BIG DATA ANALYTICS

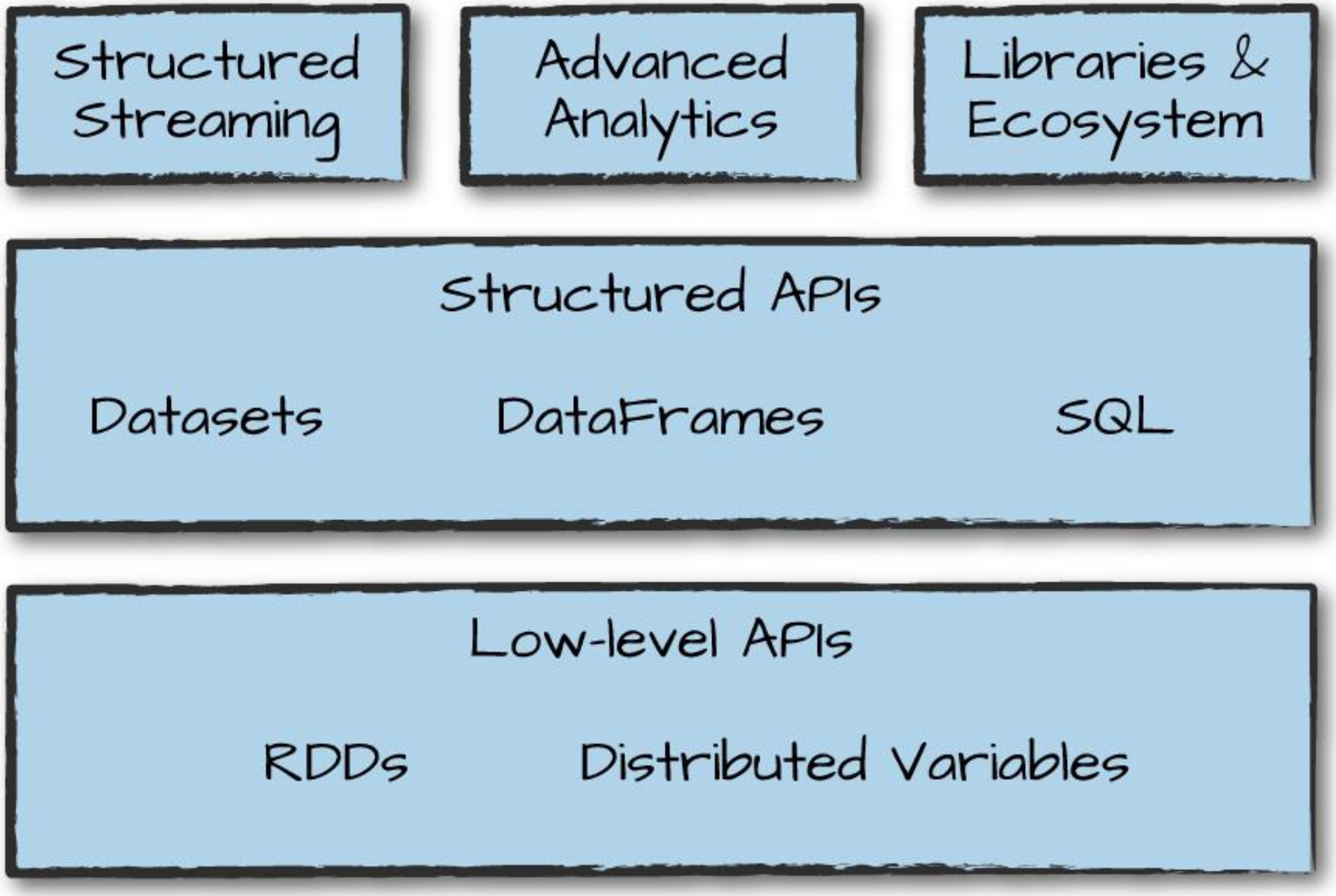
Spark's Toolset

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset

Figure. Spark's toolset



Spark's Toolset

- Spark's core concepts, like transformations and actions, in the context of Spark's Structured APIs, these simple conceptual building blocks are the foundation of Apache Spark's vast ecosystem of tools and libraries (shown in the *above Figure*)
- Spark is composed of these primitives-the lower-level APIs and the Structured APIs-and then a series of standard libraries for additional functionality.
- Spark's libraries support a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of computing and storage systems.

