Introduction to Apache Spark using Python PySpark

1. What is Apache Spark?

Apache Spark is an open-source, distributed computing system that processes large datasets quickly across clusters of computers.

It's widely used in data analytics, big data processing, and machine learning due to its speed, ease of use, and versatility.

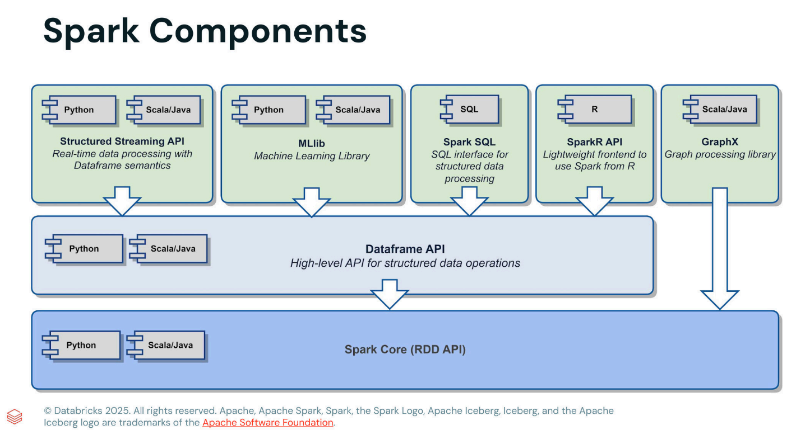
It can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and many more.

Key Features of Apache Spark:

- **Speed:** Spark can process data up to 100 times faster than traditional data-processing frameworks.
- **Distributed Computing:** It divides large datasets across multiple machines, enabling parallel processing.
- **Ease of Use:** Spark provides APIs in Python, Scala, Java, and R, making it accessible to various developers.
- **Unified Data Processing Engine:** Spark has multiple components for different tasks, including Spark SQL, MLlib for machine learning, and Spark Streaming for real-time data.

Spark Architecture and Components:

The Spark Core Engine forms the foundation for all Spark applications. It manages memory, fault recovery, scheduling, and task distribution. High-level APIs such as the DataFrame API, RDD API, and SQL API provide access to the core engine, and they are available in multiple languages including Python, Scala, Java, and R. On top of the core, specialized libraries like Spark SQL, MLlib, GraphX, and Structured Streaming extend Spark's capabilities, offering advanced functionalities for diverse use cases.



1. **Spark Core**

- RDDs (Resilient Distributed Datasets): The fundamental data structure in Spark, RDDs are immutable, fault-tolerant collections of objects distributed across nodes. They support operations like map, filter, and reduce.
- Transformations and Actions: Transformations (e.g., map, filter) create a new RDD, while actions (e.g., count, collect) execute computations and return results.

2. **Spark SQL**

- Spark SQL allows querying structured data using SQL or the DataFrame API. It's optimized for performance and commonly used for analyzing large datasets.

3. **MLlib** (Machine Learning Library)

- MLlib is Spark's library for scalable machine learning algorithms, including regression, classification, clustering, and collaborative filtering.

4. **Spark Streaming**

- Enables real-time data processing for applications needing continuous, live data feeds.

5. **GraphX**

- A graph processing library in Spark for graph-parallel computations, useful for tasks like social network analysis.

Spark Time Line

Origins of Apache Spark
Apache Spark started in 2009 at UC Berkeley's AMPLab as a research project. In 2010, it was made open source, allowing developers worldwide to contribute and adopt

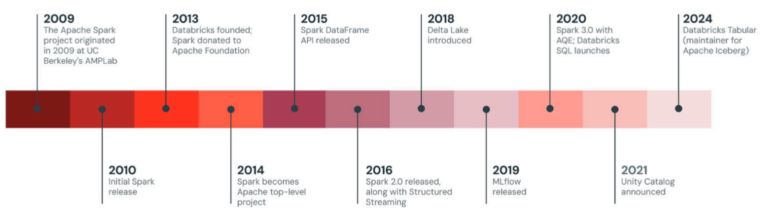
Databricks and Apache Spark
In 2013, the original creators of Spark founded Databricks. The Spark project was contributed to the Apache Software Foundation, and by 2014, Spark became a top-level Apache project.

