

Apache Spark 2.x

"FROM BASICS TO ADVANCED CONCEPTS"

What is Apache Spark?

Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Key Features:

- **Batch/streaming data:** Unify the processing of your data in batches and real-time streaming, using your preferred language: Python, SQL, Scala, Java or R.
- **SQL analytics:** Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.
- **Data science at scale:** Perform Exploratory Data Analysis (EDA) on petabyte-scale data without having to resort to downsampling.
- **Machine learning:** Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.
- Simple. Fast. Scalable. Unified.

History of Apache Spark

Apache Spark **began as a research project at the University of California, Berkeley's AMPLab in 2009.**

The project was initially led by Matei Zaharia, a Ph.D. student at the time.

Spark was designed to address the limitations of Hadoop MapReduce, particularly its slow iterative processing and lack of support for interactive data analysis.

Key Milestones:

- **2009: Spark was created as a research project at UC Berkeley's AMPLab.**
- 2010: Spark was open-sourced, allowing the community to contribute to its development.
- 2013: Spark was donated to The Apache Software Foundation, becoming an Apache incubator project.
- 2014: Spark graduated from the Apache incubator to become a top-level Apache project.

History of Apache Spark

Evolution

Spark's early focus was on fast iterative processing, making it suitable for machine learning and interactive data analysis. Over time, Spark evolved to incorporate additional features, such as:

- In-memory computing for improved performance
- Support for structured and semi-structured data
- Integration with Hadoop and other big data ecosystems
- Extensive libraries and APIs for data processing and analytics

History of Apache Spark

Impact

Apache Spark has had a significant impact on the big data and analytics landscape, enabling faster and more efficient data processing, and supporting a wide range of applications, including:

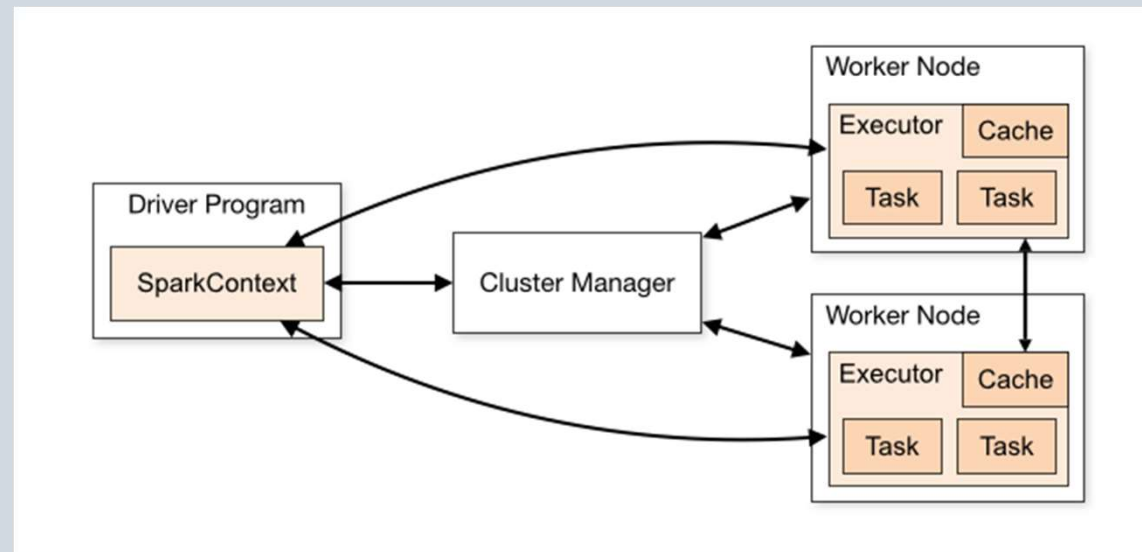
- Machine learning and artificial intelligence
- Data science and analytics
- Real-time data processing and streaming
- IoT and edge computing

Today, Apache Spark is widely used in industry and academia, and is considered one of the most popular and influential open-source projects in the big data space.

Spark Architecture

Main Components:

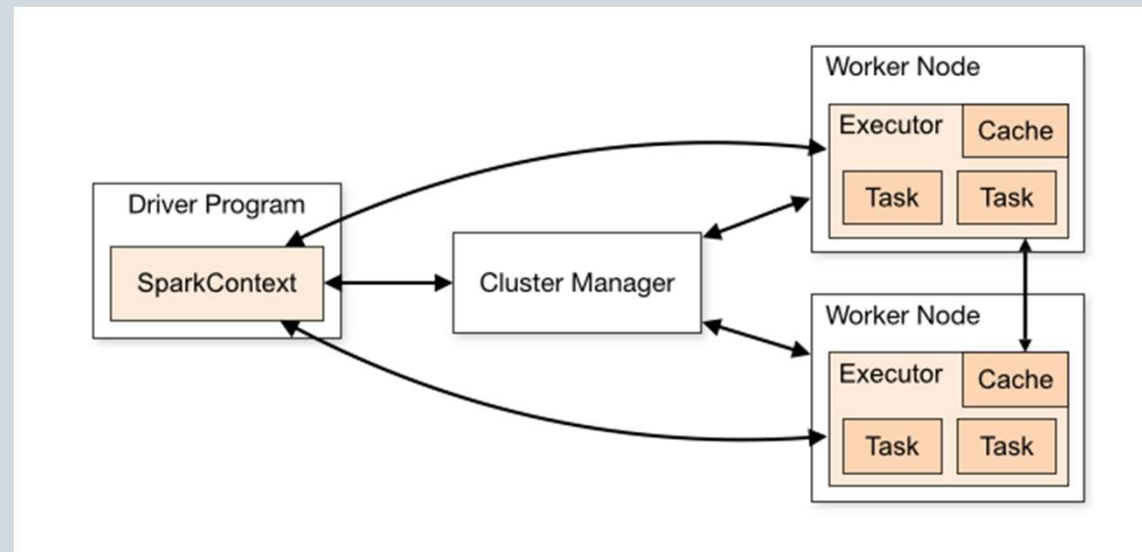
- **Driver Program:**
 - Responsible for coordinating the execution of the Spark application.
 - Run the main function
 - Creates the SparkContext, which connects to the cluster manager.
- **Cluster Manager:**
 - Responsible for allocating resources and managing the cluster on which the Spark application runs.
 - Supports various cluster managers like Apache Mesos, Hadoop YARN, and standalone cluster manager.