Spark's Toolset

This lecture covers the following:

- Running production applications with `spark-submit`
- Datasets: type-safe APIs for structured data
- Structured Streaming
- Machine learning and advanced analytics
- Resilient Distributed Datasets (RDD): Spark's low level APIs
- SparkR
- The third-party package ecosystem

Spark's Toolset: *Outline*

- Running Production Applications 
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset: Running production applications

- Spark makes it easy to develop and create big data programs.
- Spark also makes it easy to turn our interactive exploration into production applications with `spark-submit`, a built-in command-line tool.
- `spark-submit` does one thing: it lets us send our application code to a cluster and launch it to execute there.
- Upon submission, the application will run until it exits (completes the task) or encounters an error.
- We can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN.

Spark's Toolset: Running production applications

- `spark-submit` offers several controls with which we can specify the resources our application needs as well as how it should be run and its command-line arguments.
- We can write applications in any of Spark's supported languages and then submit them for execution.
- The simplest example is running an application on our local machine. We'll show this by running a sample Scala application that comes with Spark, using the following command in the directory where you downloaded Spark:

```
./bin/spark-submit \
      --class org.apache.spark.examples.SparkPi \
      --master local \
      ./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

Spark's Toolset: Running production applications

- This sample application calculates the digits of pi to a certain level of estimation.
- Here, we've told **`spark-submit`** that we want to run on our local machine, which class and which JAR we would like to run, and some command-line arguments for that class.
- We can also run a Python version of the application using the following command:

```
./bin/spark-submit \
--master local \
./examples/src/main/python/pi.py 10
```

- By changing the master argument of **`spark-submit`**, we can also submit the same application to a cluster running Spark's standalone cluster manager, Mesos or YARN.

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs 
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset: Datasets: Type-Safe Structured APIs

- The first API we'll describe is a type-safe version of Spark's structured API called *Datasets*, for writing statically typed code in Java and Scala.
- The Dataset API is not available in Python and R, because those languages are dynamically typed.
- DataFrames are a distributed collection of objects of type Row that can hold various types of tabular data.
- The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java `ArrayList` or Scala `Seq`.
- The APIs available on Datasets are type-safe, meaning that we cannot accidentally view the objects in a Dataset as being of another class than the class we put in initially. This makes Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

Spark's Toolset: Datasets: Type-Safe Structured APIs

- The `Dataset` class is parameterized with the type of object contained inside: `Dataset<T>` in Java and `Dataset[T]` in Scala.
- For example, a `Dataset[Person]` will be guaranteed to contain objects of class `Person`.
- As of Spark 2.0, the supported types are classes following the JavaBean pattern in Java and case classes in Scala. These types are restricted because Spark needs to be able to automatically analyze the type `T` and create an appropriate schema for the tabular data within our Dataset.
- One great thing about Datasets is that we can use them only when we need or want to.

Spark's Toolset: Datasets: Type-Safe Structured APIs

- In this example, we'll create a custom data type and manipulate it using map and filter functions.
- After processing, Spark automatically converts it back into a DataFrame, allowing further manipulation using its built-in functions.
- This flexibility lets us switch between low-level, type-safe coding and high-level SQL queries for faster analysis.
- Here's a simple example that combines type-safe functions with SQL-like expressions for efficient business logic.

```
// in Scala
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)
val flightsDF = spark.read.parquet("/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

Spark's Toolset: Datasets: Type-Safe Structured APIs

- One final advantage is that when we call `collect` or `take` on a Dataset, it will collect objects of the proper type in our Dataset, not DataFrame `Rows`. This makes it easy to get type safety and securely perform manipulation in a distributed and a local manner without code changes:

```
// in Scala
 flights
   .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
   .map(flight_row => flight_row)
   .take(5)
 flights
   .take(5)
   .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
   .map(fr => Flight(fr.DEST_COUNTRY_NAME, fr.ORIGIN_COUNTRY_NAME, fr.count +
       5))
```

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming 
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset: Structured Streaming

What is Structured Streaming?

- High-level API for stream processing introduced in Spark 2.2.
- Allows batch operations to run in streaming fashion.
- Reduces latency and enables incremental processing with minimal code changes.
- Easily convert batch jobs into streaming jobs.