Key Contributions Over the Years
Databricks continued driving Spark's innovation with major milestones:

- 2015 – Introduction of the DataFrame API
- 2016 – Structured Streaming for real-time processing
- 2018 – Delta Lake for reliable data lakes
- 2019 – MLflow for managing machine learning workflows

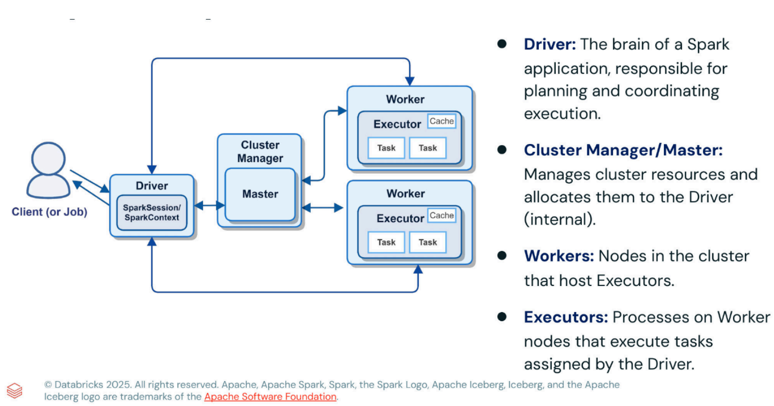
Spark Today
Today, Databricks remains a primary contributor, pushing Spark forward with:

- Adaptive Query Execution (AQE)
- Unity Catalog for data governance
- Many more enhancements supporting large-scale data and AI workloads



Spark Architecture

- The Driver is the core component responsible for managing the application's lifecycle and execution.
- The Cluster Manager (Master) allocates resources and assigns tasks to worker nodes; in Databricks clusters, this process is handled automatically.
- Workers carry out the tasks assigned by the driver, executing code and returning results. Executors run on worker nodes, managing task execution and caching data for performance.



The Spark Driver

Planning, Coordination, and Task Execution Management

- Creates the SparkSession, the entry point for all Spark applications
- Analyzes the Spark application and builds a Directed Acyclic Graph (DAG)
- Schedules and distributes tasks to Executors for execution
- Monitors task progress and manages failures
- Returns results to the client

SparkSession

Spark Application Entry Point

- Introduced in Spark 2.0 as a unified entry point for all contexts
- (previously separate: SparkContext, SQLContext, HiveContext, StreamingContext)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("MySparkApplication") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Executors

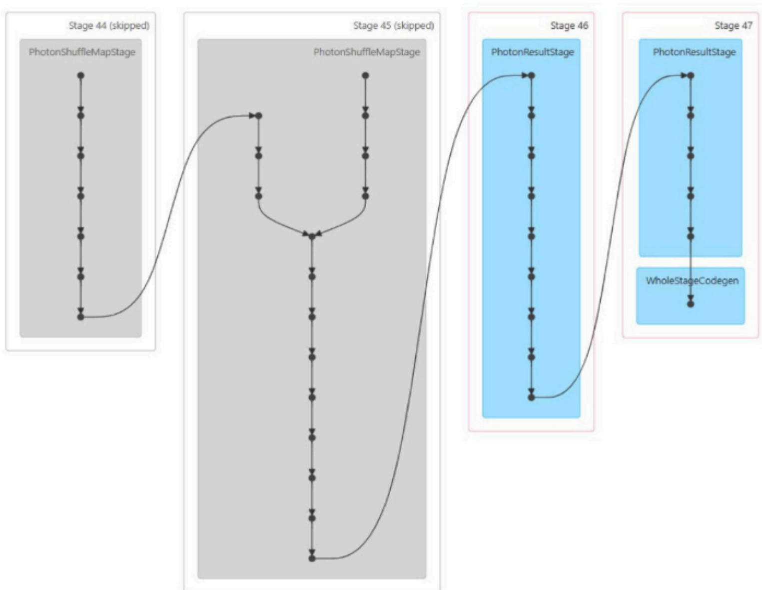
Task Execution and I/O

- Run on worker nodes in a Spark cluster
- Each worker node can host multiple Executors, depending on:
    - Available CPU cores (spark.executor.cores)
    - Memory resources (spark.executor.memory)
    - Configuration settings
- Handle task execution and data processing
- Store intermediate and final results (in memory or on disk)
- Communicate with the Driver for task coordination and data transfer

The Spark DAG

Directed Acyclic Graph

- Spark jobs are broken down into stages (groups of tasks that can run in parallel)
- Computations flow in one direction through stages (Directed)
- Stages never loop back, ensuring jobs terminate (Acyclic)
- Stages are organized into a dependency graph for execution flow (Graph)



