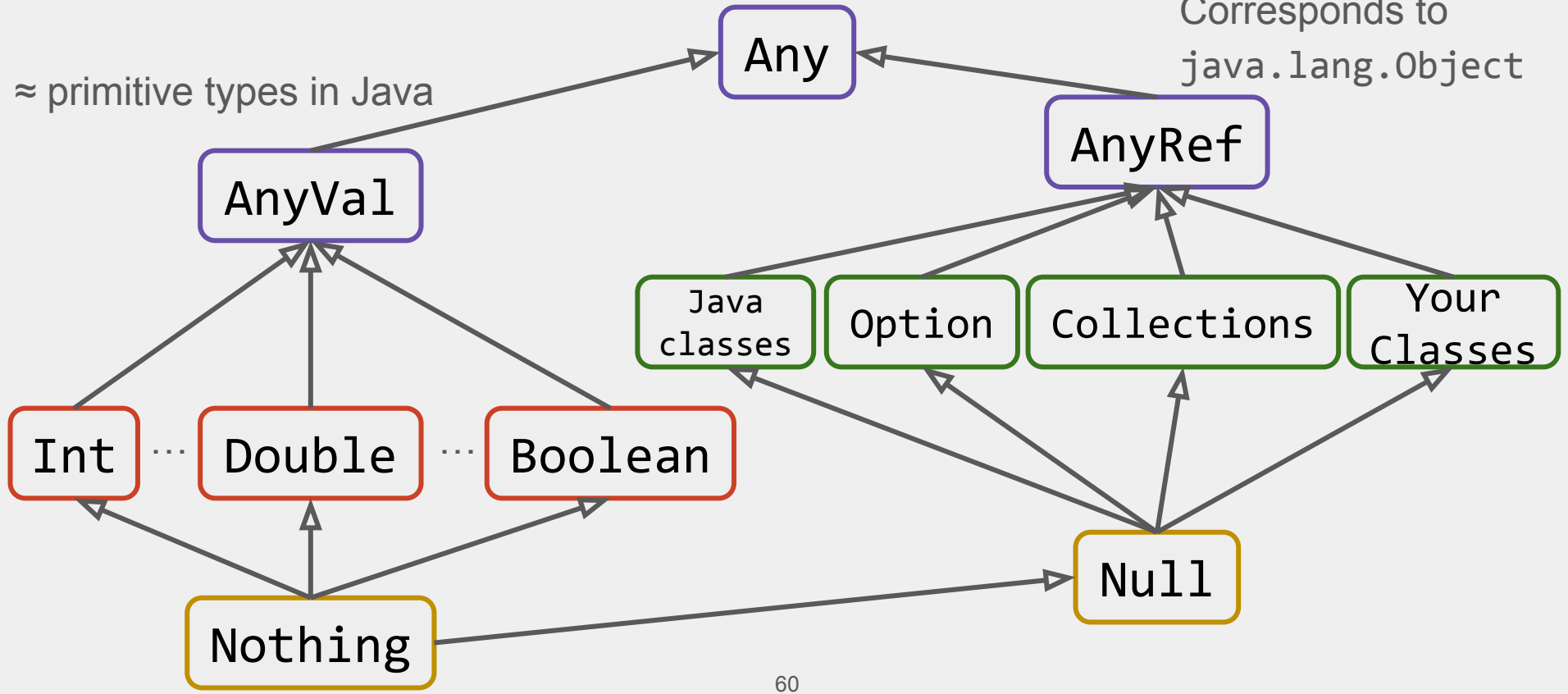


I choose Scala (mainly) for this course

- Scala is the most native language for writing Spark application
- Scala gives you the “full picture”
  - As aforementioned, Python lacks `Dataset` API. Even if you end up not using it, you should at least understand what it is and its pros/cons.
- Java sucks for writing Spark application
  - It is strictly my opinion. Java has many other good use cases, but Spark ain’t the one. Challenge me on this.

Scala Crash Course

Type Hierarchy



Visibility

- No **`public`** modifier
  - The default visibility, i.e., when there is no visibility modifier, is public. In Java, when visibility modifier is omitted, the class/method is implicitly package-private.
- **`protected`** visibility is more or less the same as that of Java
- **`private`** visibility is also similar to that of Java except:
  - When applied to the top level classes, it makes the class visible to other source and test classes in the package.
  - One can restrict the visibility of the class even narrower with **`private[this]`**.
  - Similarly, **`private[package]`** limits the visibility to the specified package. Note that the package is required to be one of the parent packages of the class.

Syntax: Method Signature

```
def methodName(param1: Type1, param2: Type1): ReturnType

// variable arguments

def methodName(varargs: Type*): ReturnType

// curried

def methodName(param1: Type1)(param2: Type2): ReturnType
```

Syntax: Generics (vs. Java)

Java	Scala

`class Clazz<T, U> {`
`  U methodName(T t) { … }`
`}`
	`class Clazz[T, U] {`
	`  def methodName(t: T): U = { … }`
	`}`

`class Clazz<T super Foo> {`
`  void methodName(T t) { … }`
`}`
	`// lower bound`
	`class Clazz[T >: Foo] {`
	`  def methodName(t: T): Unit { … }`
	`}`

`class Clazz<T extends Foo> {`
`  void methodName(T t) { … }`
`}`
	`// upper bound`
	`class Clazz[T <: Foo] {`
	`  def methodName(t: T): Unit { … }`
	`}`

Syntax: Generics with Wildcard (vs. Java)

Java	Scala
`void methodName(Clazz<? super Foo> t) { … }`	`// lower bound` `def methodName(t: _ >: Foo): Unit { … }`
`void methodName(Clazz<? extends Foo> t) { … }`	`// upper bound` `def methodName(t: _ <: Foo): Unit { … }`

Scala Classes

- `class`
  - same as Java `classes`
- `object`
  - singleton, i.e., cannot be instantiated
- `trait`
  - ≈ Java `interfaces`
- Scala has a few special classes
  - 

Companion object

- **`class`**es and **`trait`**s can be accompanied by what is known as companion object
- How to create companion objects?
  - Create an **`object`** with the same name as the **`class`**/**`trait`** it accompanies in the same file.

```
class Foo { ... }

object Foo { ... } // companion of class Foo
```

Why does Scala have companion object?

Scala does not have **`static`** modifier!

How does Scala handle static methods?

- Methods created in an **`object`** are effectively static as **`object`**s are singletons.

`sealed` Keyword

- Applies to non-`final` and `abstract` `class`es and `trait`s
- When a `class`/`trait` is sealed, it can be extended only in the same file

```
// in the same file

sealed trait Foo

class FooImpl extends Foo
```

Operator “Overloading”

- It is similar to operator overloading in C++, but in Scala, you are basically naming a method with operators (symbols).