Spark Architecture

Main Components:

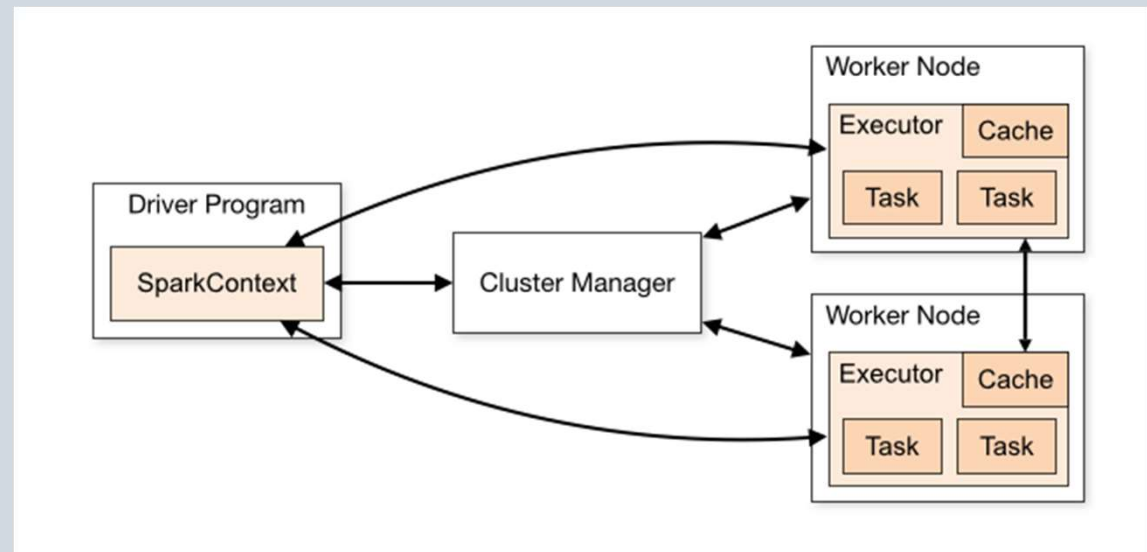
- **Executors:**
 - Responsible for executing tasks in Spark applications.
 - Launched on worker nodes and communicate with the driver program and cluster manager.
 - Run tasks concurrently and store data in memory or disk for caching and intermediate storage.
- **Spark Context:**
 - Entry point for any Spark functionality
 - Connection between Spark Cluster and the Spark Abstractions (RDDs, DAGs...)
 - Coordinate the execution of tasks



Spark Architecture

Main Components:

- **Tasks:**
 - Smallest unit of work in Spark.
 - Unit of computation that can be performed on a single partition of data.
 - Spark Jobs are divided by the Spark Driver into tasks and assigns them to the executor nodes for execution.
- **Execution Modes:**
 - Cluster mode
 - Client mode
 - Local mode



Tools and Libraries in Spark

- **Spark Core:** Base engine for distributed task execution.
- **Spark SQL:** Structured data processing.
- **Spark Streaming:** Real-time data processing.
- **MLlib:** Scalable machine learning library.
- **GraphX:** Graph computation framework.

Spark Data Abstraction

- A data abstraction is a logical data structure to the underlying data distributed on different nodes of the cluster
- It provides wide range of transformation methods (like **map()**, **reduce()**, **filter()**, etc)
- An action needs to be called to execute these transformations (like **show()**, **collect()**, **count()**, etc)
- The most important Spark Data Abstractions are:
 - RDDs
 - Datasets
 - **Dataframes**

RDD (Resilient Distributed Dataset)

Low-level, immutable distributed collections of objects.

Pros:

- Fine-grained control over data and transformations.
- Strongly typed, making it suitable for complex data manipulations.

Cons:

- Less optimized for performance compared to higher-level APIs.
- More verbose and requires manual optimization.

Use Cases: Complex transformations, custom data processing.

Spark Dataframes

Distributed collection of data organized into named columns, similar to a table in a relational database.

Pros:

- Optimized execution through Catalyst optimizer.
- Easier to use with SQL-like queries.
- Better performance due to optimized memory usage.