Spark's Toolset: Structured Streaming

Example: Retail Data Streaming

1. Static Data Analysis

- Read retail data (CSV) as a `static DataFrame`:

```
// in Scala
val staticDataFrame = spark.read.format("csv")
.option("header", "true")
.option("inferSchema", "true")
.load("/data/retail-data/by-day/*.csv")
staticDataFrame.createOrReplaceTempView("retail_data")
val staticSchema = staticDataFrame.schema
// in Python
staticDataFrame = spark.read.format("csv")\
.option("header", "true")\
.option("inferSchema", "true")\
.load("/data/retail-data/by-day/*.csv")
staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

Spark's Toolset: Structured Streaming

Example: Retail Data Streaming

1. Static Data Analysis

- Calculate total cost per day:

```
// in Scala
import org.apache.spark.sql.functions.{window, column, desc, col}
staticDataFrame
  .selectExpr(
   "CustomerId",
   "(UnitPrice * Quantity) as total_cost",
   "InvoiceDate")
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))
  .sum("total_cost")
  .show(5)
```

Spark's Toolset: Structured Streaming

Example: Retail Data Streaming

1. Static Data Analysis

- Calculate total cost per day:

```
// in Python
from pyspark.sql.functions import window, column, desc, col
staticDataFrame\
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")\
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
  .sum("total_cost")\
  .show(5)
```

Spark's Toolset: Structured Streaming

Example: Retail Data Streaming

1. Static Data Analysis

- Here's a sample of the output that we'll see:

```
+----------+--------------------+------------------+
|CustomerId|              window| sum(total_cost)|
+----------+--------------------+------------------+
| 17450.0|[2011-09-20 00:00...| 71601.44|
...
| null|[2011-12-08 00:00...|31975.590000000007|
+----------+--------------------+------------------+
```

Spark's Toolset: Structured Streaming

Streaming Data Processing

- Use `readStream` instead of read for streaming:

```
// in Scala
val streamingDataFrame = spark.readStream
     .schema(staticSchema)
     .option("maxFilesPerTrigger", 1)
     .format("csv")
     .option("header", "true")
     .load("/data/retail-data/by-day/*.csv")
// in Python
streamingDataFrame = spark.readStream\
    .schema(staticSchema)\
    .option("maxFilesPerTrigger", 1)\
    .format("csv")\
    .option("header", "true")\
    .load("/data/retail-data/by-day/*.csv")
```

Spark's Toolset: Structured Streaming

Streaming Data Processing

- Apply same streaming logic:

```
// in Scala
val purchaseByCustomerPerHour = streamingDataFrame
   .selectExpr(
     "CustomerId",
     "(UnitPrice * Quantity) as total_cost",
     "InvoiceDate")
   .groupBy(
     $"CustomerId", window($"InvoiceDate", "1 day"))
   .sum("total_cost")
// in Python
purchaseByCustomerPerHour = streamingDataFrame\
  .selectExpr(
    "CustomerId",
    "(UnitPrice * Quantity) as total_cost",
    "InvoiceDate")\
  .groupBy(
    col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
  .sum("total_cost")
```

Spark's Toolset: Structured Streaming

Output and Action

- Output to in-memory table:

```
// in Scala
purchaseByCustomerPerHour.writeStream
     .format("memory") // memory = store in-memory table
     .queryName("customer_purchases") // the name of the in-memory table
     .outputMode("complete") // complete = all the counts should be in the table
     .start()

// in Python
purchaseByCustomerPerHour.writeStream\
    .format("memory")\
    .queryName("customer_purchases")\
    .outputMode("complete")\
    .start()
```

Spark's Toolset: Structured Streaming

Output and Action

- Query in-memory results:

```
// in Scala
spark.sql("SELECT * FROM customer_purchases ORDER BY `sum(total_cost)` DESC").show(5)
// in Python
spark.sql("SELECT * FROM customer_purchases ORDER BY `sum(total_cost)` DESC").show(5)
```

This provides an overview of how Structured Streaming allows us to easily work with both batch and streaming data using Spark, enabling efficient, real-time data processing.

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics 
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset: Machine Learning and Advanced Analytics

Using Apache Spark's MLlib for Large-scale Machine Learning: Overview of Spark's built-in MLlib library for handling machine learning at scale.

Introduction to Spark's Mllib

- Built-in library for machine learning.
- Enables large-scale model training and predictions.
- Applicable to Structured Streaming for real-time predictions.
- Supports classification, regression, clustering, and deep learning.

Spark's Toolset: Machine Learning and Advanced Analytics

Data Preparation:

Data Preprocessing in Spark

- Transformation of raw data to numerical values for ML algorithms.
- Example schema for data:

```
staticDataFrame.printSchema()
```

```
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```

Spark's Toolset: Machine Learning and Advanced Analytics

Data Transformation:

Data Transformation for K-means Clustering

Scala Code:

```
import org.apache.spark.sql.functions.date_format
val preppedDataFrame = staticDataFrame
  .na.fill(0)
  .withColumn("day_of_week", date_format($"InvoiceDate", "EEEE"))
  .coalesce(5)
```

Python Code:

```
from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
  .na.fill(0)\
  .withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
  .coalesce(5)
```

Spark's Toolset: Machine Learning and Advanced Analytics

Data Splitting:

Splitting Data into Training and Test Sets

Scala Code:

```
val trainDataFrame = preppedDataFrame
  .where("InvoiceDate < '2011-07-01'")
val testDataFrame = preppedDataFrame
  .where("InvoiceDate >= '2011-07-01'")
```

Python Code:

```
trainDataFrame = preppedDataFrame.where("InvoiceDate < '2011-07-01'")
testDataFrame = preppedDataFrame.where("InvoiceDate >= '2011-07-01'")
```

Spark's Toolset: Machine Learning and Advanced Analytics

String Indexing and Encoding:

String Indexing and One-Hot Encoding

Scala Code:

```
// String Indexing
import org.apache.spark.ml.feature.StringIndexer
val indexer = new StringIndexer()
  .setInputCol("day_of_week")
  .setOutputCol("day_of_week_index")

// One-Hot Encoding
import org.apache.spark.ml.feature.OneHotEncoder
val encoder = new OneHotEncoder()
  .setInputCol("day_of_week_index")
  .setOutputCol("day_of_week_encoded")
```

Spark's Toolset: Machine Learning and Advanced Analytics

String Indexing and Encoding:

String Indexing and One-Hot Encoding

Python Code:

```
// String Indexing
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
  .setInputCol("day_of_week")\
  .setOutputCol("day_of_week_index")

// One-Hot Encoding
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder()\
  .setInputCol("day_of_week_index")\
  .setOutputCol("day_of_week_encoded")
```

Spark's Toolset: Machine Learning and Advanced Analytics

Assembling Features:

Vector Assembler for Feature Engineering

Scala Code:

```
import org.apache.spark.ml.feature.VectorAssembler
val vectorAssembler = new VectorAssembler()
  .setInputCols(Array("UnitPrice", "Quantity", "day_of_week_encoded"))
  .setOutputCol("features")
```

Python Code:

```
from pyspark.ml.feature import VectorAssembler
vectorAssembler = VectorAssembler()\
  .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
  .setOutputCol("features")
```

Spark's Toolset: Machine Learning and Advanced Analytics

Building the Pipeline:

Creating a Transformation Pipeline

Scala Code:

```
import org.apache.spark.ml.Pipeline
val transformationPipeline = new Pipeline()
  .setStages(Array(indexer, encoder, vectorAssembler))
```

Python Code:

```
from pyspark.ml import Pipeline
transformationPipeline = Pipeline()\
  .setStages([indexer, encoder, vectorAssembler])
```

Spark's Toolset: Machine Learning and Advanced Analytics

Model Training:

Training K-Means Clustering Model

Scala Code:

```
import org.apache.spark.ml.clustering.KMeans
val kmeans = new KMeans()
  .setK(20)
  .setSeed(1L)
val kmModel = kmeans.fit(transformedTraining)
```

Python Code:

```
from pyspark.ml.clustering import KMeans
kmeans = KMeans()\
  .setK(20)\
  .setSeed(1L)
kmModel = kmeans.fit(transformedTraining)
```

Spark's Toolset: Machine Learning and Advanced Analytics

Evaluating the Model:

Model Evaluation and Cost Calculation

Scala Code:

```
kmModel.computeCost(transformedTraining)
val transformedTest = fittedPipeline.transform(testDataFrame)
kmModel.computeCost(transformedTest)
```

Python Code:

```
kmModel.computeCost(transformedTraining)
transformedTest = fittedPipeline.transform(testDataFrame)
kmModel.computeCost(transformedTest)
```

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs 
- SparkR
- Spark's Ecosystem and Packages

Spark's Toolset: Lower-Level APIs

- Spark includes a number of lower-level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs).
- Virtually everything in Spark is built on top of RDDs.
- DataFrame operations are built on top of RDDs and compile down to these lower-level tools for convenient and extremely efficient distributed execution.
- There are some things that we might use RDDs for, especially when we're reading or manipulating raw data, but for the most part we should stick to the Structured APIs.
- RDDs are lower level than DataFrames because they reveal physical execution characteristics (like partitions) to end users.