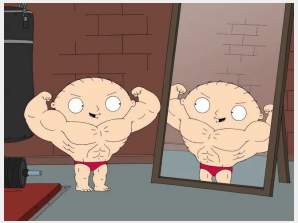
```
val numbers = Vector(2, 3, 4) // Vector is a subclass of Seq

1 +: numbers // +: prepends an element to the collection

numbers :+ 5 // :+ appends an element to the collection
```

Pattern Matching

- Switch statement on steroid!!
- Can match on types or values



```
num match {
  case 0 => // when 0
  case 1 => // when 1
  case neg if neg < 0 => // when negative
  case pos => // when positive and greater than 1
}
```

Matched from top to bottom

`apply()` Method

- A special method name that comes with a syntactic sugar
- A method named **`apply`** can be invoked with a set of parentheses.

```
// Array is an example
final class Array[T](_length: Int) {
  def apply(i: Int): T = ...
}
```

  - Instance `apply` method
    - Invoked with a set of parentheses on an instance
  - Static `apply` method
    - Refers to `apply` methods defined in an `object`
    - Invoked with a set of parentheses on the object name itself

- Typically used for creating factory methods for instantiating classes or accessing an element in collection types

Partial Functions

- Technically of type **`scala.PartialFunction`**
- As opposed to regular functions, a partial function is designed to apply the function logic it holds to a subset of inputs

```
val evenOrOdd: PartialFunction[Int, String] = {

  case x if x > 0 && x % 2 == 0 => s“$x is even”

  case x if x > 0 && x % 2 != 0 => s“$x is odd”

  // scala.MatchError is thrown for 0 and negative integers

}
```

https://www.scala-lang.org/api/current/scala/PartialFunction.html

Scala Classes

- `case class`
  - a special type of class that extends `scala.Product` trait
  - comes with **`equals`**/**`hashCode`**, **`toString`**, **`copy`**, and factory (**`apply(...)`**) methods
  - can be pattern-matched
  - limited to 22 parameters in <= Scala 2.10
    - This restriction was lifted in Scala 2.11

Let's look at `case class` closer

```
case class Profile(name: String, age: Int)
```

```
class Profile(val name: String, val age: Int) extends Product with Serializable {

  override def equals(that: AnyRef): Boolean = { … }

  override def hashCode(): Int = { … }

  override def toString(): String = { … }

  override def productElement(n: Int): Any

  override def productArity: Int

  override def productIterator: Iterator[Any]

  def copy(name: String = this.name, age: Int = this.age): Int = { … }

}
```

```
// companion object

object Profile {

  def apply(name: String, age: Int): Profile = { … }

}
```

Now you know what **`case class`** is.
Can anyone guess what **`case object`** would be?

Scala Classes

- **`case object`**
  - In addition to the characteristics of **`case class`**, it has the characteristics of **`object`** as well, i.e., a singleton.

Side Notes:

`sealed trait` + **`case object`**s is a pattern that is often employed in Scala. It is a powerful way to define enum-like values. Scala of course has **`Enumeration`** that is equivalent to **`java.lang.Enum`**.

Scala Classes

- **`Tuples`**
  - are essentially the same as `case classes` except that each element is assigned a number (index) instead of name
  - limited to 22 parameters
  - special syntax:

```
("a", 2, 3.0) // same as Tuple3("a", 2, 3.0)
```

`scala.{Option, Some, None}`

- Similar to **`java.util.Optional`** introduced in Java 8, **`scala.Option`** is an object container that allows developers to not only avoid `null`s but also take functional approach
- **`scala.Some(...)`** ≈ **`java.util.Optional.of(...)`**
- **`scala.None`** ≈ **`java.util.Optional.empty()`**
- Functional methods like **`.map(...)`**, **`.flatMap(...)`**, **`.foreach(...)`** can be invoked.
- Note that **`scala.Option`** is a **`sealed abstract class`**.

https://www.scala-lang.org/api/current/#scala.Option

scala.{Option, Some, None}

```
// simplified for illustration purpose

sealed abstract class Option[+A] { … }

final case class Some[+A](x: A) extends Option[A] { … }

case object None extends Option[Nothing] { … }
```

Side Notes:

- **None** does not need **final** modifier as **objects** are implicitly **final**.
- '+' applied to a generic type means "covariant", whereas '-' means "contravariant"

`implicits`

- A keyword that can be applied to **`class`**, **`def`**, **`var`**, **`val`**, and method parameters
- **`implicit class`**
  - Should have only 1 non-implicit parameter; otherwise, it is ignored in implicit lookup
  - Must be defined inside of another class
  - There should not be a name collision within its scope.
    - This implies that **`implicit class`**es cannot be **`case class`**es as companion objects auto-generated for them.
- **`implicit def/var/val`**
  - Letting the compiler to invoke the function on your behalf if the implicit function with the same type is available in the scope
  - There cannot be two or more implicit functions with the same type as the compiler cannot disambiguate.

`implicits`

- Often used in so-called *pimp-my-library* pattern
  - Example: `scala.collection.JavaConverters`

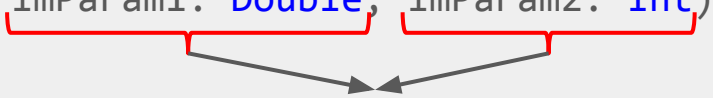
implicits

- `implicit` method parameters
  - If there is an implicitly defined value with the matching type in the caller's scope, the caller does not need to explicitly pass in the parameter, hence implicit.
  - If there are two or more implicitly defined value of the same type, the compiler cannot disambiguate.
  - To mix with explicit parameters, parameters need to be curried.

```
def method(exParam: String, implicit imParam: Double): Int // No!

def method(exParam: String)(implicit imParam: Double): Int // Yes!

def method(exParam: String)(implicit imParam1: Double, imParam2: Int): Int // Yes!
```

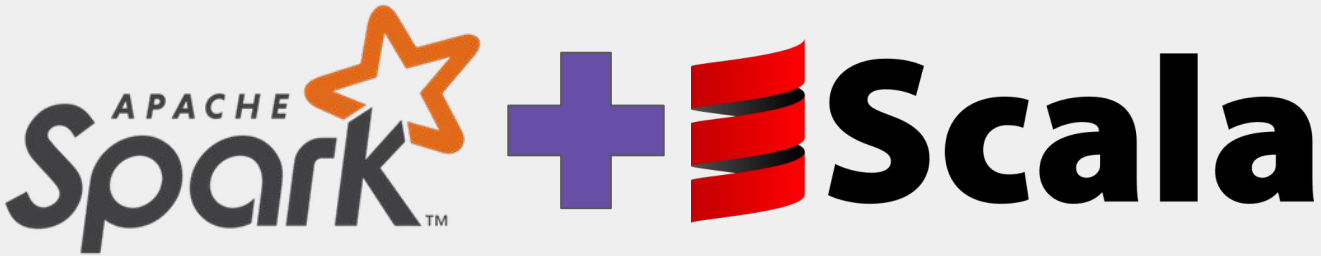


Both parameters are implicit.