Cons:

- Less type-safe compared to Datasets.

Use Cases: ETL operations, SQL-based data analysis.

Spark Datasets

Combines the benefits of RDDs and DataFrames, providing type safety with the optimization benefits of DataFrames.

Pros:

- Type-safe, enabling compile-time checks.
- Optimized through Catalyst and Tungsten optimizers.
- Allows both object-oriented and functional programming

Cons:

- Slightly more complex API compared to DataFrames.

Use Cases: Applications requiring type safety and performance, complex data transformations.

Comparing Spark Data Abstractions

| Feature | RDD | DataFrame | DataSet |
|-----------------|-------------------------|------------------------------|--|
| Immutable | Yes | Yes | Yes |
| Type Safety | Yes | No | Yes |
| Optimization | Limited | High (Catalyst Optimizer) | High (Catalyst & Tungsten Optimizers) |
| Ease of Use | Low (more verbose) | High (SQL-like syntax) | Medium (combines RDD and DataFrame APIs) |
| Use Cases | Complex transformations | SQL-based data analysis, ETL | Type-safe, performance-critical applications |
| Fault tolerance | Yes | Yes | Yes |
| Level | Low | High | High |
| Schema | No | Yes | Yes |

Writing Queries in Spark

```
val df = spark.read.json("data.json")

// SELECT example
val selected = df.select("name", "age")

// GROUP BY example
val grouped = df.groupBy("city").count()

// JOIN example
val joined = df.join(otherDf, Seq("id"), "inner")

// UNION example
val unioned = df.union(otherDf)

// Adding new column with withColumn
val withNewCol = df.withColumn("is_adult", $"age" > 18)

// Broadcast example
val broadcasted = broadcast(df).join(otherDf, "id")

// Cache example
df.cache()
```

```
def joinNullSafe(dfLeft: DataFrame,
                 dfRight: DataFrame,
                 columnsToJoin: Array[String],
                 joinType: String = "inner") : DataFrame = {

    val usedSuffix: String = "_nullsafe"

    val dfRightRenamed : DataFrame =
        dfRight.select(dfRight.columns.map { columnName =>
            if (columnsToJoin.contains(columnName))
                col(columnName).as(s"$columnName$usedSuffix")
            else
                col(columnName)
        }: _*)

    val columnExpr : Column =
        dfLeft(columnsToJoin.head) <=> dfRightRenamed(s"${columnsToJoin.head}$usedSuffix")

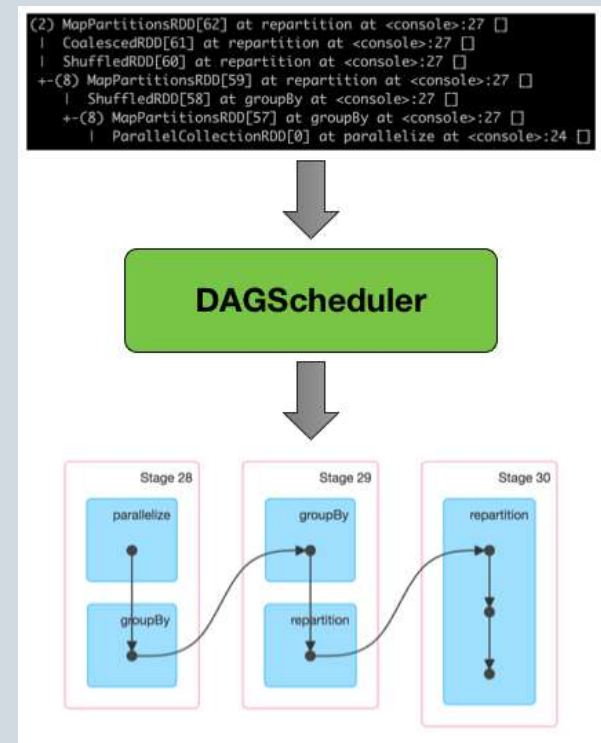
    val fullColumnExpr : Column = columnsToJoin.tail.foldLeft(columnExpr) {
        (columnExpr, columnName) =>
            columnExpr && dfLeft(columnName) <=> dfRightRenamed(s"$columnName$usedSuffix")
    }

    val dfJoined : DataFrame = dfLeft.join(dfRightRenamed, fullColumnExpr, joinType)
```

DAG - Directed Acyclic Graph

Based on the provided search results, a Directed Acyclic Graph (DAG) in Apache Spark represents the logical execution plan of a computation.

It's a series of stages (each containing tasks) that detail the sequence of transformations and operations applied to data.



Lazy Evaluation

Lazy evaluation is a key concept in Apache Spark that enables efficient processing of large-scale data.

It allows Spark to **delay the execution** of transformations until an action is triggered, reducing the computational overhead and memory usage.

Lazy evaluation provides several benefits:

- **Efficiency:** By delaying the execution of transformations, Spark can optimize the computation path and reduce the number of passes over the data.
- **Memory Savings:** Only the necessary data is processed, reducing memory consumption and avoiding unnecessary data loading.
- **Flexibility:** Lazy evaluation enables Spark to handle infinite data structures, as it only processes the data required for the requested result.

Spark Transforms vs. Actions

Transformation in Apache Spark refers to a function that produces a new Data Abstraction from an existing Data Abstraction, without modifying the original data.