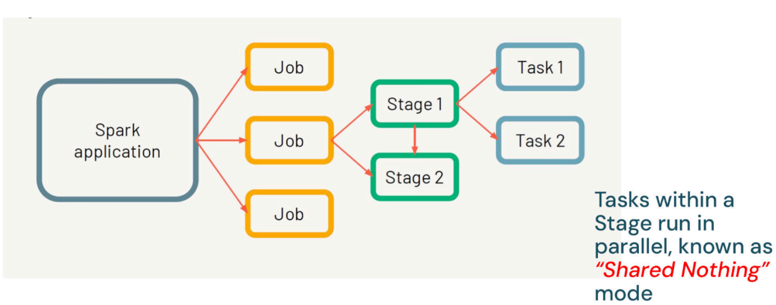
The diagram illustrates how a **Spark job** is executed internally:

1. **Stages in Execution**
   - The rectangles represent **stages**, each made up of multiple tasks.
   - These stages are created based on shuffle boundaries (when data needs to be moved across nodes).
2. **Directed Flow**
   - The arrows show that computations move **forward only** from one stage to the next (hence *Directed*).
   - There are no backward arrows, which means Spark jobs can't loop infinitely.
3. **Acyclic Nature**
   - The stages form a chain without cycles (hence *Acyclic*).
   - Once a stage finishes, execution moves to the next stage until the job completes.
4. **Dependency Graph**
   - The curved arrows represent **dependencies** between stages.
   - For example, the output of Stage 1 becomes input to Stage 2.

In short, the image **visualizes how Spark breaks down a job into stages and tasks, executes them in order, and ensures progress moves forward without looping**.

Spark Application

Jobs in Spark are high-level operations triggered by actions such as collect() or save().
Each job is divided into stages, which are smaller, independent units that can run in parallel.
Within stages, the work is further broken down into tasks, the smallest units of execution handled by Executors.
During parallel execution, tasks within a stage operate independently, following a "Shared Nothing" architecture, meaning no task shares memory or data with another.



The Spark UI

Visualising Spark Applications

Spark provides web interfaces for monitoring and management, including:

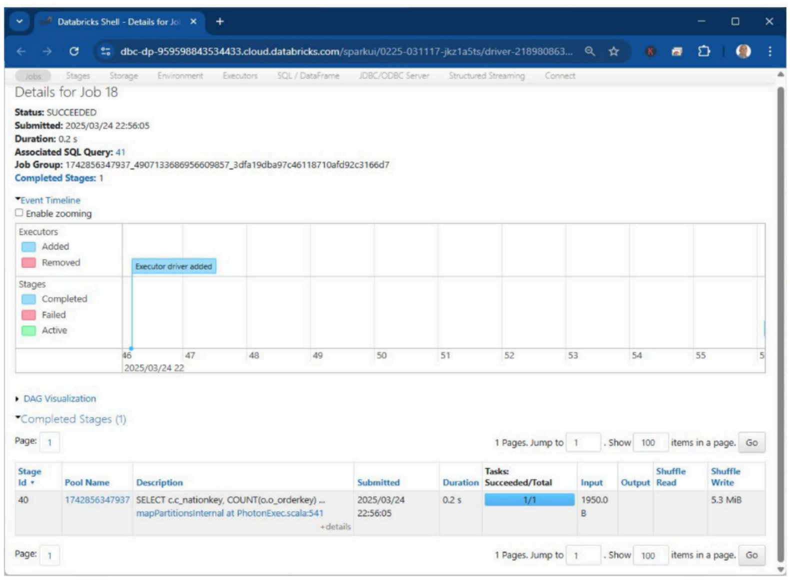
Application UI – per application (SparkSession)

- focuses on a single Spark job/session
- Tracks application progress and task execution
- Displays DAG visualization and stage details
- Shows resource usage and performance metrics

Master UI – per cluster

- focuses on cluster-wide monitoring
- Displays worker node status, health, and resource allocation
- Shows all running applications and available resources

Application UI especially useful for **debugging, troubleshooting, and performance optimization**, since it visualizes DAGs, bottlenecks, and system health.



Introduction to Databricks

Databricks is a cloud-native data and AI platform designed to unify the workflows of data engineering, data science, machine learning, and business analytics. Founded in 2013 by the creators of Apache Spark, Databricks builds on Spark's distributed processing capabilities while providing a fully managed, collaborative, and scalable environment for working with data at scale. Its goal is to remove the operational complexity of big data and AI systems so that organizations can focus on generating insights and building innovative applications.

- Databricks is a cloud-based data and AI platform designed to unify data engineering, data science, and machine learning workflows.
- It was founded in 2013 by the original creators of Apache Spark and has since become one of the leading platforms for large-scale data processing and analytics.
- Built on top of Apache Spark, Databricks simplifies the process of working with big data by providing a collaborative workspace that integrates with popular cloud providers such as AWS, Azure, and Google Cloud.
- The platform supports data ingestion, transformation, and advanced analytics through features like Delta Lake for reliable data lakes, MLflow for machine learning lifecycle management, and the Unity Catalog for data governance. Databricks provides a serverless and scalable environment where teams can run notebooks, SQL queries, streaming jobs, and machine learning models seamlessly.
- It is widely used by enterprises for building modern data architectures, enabling real-time analytics, and accelerating AI development.