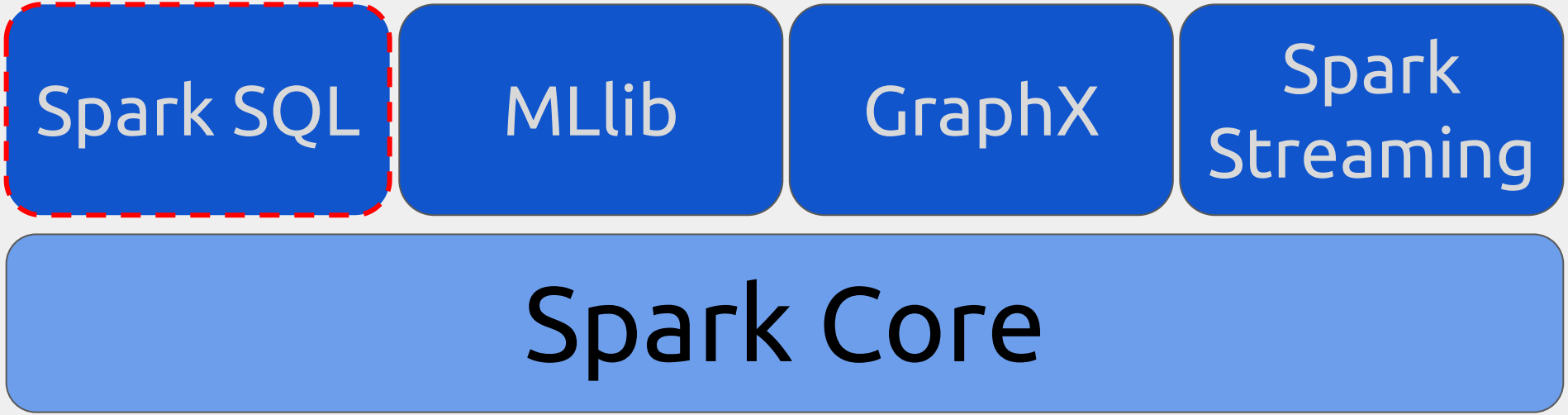
Let’s code!

http://tinyurl.com/sdsc-2019-atl-scala

We are now ready to write Spark applications in Scala



Spark SQL

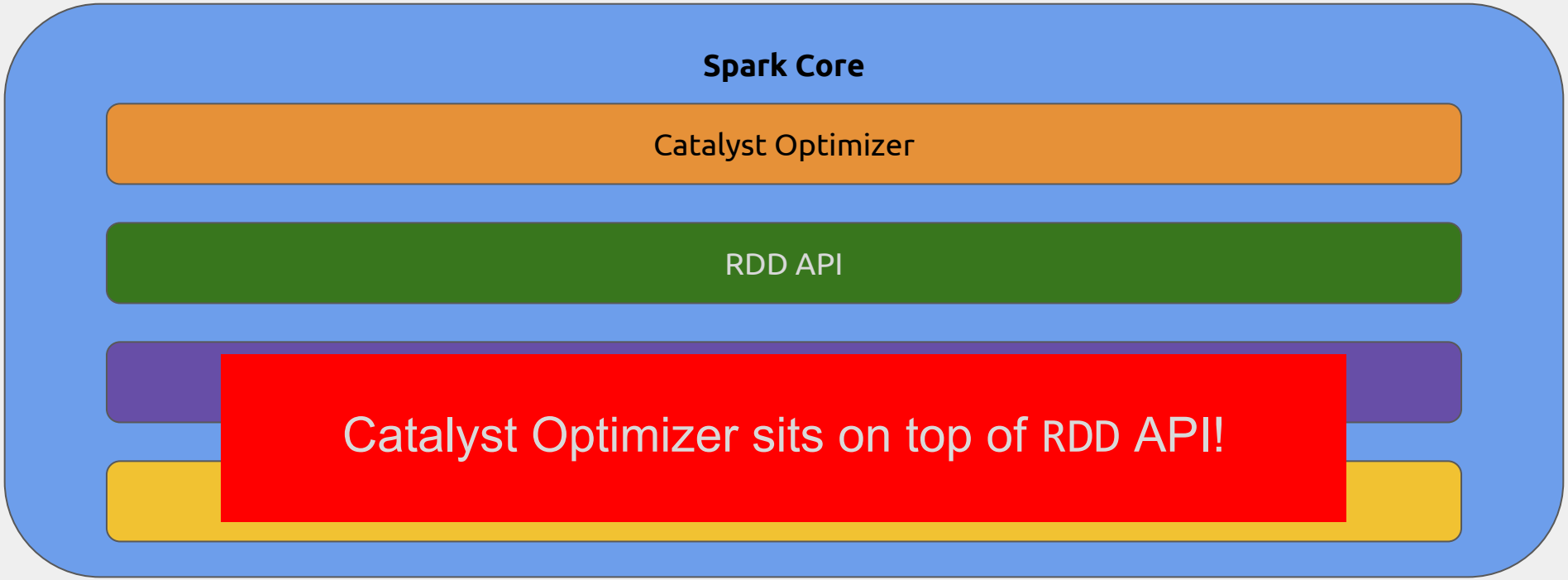


Spark SQL

- An abstraction for structured data processing
- Ways to write:
  - By crafting queries with an actual SQL syntax, e.g., Hive
  - `Dataset` API
  - `DataFrame` API
- Why use Spark SQL over `RDD`?
  - Remember the architecture of Spark Core?

https://spark.apache.org/docs/latest/sql-programming-guide.html

Refresher...


Spark Core
Catalyst Optimizer
RDD API
Catalyst Optimizer sits on top of RDD API!

Transformations expressed in Spark SQL are optimized through Catalyst Optimizer.

Spark SQL: Spark 1.x vs. Spark 2.x

1.x	2.x
N/A	`org.apache.spark.sql.`**`SparkSession`**
`org.apache.spark.`**`SparkContext`**	`org.apache.spark.sql.`**`SparkSession`** has **`sparkContext: SparkContext`** as a member
`org.apache.spark.sql.`**`SQLContext`**	`org.apache.spark.sql.`**`SparkSession`** has **`sqlContext: SQLContext`** as a member
`org.apache.spark.sql.hive.`**`HiveContext`***	Building `org.apache.spark.sql.`**`SparkSession`** with `enableHiveSupport()` adds Hive access to **`sqlContext: SQLContext`**

*__`HiveContext`__ is a subclass of __`SQLContext`__

In Python, package, or module in terms of Python, is not **`org.apache.spark`**, but it is **`pyspark`**, e.g., **`pyspark.sql.SparkSession`**.

`DataFrame` API

- One can think of `DataFrame` as an API for performing SQL-like operations more programmatically
- When reading data from a table (Hive for instance) via **`SQLContext`** or **`SparkSession`**, you get `DataFrame`.
- Can be mapped from `RDD`
  - Explicitly
  - Implicitly

  Examples in the notebook and the repository (src, test)
- Can be converted to `RDD`
  - `RDD[Row]` is returned when calling `.rdd` on the `DataFrame` instance.

`Dataset` API

- Advantages of `RDD` and `DataFrame` APIs rolled into one
  - type-safety of `RDD`
  - Optimizations performed on `DataFrame`
- In Spark 2.x, DataFrame is the type alias to `Dataset[Row]`.

```
type DataFrame = Dataset[Row]
```

https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/package.scala#L46

Spark SQL Data Types

Data types in typical SQL languages are supported.