Transformations are lazy, meaning they don't actually perform the computation until an **Action** is triggered. This allows Spark to optimize the execution plan and avoid unnecessary work.

Transformations

- Create new DataFrames or RDDs from existing ones
- Produce intermediate results, which are stored in memory or disk
- Are lazy operations, meaning they don't execute immediately
- Examples: map, filter, groupBy, pivot, join
- Transformations do not write data to external storage; instead, they create new in-memory representations of the data

Spark Transforms vs. Actions

Actions

- Trigger the execution of transformations, materializing the intermediate results
- Produce final output, which can be written to external storage (e.g., files, databases)
- Examples: count, collect, save, write
- Actions are responsible for evaluating the transformations and producing the final result

To illustrate the difference, consider a simple example:

- `df.filter(df.column > 5)` is a transformation, creating a new DataFrame with filtered rows.
- `df.filter(df.column > 5).count()` is an action, triggering the execution of the filter transformation and producing a count of the filtered rows.

In summary, transformations create intermediate results, while actions materialize these results and produce the final output. Understanding the distinction between transformations and actions is crucial for efficient and effective data processing in Spark DataFrames.

Catalyst Optimizer

The Catalyst Optimizer is a **core component of Apache Spark's SQL engine, responsible for optimizing the execution of Spark SQL queries**. It leverages advanced programming language features, such as Scala's pattern matching, to build an extensible query optimizer.

Key Features

- **Rule-based optimization:** Catalyst Optimizer applies a set of predefined rules to manipulate the query plan, enabling optimizations such as predicate pushdown, projection pruning, and filter pushing.
- **Cost-based optimization:** The optimizer uses a cost model to select the most efficient physical plan from multiple options.
- **Extensibility:** Developers can extend the optimizer by adding data source-specific rules, supporting new data types, and integrating with external systems.
- **Phase-based optimization:** The optimizer consists of several phases, including analysis, logical optimization, physical planning, and code generation, each addressing specific aspects of the query execution.

Catalyst Optimizer

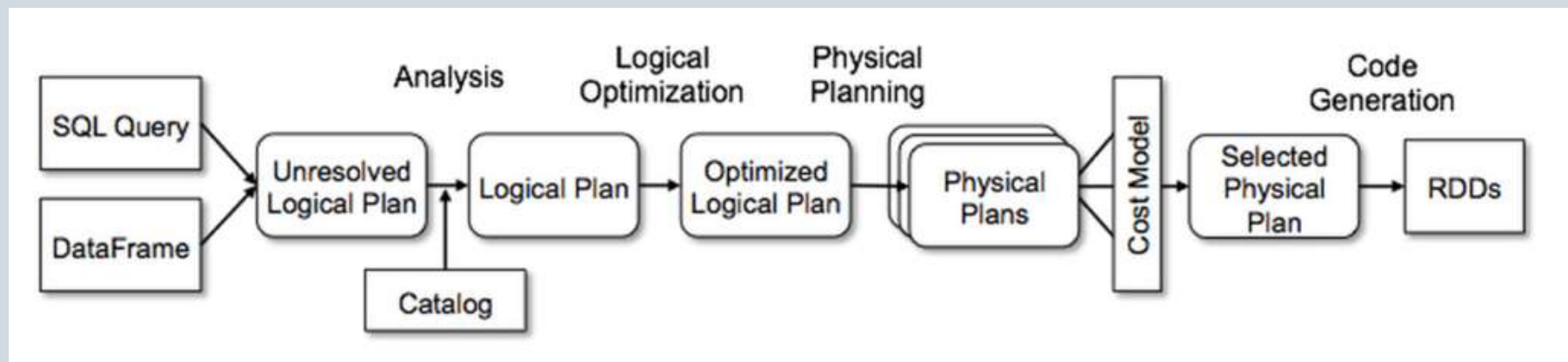
Phases of Optimization

- Analysis: Resolves types for all columns and identifies the query's logical structure.
- Logical Optimization: Applies rule-based optimizations, such as predicate pushdown and projection pruning, to the logical plan.
- Physical Planning: Generates one or more physical plans using physical operators that match the Spark execution engine.
- Code Generation: Converts the optimized logical plan into bytecode that can be executed on each machine in the cluster.

Examples of Optimization Techniques

- Join Reordering: Catalyst determines the most efficient order to join tables based on their sizes, reducing data shuffling during join operations.
- Filter Pushdown: Pushes filters into data sources that support predicate pushdown, reducing the amount of data processed.
- Projection Pruning: Eliminates columns not needed in the query result, reducing memory usage and I/O operations.

Catalyst Optimizer



Spark Shuffle Mechanism

Shuffling is a fundamental concept in Apache Spark, a distributed data processing framework.

It refers to the process of redistributing or reorganizing data across the partitions of a distributed dataset.

Shuffling is necessary when certain operations, such as grouping, aggregating, or joining data, require data to be rearranged to facilitate efficient processing.

Shuffling occurs in Spark when:

- Grouping or aggregating data: Operations like `groupByKey`, `reduceByKey`, or `aggregateByKey` require data to be rearranged to group similar values together.
- Joining datasets: Shuffling is necessary when joining two datasets based on a common key, as it ensures that all values for a given key are co-located on the same partition.
- Data skew: When data is not evenly distributed across partitions, shuffling helps to redistribute the data to balance the load.