Spark's Toolset: Lower-Level APIs

- One thing that we might use RDDs for is to parallelize raw data that we have stored in memory on the driver machine.
- For instance, let's parallelize some simple numbers and create a DataFrame after we do so.
- We then can convert that to a DataFrame to use it with other DataFrames:

```
// in Scala
spark.sparkContext.parallelize(Seq(1, 2, 3)).toDF()

# in Python
from pyspark.sql import Row
spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toDF()
```

Spark's Toolset: Lower-Level APIs

RDDs vs. DataFrames in Spark

- RDDs (Resilient Distributed Datasets):
  - Available in both Scala and Python, but with distinct execution characteristics due to underlying implementations.
  - Primarily used for handling very raw, unprocessed, and unstructured data.
  - Less frequently needed for modern Spark applications, mainly relevant for maintaining older code bases.
- DataFrames
  - Uniform execution characteristics across Scala and Python due to the DataFrame API.
  - Recommended for most data tasks in modern Spark environments due to better optimization and structure.
- Note: The transition from RDDs to DataFrames signifies a move towards more structured and efficient data handling within Spark's ecosystem.

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR 
- Spark's Ecosystem and Packages

Spark's Toolset: SparkR

- SparkR is a tool for running R on Spark.
- It follows the same principles as all of Spark's other language bindings.
- To use SparkR, we simply import it into our environment and run our code.
- It's all very similar to the Python API except that it follows R's syntax instead of Python.
- For the most part, almost everything available in Python is available in SparkR:

```
# in R
library(SparkR)
sparkDF <- read.df("/data/flight-data/csv/2015-summary.csv",
source = "csv", header="true", inferSchema = "true")
take(sparkDF, 5)

# in R
collect(orderBy(sparkDF, "count"), 20)
```

Spark's Toolset: SparkR

- R users can also use other R libraries like the pipe operator in magrittr to make Spark transformations a bit more R-like.
- This can make it easy to use with other libraries like ggplot for more sophisticated plotting:

```
# in R
library(magrittr)
sparkDF %>%
  orderBy(desc(sparkDF$count)) %>%
  groupBy("ORIGIN_COUNTRY_NAME") %>%
  count() %>%
  limit(10) %>%
  collect()
```

We will not include R code samples as we do in Python, because almost every concept that applies to Python also applies to SparkR. The only difference will by syntax.

Spark's Toolset: *Outline*

- Running Production Applications
- Datasets: Type-Safe Structured APIs
- Structured Streaming
- Machine Learning and Advanced Analytics
- Lower-Level APIs
- SparkR
- Spark's Ecosystem and Packages 

Spark's Toolset: Spark's Ecosystem and Packages

- One of the best parts about Spark is the ecosystem of packages and tools that the community has created.
- Some of these tools even move into the core Spark project as they mature and become widely used.
- As of this writing, the list of packages is rather long, numbering over 500-and more are added frequently (updated as on 23-09-2024).
- We can find the largest index of Spark Packages at `spark-packages.org`, where any user can publish to this package repository.
- There are also various other projects and packages that we can find on the web; for example, on GitHub.

Spark's Toolset: Spark's Ecosystem and Packages

Sample list of Apache Spark Packages:

- Core (15): spark-indexedrdd, spark-sorted, elasticsearch-Hadoop, spark-skewjoin, spark-tutorial, spark-crossdata, spark-metrics, rich-spark, spark-mergejoin etc.
- Data Sources (57): spark-avro, spark-redshift, spark-csv, deep-spark, pyspark-csv, spark-mongodb, pipeline, spark-xml, spark-netflow, spark-kafka-writer etc.
- Machine Learning (102): generalized-kmeans-clustering, spark-ml-streaming, MLlib-dropout, spark-ml-class, sparkit-learn, spark-knn-graphs, ScalaNetwork etc.
- Streaming (63): kafka-spark-consumer, spark-ml-streaming, spark, spark-testing-base, spark-kafka, spark-power-bi, spark-streamingsql, pipeline etc.
- Graph (21): spark-knn-graphs, TopicModeling, pipeline, graphx-diameter, graphx-scala, graphframes etc.
- PySpark (25): pyspark-cassandra, pyspark-notebook, smartframes, spark_jupyter etc.
- Applications (17): spark-jobserver, spark-notebook, pipeline, sparkplug, spark-crossdata, spark-metrics, spark-tools etc.
- Deployment (12): spark_azure, spark-deployer, pipeline, sparkplug, spark-openstack etc.

…. And so on