Data Type	Scala/Java Type	Notes
`IntegerType`, `LongType`	`Int` / `java.lang.Integer*`, `Long` / `java.lang.Long*`	
`FloatType`, `DoubleType`	`Float` / `java.lang.Float*`, `Double` / `java.lang.Double*`	
`BooleanType`	`Boolean` / `java.lang.Boolean*`	
`StringType`	`String` (type alias to `java.lang.String`)	
`TimestampType`, `DateType`	`java.sql.Timestamp`, `java.sql.Date`	No native Scala type support
`ArrayType`	`scala.collection.Seq`	`scala.collection.mutable.WrappedArray` is typically the concrete type (if you need to cast for some reason)
`MapType`	`scala.collection.Map`	
`StructType`	`org.apache.spark.sql.Row`	An actual row in a table is represented by `Row`

*Use Java types if the column is nullable.

**A full list can be found in: https://spark.apache.org/docs/latest/sql-programming-guide.html#data-types

A little bit more about `org.apache.spark.sql.Row`

row_id	name	gender	occupation	alive
1	James McGill 	male	NULL	true
2	Saul Goodman 	male	Lawyer	NULL
3	Walter White 	male	Chemistry Teacher	false
4	Kim Wexler 	female	Lawyer	true

How to represent each row as **`org.apache.spark.sql.Row`**?

A little bit more about `org.apache.spark.sql.Row`

row_id	name	gender	occupation	alive
1	James McGill 	male	NULL	true
2	Saul Goodman 	male	Lawyer	NULL
3	Walter White 	male	Chemistry Teacher	false
4	Kim Wexler 	female	Lawyer	true

`IntType` `StringType` Nullable `StringType` Nullable `BooleanType`

A little bit more about `org.apache.spark.sql.Row`

row_id	name	gender	occupation	alive
1	James McGill 	male	NULL	true
2	Saul Goodman 	male	Lawyer	NULL
3	Walter White 	male	Chemistry Teacher	false
4	Kim Wexler 	female	Lawyer	true

```
import java.{lang => jl} // Side Note: Yes, you can alias imports in Scala.

Row(1, “James McGill”, “male”, null.asInstanceOf[String], true.asInstanceOf[jl.Boolean])
Row(2, “Saul Goodman”, “male”, “Lawyer”, null.asInstanceOf[jl.Boolean])
Row(3, “Walter White”, “male”, “Chemistry Teacher”, false.asInstanceOf[jl.Boolean])
Row(4, “Kim Wexler”, “female”, “Lawyer”, true.asInstanceOf[jl.Boolean])
```

A little bit more about `org.apache.spark.sql.Row`

```
import java.{lang => jl}

Row(1, “James McGill”, “male”, null.asInstanceOf[String], true.asInstanceOf[jl.Boolean])
Row(2, “Saul Goodman”, “male”, “Lawyer”, null.asInstanceOf[jl.Boolean])
Row(3, “Walter White”, “male”, “Chemistry Teacher”, false.asInstanceOf[jl.Boolean])
Row(4, “Kim Wexler”, “female”, “Lawyer”, true.asInstanceOf[jl.Boolean])
```

Note that `Row` instances created in this manner lack schema information.

```
val row = Row(1, “James McGill”, “male”, null.asInstanceOf[String], true.asInstanceOf[jl.Boolean])

row.schema == null // schema is of type StructType
```

`StructType` defines the schema associated with `Row`.

How to create **org.apache.spark.sql.types.StructType**?

If you need to create manually,

1. Define **org.apache.spark.sql.types.StructField** for each column.

```
import org.apache.spark.sql.types.DataTypes._

StructField(“row_id”, IntegerType, nullable = false) // StructFields are nullable by default
StructField(“name”, StringType, nullable = false)
StructField(“gender”, StringType, nullable = false)
StructField(“occupation”, StringType)
StructField(“alive”, BooleanType)
```

2. Construct **org.apache.spark.sql.types.StructType** with the **org.apache.spark.sql.types.StructField** instances.

```
val schema = StructType(Seq(
    StructField(“row_id”, IntegerType, nullable = false),
    StructField(“name”, StringType, nullable = false),
    StructField(“gender”, StringType, nullable = false),
    StructField(“occupation”, StringType),
    StructField(“alive”, BooleanType)))
```

What to do with **org.apache.spark.sql.types.StructType**?

Primary use case in my experience is to create DataFrame

```
import org.apache.spark.sql.types.DataTypes._

val schema = StructType(Seq(
    StructField(“row_id”, IntegerType, nullable = false),
    StructField(“name”, StringType, nullable = false),
    StructField(“gender”, StringType, nullable = false),
    StructField(“occupation”, StringType),
    StructField(“alive”, BooleanType)))

val sqlContext: SQLContext = …

val rdd: RDD[Row] = …

val df: DataFrame = sqlContext.createDataFrame(rdd, schema)
```

There is another way to create `DataFrame` from `RDD[Row]`.

Converting RDD[T] to DataFrame implicitly

via reflection

```
// Instead of StructType, create a case class that matches the table schema
final case class Schema(
  row_id: Int,
  name: String,
  gender: String,
  occupation: String,
  alive: java.lang.Boolean)
// Note that with this approach, `name`, `gender`, and `occupation` are all nullable
// as String is a class

val sqlContext: SQLContext = …

import sqlContext.implicits._ // Remember implicits?

val rdd: RDD[Schema] = … // Do not manually create RDD[Row].

val df: DataFrame = rdd.toDF() // implicits method call (so-called “pimp-my-library” pattern)
```

Don’t do what I just showed you!

Unless you absolutely need to use `RDD` API…

What is the alternative then?

Dataset API

What was the difference between `DataFrame` in Spark 1.x and Spark 2.x?

What was the difference between `DataFrame` in Spark 1.x and Spark 2.x?

`DataFrame` in Spark 1.x is an abstraction, i.e., a `class`, whereas in Spark 2.x, it is merely a `type` alias to `Dataset[Row]`.

How do we convert from `Dataset[T]` to `DataFrame`?

Simply by calling `.toDF()` on the `Dataset` instance. Note that as opposed to Spark 1.x, `.toDF()` in Spark 2.x is not implicit.

How do we convert from `DataFrame` to `Dataset[T]`?

Conversion from DataFrame to Dataset[T]

```
final case class Schema(
  row_id: Int,
  name: String,
  gender: String,
  occupation: String,
  alive: java.lang.Boolean)

val spark: SparkSession = …

import spark.implicits._

val df: DataFrame = …

val ds: Dataset[Schema] = df.as[Schema]
```

Alternatively, Option[Boolean] can be used in Spark 2.x.

Similar to the conversion from RDD[T] to DataFrame in Spark 1.x, Spark 2.x reflectively maps Row to type T specified in .as[T].

Note that .as[T] can be called with any Product type including case classes and Tuples.