Spark Shuffle Mechanism

Shuffling can be a resource-intensive operation, affecting both time and network utilization. Transferring and reorganizing data across the network can significantly slow down processing, especially with large datasets.

To minimize the impact of shuffling:

- Use keys with a large range of values: This helps to distribute data more evenly across partitions.
- Avoid unnecessary actions: Remove actions like count, show, or collect that can trigger redundant shuffling.
- Cache or persist data: Use `cache()` or `persist()` to store intermediate results in memory or on disk, reducing the need for repeated shuffling.
- Configure Spark settings: Adjust settings like `spark.sql.adaptive.coalescePartitions.enabled` to optimize partitioning and shuffling.
- Use composite keys: Instead of using a single primary key, use composite keys to improve data distribution.

Starting in Spark

Set Up Spark:

- Install Spark locally or deploy it on a cluster (e.g., Hadoop, Kubernetes, or Standalone mode).
- Configure Spark to connect with your data sources (e.g., HDFS, S3, or local files).

Write Your First Spark Application:

- Use one of Spark's supported languages: Scala, Python, Java, or R.
- Start by creating a SparkSession to work with DataFrames or Datasets.

```
val spark = SparkSession.builder()  
  .appName("SparkApp")  
  .master("local[*]")  
  .getOrCreate()
```

Starting in Spark

```
package libs

import org.apache.spark.sql.SparkSession
import libs.LogLib.showInfo
import vars.DefaultVars

object StartLib {

  private def getSparkConfigVarsForLocalSession: Map[String, String] = {
    DefaultVars.commonSparkConfigVars
  }

  def buildSparkLocalSession(appName: String,
                             cors: Int = 1): SparkSession = {

    val sparkBuilder: SparkSession.Builder = SparkSession.builder()
      .master(s"local[$cors]")
      .appName(appName)

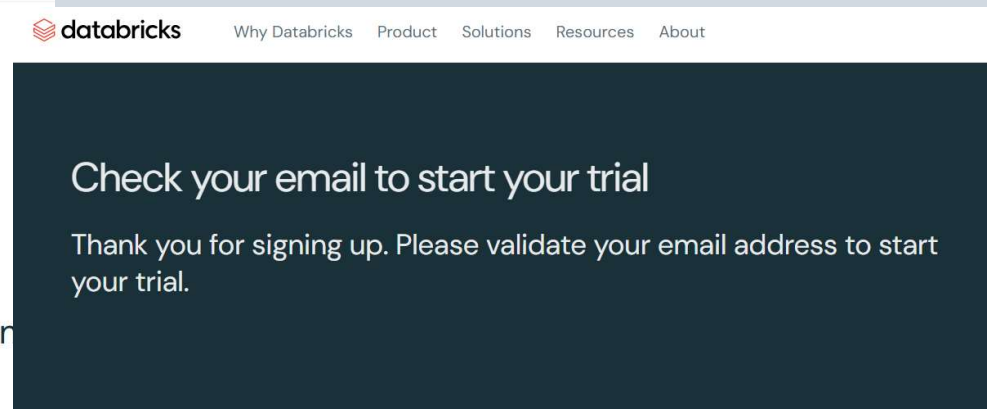
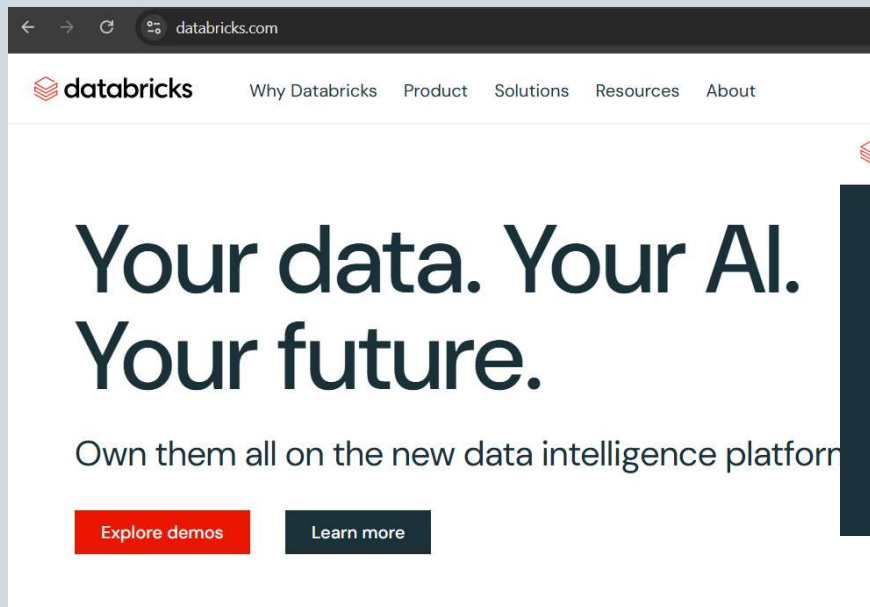
    for (configVar <- getSparkConfigVarsForLocalSession) {
      sparkBuilder.config(configVar._1, configVar._2)
    }

    val spark = sparkBuilder.getOrCreate()
    showInfo(s"Spark app ${spark.conf.get("spark.app.name")} started...")

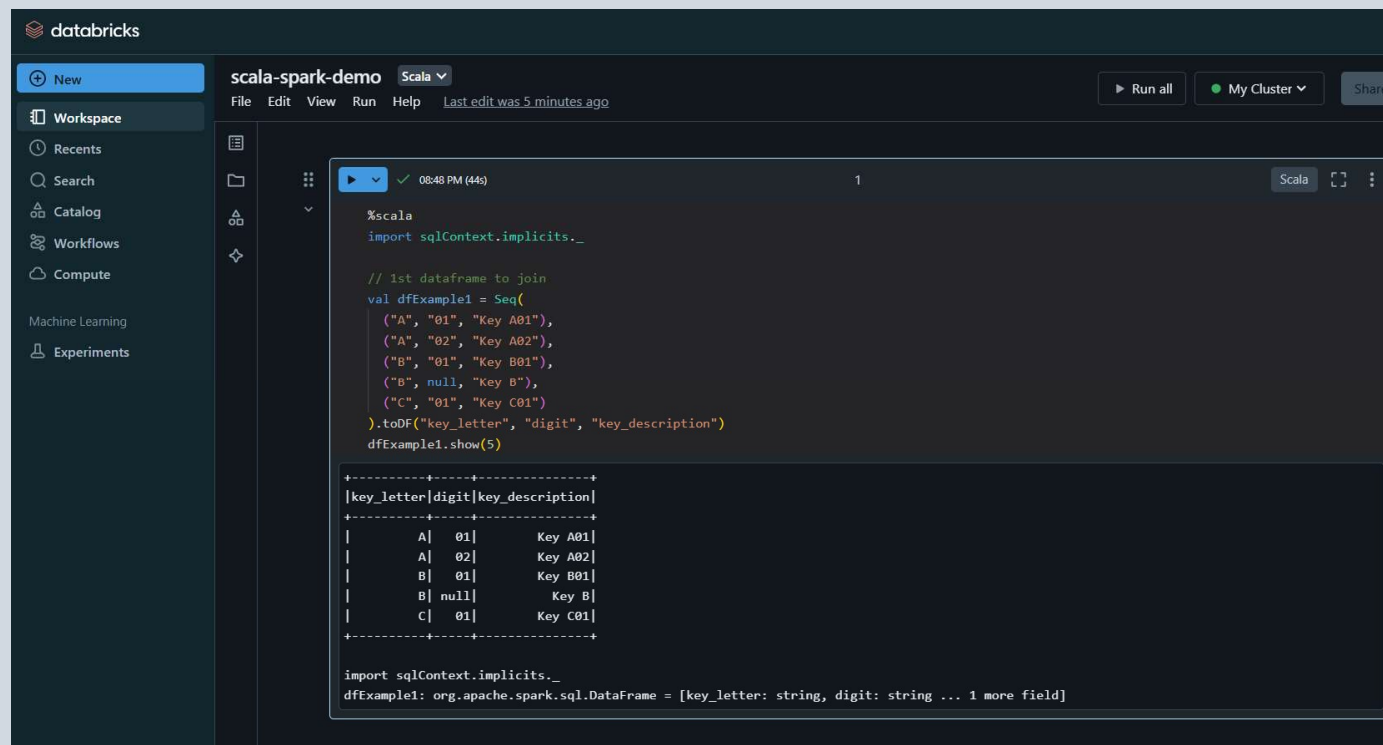
    spark
  }
}
```