DataFrame vs. Dataset

	`DataFrame`	`Dataset`
Data Representation	`org.apache.spark.sql.Row`	Scala classes that extend `Product` and `Serializable` traits, e.g., `case class`. The types that are not implicitly convertible by importing spark.implicits._, which has `Encoders` (`org.apache.spark.sql.Encoder`) for common types pre-defined, one needs to write custom `Encoder` for the type.
Type Safety	No compile-time type safety	Type safety is guaranteed at compile-time
Optimization	Catalyst Optimizer	Catalyst Optimizer
Memory-management	Off-heap	Off-heap
Language Support	Scala, Java, Python, R	Scala, Java

Ways to load data in Spark

- Hive table
  - `spark.table(“database.table_name”)`
- Relational database, e.g., MySQL
  - `spark.read.format(“jdbc”).options(...)`
    https://medium.com/@radek.strnad/tips-for-using-jdbc-in-apache-spark-sql-396ea7b2e3d3
- HDFS
  - `spark.read.{csv, json, parquet, orc}(...)`
- CSV
  - ```
    spark.read
      .option("header", value = true)
      .option("inferSchema", value = true)
      .csv(path)
    ```
    ```

Advices on using SQL as a data source

- The JDBC for the relational database needs to be available.
  - Public Maven repositories have JDBCs for most of major database systems.
    - Surprisingly, it's hard to find Microsoft SQL Server JDBC in Maven repositories.
- If you need to apply `WHERE` and/or `LIMIT` when reading from a SQL table, you need to push the query down `to SQL`.

```
spark.read.format("jdbc")
  .options(...)
  .where(...) // applied after reading from the entire table

// do this instead
spark.read.format("jdbc")
  .option("dbtable", "(SELECT * FROM db.table WHERE ... ) AS alias")
```

Advices on using SQL as a data source

- Increase `numPartitions` for higher parallelism
- Parameters to tweak for improving reading performance
  - `fetchSize`
  - `partitionColumn`, `lowerBound`, `upperBound`
- Parameters to tweak for improving writing performance
  - `batchSize`

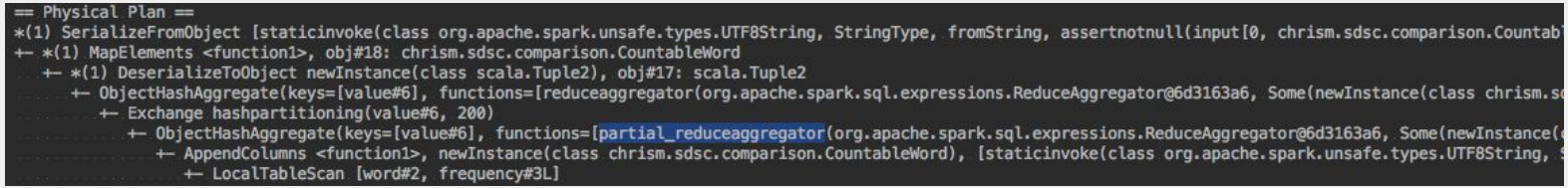
https://spark.apache.org/docs/latest/sql-programming-guide.html#jdbc-to-other-databases

Physical Plan: `DataFrame` vs. `Dataset`

Take a look at `PhysicalPlans.scala` and run `PhysicalPlansTest.scala` in the code example.

Notes:

- `Dataset` physical plan is more complex than that of DataFrame.
- Unless `mapGroups` is more suitable, use `reduceGroups` when processing `KeyValueGroupedDataset`, which is what `groupByKey` returns.
  - reduceGroups uses partial aggregator to reduce the amount of data being transferred cross nodes.



https://github.com/chrismin1202/SDSC2018-Spark-Bootcamp/tree/orlando

Joins

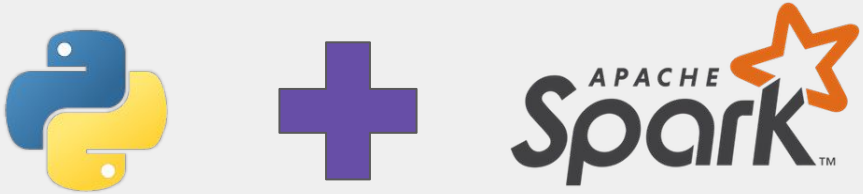
Type	`JoinType` in Spark
Inner	Inner
Outer	FullOuter
Left Outer	LeftOuter

*`JoinTypes` are case-insensitive and you can use underscore instead of camelCasing.

Examples can be found in
https://github.com/chrismin1202/spark-bootcamp/blob/sdsc-2019-atlanta/scala-sbt/src/main/scala/chrism/sdsc/join/joins.scala

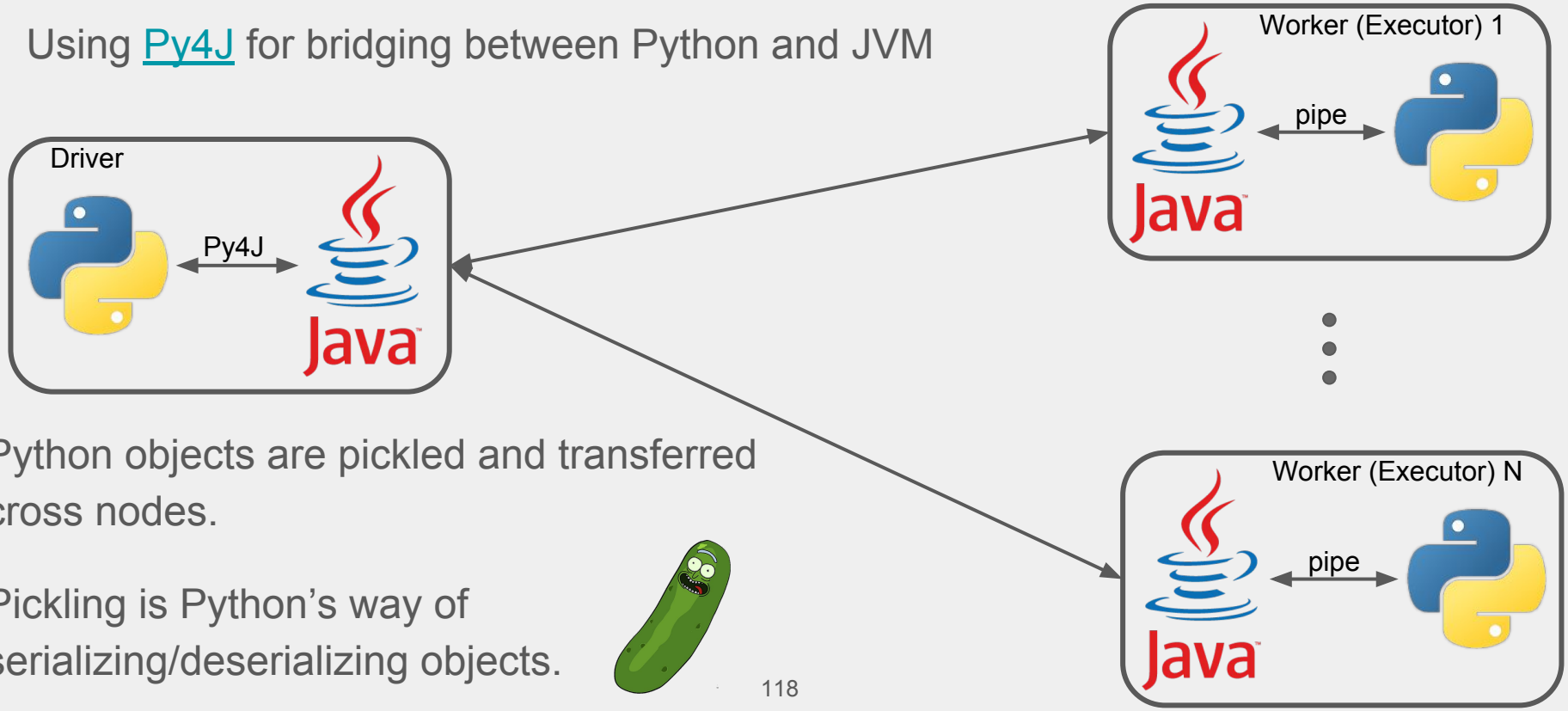
https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-joins.html

A brief introduction to PySpark



How does PySpark work?

Using Py4J for bridging between Python and JVM

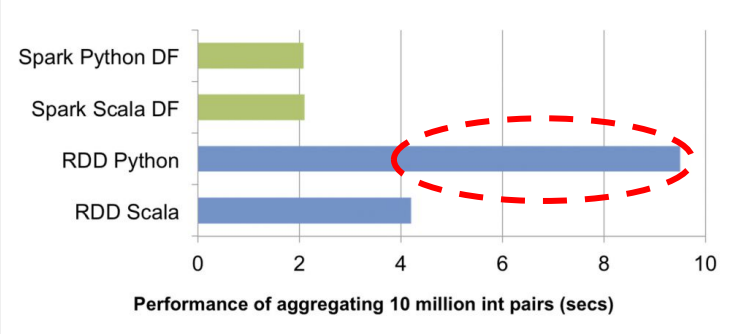


Python objects are pickled and transferred cross nodes.

Pickling is Python's way of serializing/deserializing objects.

What are the drawbacks of PySpark?

- JVM communication overhead
- No `Dataset` API yet
- Pickles are expensive!
  - Pickling (serialization/deserialization) is memory- and CPU-intensive operations.
    Note that this problem is not specific to PySpark. It's just worse in PySpark than in Java/Scala version. One of the reasons Spark came up with `Encoder` was to tackle inefficiencies in serialization/deserialization.



https://databricks.com/blog/2015/04/24/recent-performance-improvements-in-apache-spark-sql-python-dataframes-and-more.html

User-Defined Function (UDF) in Spark SQL

Although Spark SQL has many built-in functions, e.g., `lower`, `upper`, etc., you sometimes need to define your own function (hence “user-defined”).

Example: *You already defined your business logic as a method and you’d like to apply the method to the value(s) in your* `DataFrame`*.*

Assuming that you defined the method in the same language as the language you are using for Spark SQL, you can turn the method into UDF.

Note: As of now, UDF can only take up to 10 parameters.

User-Defined Function (UDF) in Spark SQL

Scala example:

```
import java.{lang => jl}
import org.apache.spark.sql.functions
import org.apache.spark.sql.expressions.UserDefinedFunction

// Suppose there exists a method you'd like to use as a UDF
def toBoolean(i: jl.Integer): jl.Boolean =
  i match {
    case 0 => false
    case 1 => true
    case _ => null
  }
```

User-Defined Function (UDF) in Spark SQL

Scala example continued:

```
def toBoolean(i: jl.Integer): jl.Boolean = …

// To use the method with DataFrame API, wrap the method as UDF
val to_boolean: UserDefinedFunction = functions.udf(toBoolean(_: jl.Integer))

// To actually use the UDF, apply the function to the column(s)
// that you are passing to the UDF.
// Suppose there exists a column named `i` of type INT:
val column = functions.col("i")
val converted_column = to_boolean(column)
// Optionally, you can alias the column by calling `as` or `alias`:
val converted_column = to_boolean(column).as("new_name")
```

More elaborate example can be found

User-Defined Function (UDF) in Spark SQL

Scala example continued:

```
def toBoolean(i: jl.Integer): jl.Boolean = …

// To use the method in SQL literal,
// you need wrap the method as UDF and register it to SparkSession
spark.udf.register("to_boolean", toBoolean(_: jl.Integer))

// Once registered, you can apply the UDF just like built-in UDFs.
// Suppose there exists a column named `i` of type INT:
spark.sql("SELECT to_boolean(`i`) AS `new_name` FROM `db`.`table`" )
```

More elaborate example can be found in the

User-Defined Function (UDF) in Spark SQL

Python example:

```
from pyspark.sql import java.{lang => jl}
import org.apache.spark.sql.functions
import org.apache.spark.sql.expressions.UserDefinedFunction

# Suppose there exists a method you'd like to use as a UDF
def to_boolean(num):
  if num == 0:
    return False
  elif num == 1:
    return True
  else:
    return None
```

User-Defined Function (UDF) in Spark SQL

Python example continued:

```
from pyspark.sql import functions
from pyspark.sql.types import BooleanType

to_boolean(): # takes in an INT and converts to BOOLEAN

# Similar to Scala version, you wrap the method as a UDF.
to_boolean_udf = functions.udf(to_boolean, BooleanType())

# Apply the function to the applicable column(s).
# Suppose there exists a column named `i` of type INT:
val column = functions.col("i")
val converted_column = to_boolean(column)
# Optionally, you can alias the column by calling `alias`:
val converted_column = to_boolean(column).alias("new_name")
```

User-Defined Function (UDF) in Spark SQL

Python example continued:

```
to_boolean(): # takes in an INT and converts to BOOLEAN

# Again, to use the method in SQL literal,
# you need wrap the method as UDF and register it to SparkSession
spark.udf.register("to_boolean", to_boolean, "BOOLEAN")

# Once registered, you can apply the UDF just like built-in UDFs.
# Suppose there exists a column named `i` of type INT:
spark.sql("SELECT to_boolean(`i`) AS `new_name` FROM `db`.`table`" )
```

User-Defined Function (UDF) in Spark SQL

More elaborate examples can be found in the notebook and the repository.

- Notebook
- Scala example: src, test
- Python example: src, test

UDFs in PySpark

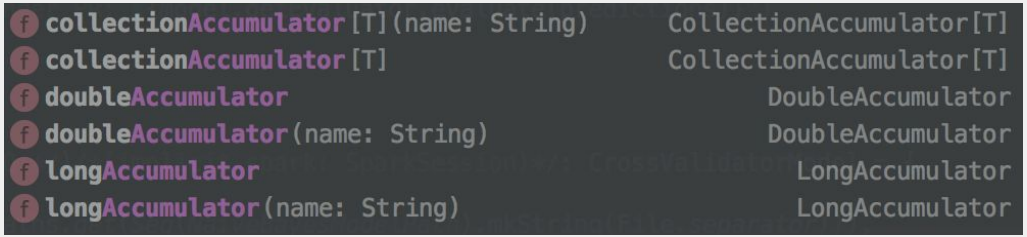
- Tends to be slower than Scala UDFs due to serialization tax.
  - You incur serialization overhead for each invocation (each row) of the UDF
- Scala UDFs can be used.
  - You need to compile a jar with UDF(s) you need and pass it to `spark-submit` with `--jar` switch.
  - If you are interested, take a look at this Medium post.

User-Defined Aggregate Function (UDAF) in Spark SQL

- UDAFs are more complex to define than UDFs.
  - Scala example: https://docs.databricks.com/spark/latest/spark-sql/udaf-scala.html
- It is even more difficult in Python.

Shared Variable

- Broadcast variable
  - A read-only variable that is shipped to all executors and cached in each node
- Accumulator
  - A shared variable that can be “accumulated” by executors
  - Similar to counters in MapReduce
  - Viewable in the web UI
  - Has native support for numeric types



Broadcast Variable

```
val sc: SparkContext = …

val lookUpMap = Map(1 -> “One”, 2 -> “Two”, 3 -> “Three”, … )

// lookUpMap is of type Map[Int, String],
// whereas broadcastLookUpMap is of type Broadcast[Map[Int, String]]
val broadcastLookUpMap = sc.broadcast(lookUpMap)

// To use in an executor
broadcastLookUpMap.value
// The copy cached in the executor’s node is used
```

Accumulator

```
val sc: SparkContext = …

// built-in accumulators
val doubleAccum = sc.doubleAccumulator(“Double Counter”)
val longAccum = sc.longAccumulator(“Integer Counter”)

// To use in an executor
doubleAccum.add(3.14)
longAccum.add(1L)

// Note that the methods that access the accumulated value
// should be called in the driver only
```

How to submit Spark Applications?

Using `spark-submit` script

The script `spark-submit` is in Spark's `bin` directory. In a Spark cluster (typically a Hadoop cluster with Spark installed), `spark-submit` script should be available as a command.

```
spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Example

```
spark-submit \
  --class chrism.sdsc.wordcount.WordCount \
  --master local[*] \
  --deploy-mode client \
  --driver-memory 1g \
  --executor-memory 2g \
  --conf spark.executor.memoryOverhead=512 \
  /location/to/SDSC2018-Spark-Workshop-assembly-0.0.1.jar
```

https://spark.apache.org/docs/latest/submitting-applications.html

Spark ML Overview

Machine Learning in Spark

- **`spark.mllib.*`**
  - **`RDD`**-based API
  - As of Spark 2.0, this package is in maintenance mode.

- **`spark.ml.*`**
  - **`DataFrame`**-based API
  - Unless you must use RDD directly, this packages should be preferred

https://spark.apache.org/docs/latest/ml-guide.html

`spark.ml` Pipelines

- Built with **`DataFrame`** API
  - Supports all Spark SQL data types
- Components:
  - Transformer
  - Estimator
  - Parameter
  - Pipeline

https://spark.apache.org/docs/latest/ml-pipeline.html

ML Pipelines

- Components:
  - **Transformer**
  - Estimator
  - Parameter
  - Pipeline

https://spark.apache.org/docs/latest/ml-pipeline.html

Transformer

```
abstract class Transformer extends PipelineStage {
  def transform(dataset: Dataset[_]): DataFrame
  // And several other overloads and methods
}
```

Transformer

DataFrame is a type alias for **Dataset[Row]** in Spark 2

```
def transform(dataset: Dataset[_]): DataFrame
```

ML Pipelines

- Components:
  - Transformer
  - **Estimator**
  - Parameter
  - Pipeline

https://spark.apache.org/docs/latest/ml-pipeline.html

Estimator

```
abstract class Estimator[M <: Model[M]]
  extends PipelineStage {

  def fit(dataset: Dataset[_]): M

  // And several other overloads and methods
}
```

Estimator

Takes in a `Dataset[_]` and produces a model that “fits” the given `Dataset[_]`, i.e., this is where you define the learning algorithm.

```
abstract class Model[M <: Model[M]] extends Transformer
```

```
def fit(dataset: Dataset[_]): M
```

ML Pipelines

- Components:
  - Transformer
  - Estimator
  - **Parameter**
  - Pipeline

https://spark.apache.org/docs/latest/ml-pipeline.html

Parameters

The parameters for **`Transformer`**s and **`Estimator`**s such as the number of iterations and various thresholds.

- `Param[T]`
  - A parameter “key” that holds name, documentation, and validation logic
- `ParamPair[T]`
  - A key-value pair where the key is of type `Param[T]` and the value is of type `T`
- `ParamMap`
  - A map that hold parameter pairs
- ParamGridBuilder
  - `.build()` returns `Array[ParamMap]`, which contains all combinations specified by the grid

https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/ml/param/params.scala

https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/ml/param/shared/sharedParams.scala

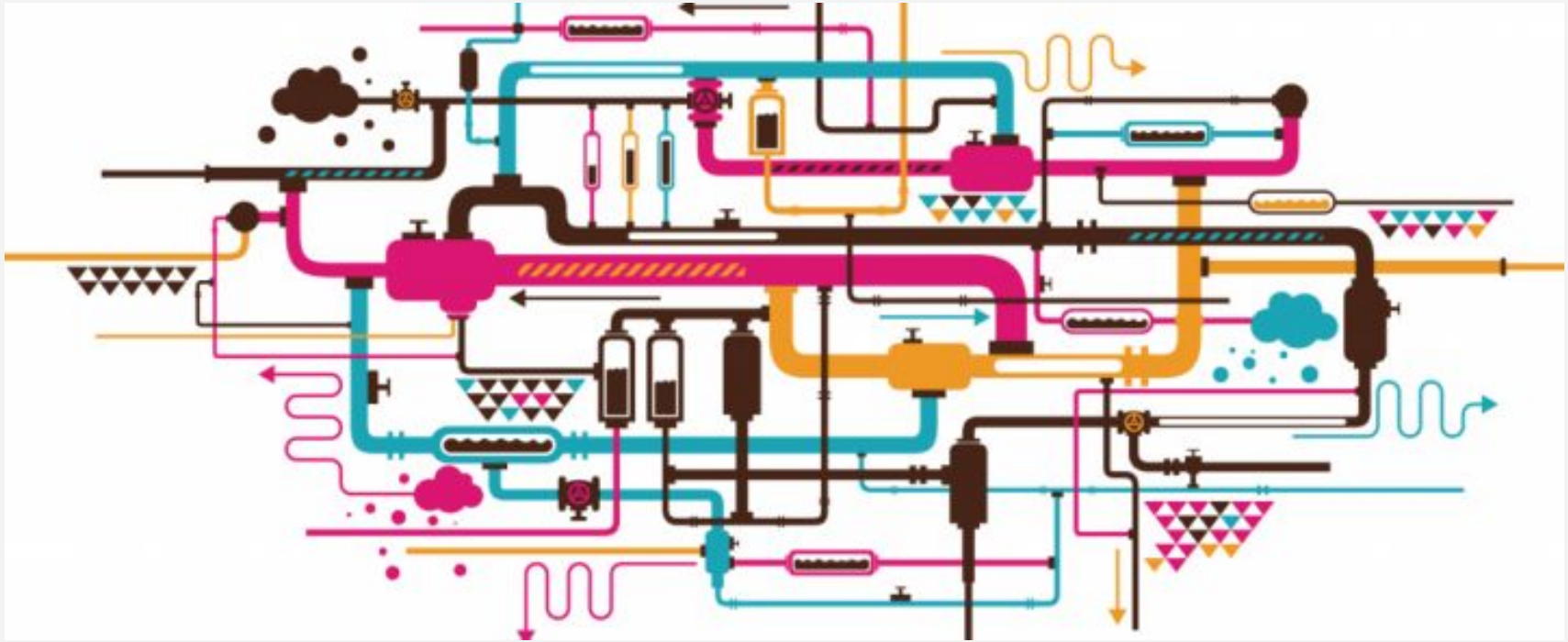
https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/ml/tuning/ParamGridBuilder.scala