Source: <https://github.com/robertoprjr/ss-lib/blob/main/src/main/scala/libs/StartLib.scala>

Starting in Spark: Databricks



Starting in Spark: Databricks



The screenshot shows the Databricks workspace interface. On the left is a sidebar with navigation options: New, Workspace, Recents, Search, Catalog, Workflows, Compute, Machine Learning, and Experiments. The main area displays a notebook titled "scala-spark-demo" in Scala. The notebook contains the following code:

```
%scala
import sqlContext.implicits._

// 1st dataframe to join
val dfExample1 = Seq(
  ("A", "01", "Key A01"),
  ("A", "02", "Key A02"),
  ("B", "01", "Key B01"),
  ("B", null, "Key B"),
  ("C", "01", "Key C01")
).toDF("key_letter", "digit", "key_description")
dfExample1.show(5)
```

The output of the code is displayed below the code block:

| key_letter | digit | key_description |
|------------|-------|-----------------|
| A | 01 | Key A01 |
| A | 02 | Key A02 |
| B | 01 | Key B01 |
| B | null | Key B |
| C | 01 | Key C01 |

Below the table, the output is also shown as a Scala DataFrame:

```
import sqlContext.implicits._
dfExample1: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 1 more field]
```

Databricks Notebook

Notebook with examples:

<https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/2344650012048761/1284899095684679/2657778362986166/latest.html>

Creating Dataframes

```
%scala
import sqlContext.implicits._

// 1st dataframe to join
val dfExample1 = Seq(
  ("A", "01", "Key A01"),
  ("A", "02", "Key A02"),
  ("B", "01", "Key B01"),
  ("B", null, "Key B"),
  ("C", "01", "Key C01")
).toDF("key_letter", "digit", "key_description")
dfExample1.show(5)
```

```
+-----+-----+-----+
|key_letter|digit|key_description|
+-----+-----+-----+
|      A|  01|      Key A01|
|      A|  02|      Key A02|
|      B|  01|      Key B01|
|      B| null|      Key B|
|      C|  01|      Key C01|
+-----+-----+-----+
```

```
import sqlContext.implicits._
dfExample1: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 1 more field]
```