ML Pipelines

- Components:
  - Transformer
  - Estimator
  - Parameter
  - **Pipeline**

https://spark.apache.org/docs/latest/ml-pipeline.html

Pipeline



Pipeline

A workflow that specifies a sequence of stages

Pipeline

A workflow that specifies a sequence of stages

A *stage* is either a **`Transformer`** or a **`Model`**.

A pipeline must form a DAG
(**D**irected **A**cyclic **G**raph)

Let’s actually build a pipeline!

http://tinyurl.com/sdsc-2019-atl-ml

Spark on Kubernetes Demo

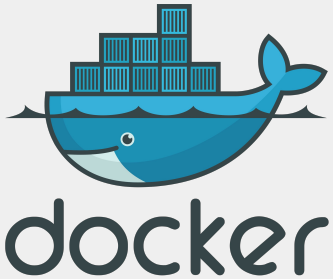
kubernetes

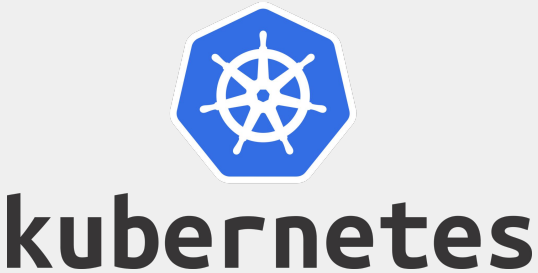
Things to note about running Spark on Kubernetes

- Native support for Kubernetes in Spark 2.3.0 and higher.
  - Was introduced as an experimental feature in Spark 2.2.0.
- **Resource Staging Server** (RSS) has not been officially adopted.
  - RSS is a “proposed” way to manage dependencies for running Spark on Kubernetes.
  - apache-spark-on-k8s has been deprecated (circa Spark 2.2.0) and its features are gradually ported to Spark’s source code (since Spark 2.3.0).
- Need a Kubernetes cluster (obviously…).
- Need custom (Docker) image.
  - To run a custom Spark application, all of its dependencies need to be included in the image.
  - Spark is shipped with an example Dockerfile
    - In **`kubernetes/dockerfiles/spark`** directory of the official releases or in **`resource-managers/kubernetes/docker/src/main/dockerfiles/spark`** directory of the source code.

To run the example, you need

Refer to the project README for details.





PySpark + Kubernetes = ?

- PySpark Dockerfile
  https://github.com/apache/spark/blob/branch-2.4/resource-managers/kubernetes/docker/src/main/dockerfiles/spark/bindings/python/Dockerfile
- Python dependency management*
  - In the Docker image for the executors, you can include all dependencies you need instead of packaging dependencies and shipping via `--py-files` every time.
  - Many ML libraries have C or C++ dependencies and if you are not the one who manages the cluster on which you run your Spark applications, e.g., Hadoop, you need to ask the admin to deploy the dependencies for you or you need to figure out another way to deploy them to all worker nodes.

**I have not actually tried myself, but in Hadoop, one of the ways to manage PySpark (Python dependencies in general) is to install them in all nodes.*

```
18  ARG base_img
19  FROM $base_img
20  WORKDIR /
21  RUN mkdir ${SPARK_HOME}/python
22  # TODO: Investigate running both pip and pip3 via virtualenvs
23  RUN apk add --no-cache python && \
24      apk add --no-cache python3 && \
25      python -m ensurepip && \
26      python3 -m ensurepip && \
27      # We remove ensurepip since it adds no functionality since pip is
28      # installed on the image and it just takes up 1.6MB on the image
29      rm -r /usr/lib/python*/ensurepip && \
30      pip install --upgrade pip setuptools && \
31      # You may install with python3 packages by using pip3.6
32      # Removed the .cache to save space
33      rm -r /root/.cache
34
35  COPY python/lib ${SPARK_HOME}/python/lib
36  ENV PYTHONPATH ${SPARK_HOME}/python/lib/pyspark.zip:${SPARK_HOME}/python/lib/py4j-*.zip
37
38  WORKDIR /opt/spark/work-dir
39  ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

https://github.com/apache/spark/blob/branch-2.4/resource-managers/kubernetes/docker/src/main/dockerfiles/spark/bindings/python/Dockerfile

Some Tips…

- Careful with Java `enum` when it needs to be used as the key in a Spark job
  - Partitioners are typically hash-based and Java `enum`'s `hashCode` is different for each JVM instance and it is not overridable.

  http://dev.bizo.com/2014/02/beware-enums-in-spark.html

- Consider using Hive for IO
  - Your application does not need to be tied to the file formats of input(s) and output(s) as Hive-enabled `SparkSession` or `HiveContext` handles reading and writing.
- Skewed data
  - When data is skewed, that is, when certain partitions hold a lot more data than other partitions, consider calling `.repartition()` or `.coalesce()` either when you read the data (if the input data is skewed) or when you write the data (if your job somehow produces skewed data).

More Tips…

- Task Not Serializable!
  - The transformation code that are serialized from the driver and shipped to executors must be serializable.

Tips on shuffles

- Shuffles
  - RDD
    - The transformations that require shuffle typically takes in the number of partitions as a parameter.
    - The default number of partitions used in shuffle is *`max(rdd1.partitions.length, rdd2.partitions.length)`*
    - How many is appropriate?
      - Depends on your application (amount of data, how CPU- and memory-intensive your application is) and the amount of resources that can be allocated to your job
      - You need to experiment.
      - I was almost always able to find the appropriate number of partitions somewhere between *`max(rdd1.partitions.length, rdd2.partitions.length)`* and *`rdd1.partitions.length + rdd2.partitions.length`*.

Tips on shuffles continued…

- `Dataset`/`DataFrame` shuffles
  - Unlike RDD API, the number of shuffle partitions cannot be controlled dynamically in Spark SQL jobs.
  - One needs to set `spark.sql.shuffle.partitions` (default is 200) when submitting the application, e.g., `--conf spark.sql.shuffle.partitions=5003`
  - For large `Datasets`, the default value of 200 is almost always not large enough.

Last but not least

- Memory configuration in YARN (Refer to the slide about memory allocation in YARN)
  - Try allocating more heap memory, i.e., executor memory, when you get **`OutOfMemoryError`** thrown or the exit code is **137**.
  - Try allocating more memory overhead if your executors are killed by YARN. The error message typically tells you to boost memory overhead.

Questions?


Thank you