Creating Dataframes

```
// 2nd dataframe to join
val dfExample2 = Seq(
  ("A", "01", 1, "Item A01.1"),
  ("A", "01", 2, "Item A01.2"),
  ("A", "02", 1, "Item A02.1"),
  ("B", "01", 1, "Item B01.1"),
  ("B", "01", 2, "Item B01.2"),
  ("B", null, 1, "Item B.1"),
  ("B", null, 2, "Item B.2"),
  ("D", "01", 1, "Item D01.1")
).toDF("key_letter", "digit", "item", "item_description")
dfExample2.show(5)
```

dfExample2: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 2 more fields]

```
+-----+-----+-----+
|key_letter|digit|item|item_description|
+-----+-----+-----+
|      A|  01|  1|    Item A01.1|
|      A|  01|  2|    Item A01.2|
|      A|  02|  1|    Item A02.1|
|      B|  01|  1|    Item B01.1|
|      B|  01|  2|    Item B01.2|
+-----+-----+-----+
```

only showing top 5 rows


dfExample2: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 2 more fields]

Join Example

```
// Columns defined for the join
val columnsToJoin = Array("key_letter", "digit")

// Example of the result without the null safe function
val dfJoined = dfExample1.join(dfExample2, columnsToJoin, "inner")
dfJoined.show(5)
```

► (4) Spark Jobs

►  dfJoined: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 3 more fields]

```
+-----+-----+-----+-----+
|key_letter|digit|key_description|item|item_description|
+-----+-----+-----+-----+
|      A|  01|      Key A01|  1|      Item A01.1|
|      A|  01|      Key A01|  2|      Item A01.2|
|      A|  02|      Key A02|  1|      Item A02.1|
|      B|  01|      Key B01|  1|      Item B01.1|
|      B|  01|      Key B01|  2|      Item B01.2|
+-----+-----+-----+-----+
```

columnsToJoin: Array[String] = Array(key_letter, digit)

dfJoined: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 3 more fields]

Select Example

```
▶ 09:15 PM (1s) 4
val dfSelected = dfJoined.select("key_letter", "digit", "item")
dfSelected: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 1 more field]

▶ 09:16 PM (2s) 5
dfSelected.show(10)
(4) Spark Jobs
+-----+-----+
|key_letter|digit|item|
+-----+-----+
|      A|  01|  1|
|      A|  01|  2|
|      A|  02|  1|
|      B|  01|  1|
|      B|  01|  2|
+-----+-----+
```

New Columns Example

```
▶ 09:35 PM (1s) 6

import org.apache.spark.sql.functions._

val newColumnValueKey = when(col("key_letter") === lit("A"), lit(100)).otherwise(lit(500))
val dfWithValueKey = dfJoined.withColumn("value_letter", newColumnValueKey)

▶ dfWithValueKey: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 4 more fields]

import org.apache.spark.sql.functions._
newColumnValueKey: org.apache.spark.sql.Column = CASE WHEN (key_letter = A) THEN 100 ELSE 500 END
dfWithValueKey: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 4 more fields]
```

```
▶ 09:35 PM (1s) 7

val newColumnValueItem = when(col("item") === lit("1"), lit(10)).otherwise(lit(50))
val dfWithValueItem = dfWithValueKey.withColumn("value_item", newColumnValueItem)

▶ dfWithValueItem: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 5 more fields]

newColumnValueItem: org.apache.spark.sql.Column = CASE WHEN (item = 1) THEN 10 ELSE 50 END
dfWithValueItem: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 5 more fields]
```

New Columns Example

```
▶ 09:35 PM (1s) 8
dfWithValueItem.show(10)
▶ (4) Spark Jobs
```

| key_letter | digit | key_description | item | item_description | value_letter | value_item |
|------------|-------|-----------------|------|------------------|--------------|------------|
| A | 01 | Key A01 | 1 | Item A01.1 | 100 | 10 |
| A | 01 | Key A01 | 2 | Item A01.2 | 100 | 50 |
| A | 02 | Key A02 | 1 | Item A02.1 | 100 | 10 |
| B | 01 | Key B01 | 1 | Item B01.1 | 500 | 10 |
| B | 01 | Key B01 | 2 | Item B01.2 | 500 | 50 |

```
▶ 09:38 PM (1s) 9
val newColumnValueFinal = col("value_letter") + col("value_item")
val dfWithValue = dfWithValueItem.withColumn("value_final", newColumnValueFinal)
dfWithValue: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 6 more fields]
newColumnValueFinal: org.apache.spark.sql.Column = (value_letter + value_item)
dfWithValue: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 6 more fields]
```

Group By Example

09:43 PM (4s)

10

```
val dfAgg = dfWithValue.groupBy("key_letter", "digit").agg(  
  sum("value_letter").as("sum_value_letter"),  
  sum("value_final").as("sum_value_final"),  
  count(lit(1)).as("cnt_lines")  
)
```

```
dfAgg.show(10)
```

▶ (3) Spark Jobs

▶ dfAgg: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 3 more fields]

| key_letter | digit | sum_value_letter | sum_value_final | cnt_lines |
|------------|-------|------------------|-----------------|-----------|
| A | 01 | 200 | 260 | 2 |
| A | 02 | 100 | 110 | 1 |
| B | 01 | 1000 | 1060 | 2 |

dfAgg: org.apache.spark.sql.DataFrame = [key_letter: string, digit: string ... 3 more fields]

Filtering Example

```
09:49 PM (1s) 11 Scala
val dfFilteredA = dfWithValue.filter(col("key_letter") === lit("A") && col("digit") === lit("01"))
dfFilteredA.show(10)
```

▶ (2) Spark Jobs

▶ dfFilteredA: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [key_letter: string, digit: string ... 6 more fields]

| key_letter | digit | key_description | item | item_description | value_letter | value_item | value_final |
|------------|-------|-----------------|------|------------------|--------------|------------|-------------|
| A | 01 | Key A01 | 1 | Item A01.1 | 100 | 10 | 110 |
| A | 01 | Key A01 | 2 | Item A01.2 | 100 | 50 | 150 |

dfFilteredA: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [key_letter: string, digit: string ... 6 more fields]

```
09:50 PM (1s) 12 Scala
val dfFilteredB = dfWithValue.filter(col("key_letter") === lit("B"))
dfFilteredB.show(10)
```

▶ (2) Spark Jobs

▶ dfFilteredB: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [key_letter: string, digit: string ... 6 more fields]

| key_letter | digit | key_description | item | item_description | value_letter | value_item | value_final |
|------------|-------|-----------------|------|------------------|--------------|------------|-------------|
| B | 01 | Key B01 | 1 | Item B01.1 | 500 | 10 | 510 |
| B | 01 | Key B01 | 2 | Item B01.2 | 500 | 50 | 550 |

dfFilteredB: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [key_letter: string, digit: string ... 6 more fields]

Union Example

▶ ✓ 09:51 PM (2s)

13

```
val dfUnion = dfFilteredA.union(dfFilteredB).show(10)
```

▶ (4) Spark Jobs

| key_letter | digit | key_description | item | item_description | value_letter | value_item | value_final |
|------------|-------|-----------------|------|------------------|--------------|------------|-------------|
| A | 01 | Key A01 | 1 | Item A01.1 | 100 | 10 | 110 |
| A | 01 | Key A01 | 2 | Item A01.2 | 100 | 50 | 150 |
| B | 01 | Key B01 | 1 | Item B01.1 | 500 | 10 | 510 |
| B | 01 | Key B01 | 2 | Item B01.2 | 500 | 50 | 550 |

```
dfUnion: Unit = ()
```

Spark 3.x New Concepts

Dynamic Partition Pruning (DPP): Introduced in Spark 3.0, DPP is a performance improvement for SQL analytics workloads. It enables the optimizer to prune partitions dynamically during query execution, reducing the amount of data being processed and improving query performance.

Adaptive Query Execution: Spark 3.x introduces adaptive query execution, which allows the optimizer to adjust query plans based on runtime statistics. This feature helps optimize query performance and reduces the need for manual tuning.

ANSI SQL Compliance: Spark 3.x improves ANSI SQL compliance, making it easier to integrate with BI tools and other SQL-based systems. This feature provides better support for standard SQL features and reduces the need for custom SQL syntax.

Pandas API Redesign: The Pandas API has been redesigned in Spark 3.x, with improved type hints, new UDF types, and more Pythonic error handling. This redesign aims to make Pandas-based data processing more efficient and easier to use.

Spark 3.x New Concepts

Trigger.AvailableNow: Introduced in Spark 3.3.0, Trigger.AvailableNow allows running streaming queries like Trigger.Once in multiple batches, improving the flexibility and scalability of streaming workloads.

DS V2 Push Down Capabilities: Spark 3.x introduces more comprehensive DS V2 push down capabilities, enabling more efficient data processing and reducing the need for data movement.

Logical Plan Visitor: A new logical plan visitor has been added to propagate distinct attributes, improving the accuracy and efficiency of query optimization.

Accelerator-Aware Scheduler: Spark 3.x includes an accelerator-aware scheduler, which optimizes query execution by taking into account the availability and performance characteristics of various accelerators (e.g., GPUs, TPUs).

Why Apache Spark?

Scalable and Fast: Apache Spark processes large-scale data efficiently with in-memory computation and parallel processing across distributed systems.

Versatile: It supports diverse workloads, including batch processing, real-time streaming, machine learning, and graph analytics, all within a unified framework.

Future-Ready: With advanced optimizations like Catalyst, Adaptive Query Execution, and integrations with modern hardware and cloud platforms, Spark is designed for the evolving data landscape.

Simple. Fast. Scalable. Unified.

Questions?

Challenge

- Create an account in the Databricks
- Use the example notebook and create your own data transformations
- Get some inspirations in the Project used as an example
 - <https://github.com/robertoprjr/ss-lib/tree/main>

Contact

Roberto Passos Rodrigues Jr.

Developer / Data Engineer / Tech Lead at Natixis

Email: robertoprjr@gmail.com

GitHub: <https://github.com/robertoprjr>

LinkedIn: <https://www.linkedin.com/in/roberto-passos-rodrigues-jr/>