Core Architecture

The Databricks platform follows a **multi-layered architecture** that abstracts infrastructure while providing flexible compute and storage integration.

1. **Workspace** A workspace is the primary environment where users interact with Databricks. It provides collaborative tools such as notebooks, dashboards, and SQL editors. Multiple users and teams can work in the same workspace while maintaining controlled access to resources.
2. **Clusters** Clusters are groups of virtual machines managed by Databricks. They provide the compute resources necessary to run Spark applications, notebooks, jobs, and ML workflows. Clusters can be created on demand, auto-scaled, and terminated automatically, reducing operational overhead and costs.
3. **Jobs** Jobs are scheduled workloads that run on clusters. They allow organizations to automate data ingestion, ETL pipelines, model training, and reporting. Jobs can be triggered manually, on a schedule, or based on events.
4. **Data Management Layer** Databricks integrates with cloud object storage (e.g., AWS S3, Azure Data Lake, Google Cloud Storage) while adding features such as **Delta Lake**. Delta Lake ensures data reliability by bringing ACID transactions, schema enforcement, and time travel to traditional data lakes, effectively turning them into robust "lakehouses."
5. **Libraries and APIs** Databricks supports multiple programming languages, including Python, Scala, Java, R, and SQL. Users can leverage Spark APIs, SQL interfaces, and machine learning frameworks such as TensorFlow, PyTorch, and Scikit-learn within the same environment.

Ecosystem Components

Databricks extends beyond Spark with a set of specialized components:

- **Delta Lake** Provides reliable data storage and management with ACID transactions, enabling the Lakehouse architecture.
- **MLflow** An open-source framework for managing the machine learning lifecycle, including experiment tracking, model packaging, and deployment.
- **Unity Catalog** A unified governance solution that centralizes data access policies, lineage tracking, and auditing across the platform.
- **Structured Streaming** Enables real-time data processing and analytics, allowing organizations to build streaming applications for event-driven workloads.
- **Databricks SQL** Provides a SQL-native interface for business analysts, integrating dashboards and visualization tools for data exploration.

Collaboration and Productivity

One of Databricks' distinguishing features is its **collaborative notebook interface**, which allows multiple users to contribute to the same notebook simultaneously. This feature fosters cross-functional teamwork, enabling data engineers, scientists, and analysts to work together seamlessly. Built-in version control, commenting, and integration with Git further enhance productivity.

Integration with Cloud Providers

Databricks runs on top of major cloud platforms, including **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)**. This ensures elasticity, high availability, and scalability while allowing organizations to leverage existing cloud infrastructure for storage, networking, and identity management.

Use Cases

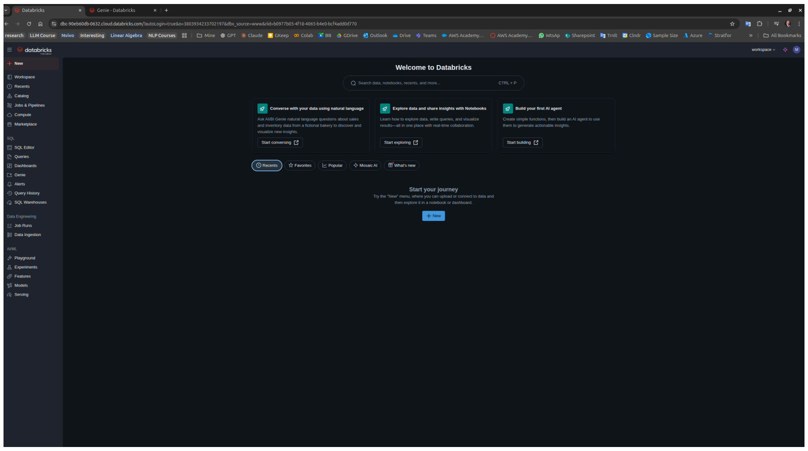
Databricks supports a wide variety of data and AI workloads, including:

- **Data Engineering**: ETL pipelines, batch and streaming data processing.
- **Data Science and AI**: Experimentation, training, and deployment of machine learning models.
- **Business Intelligence**: SQL-based data exploration, visualization, and reporting.
- **Real-Time Analytics**: Processing of streaming data for fraud detection, monitoring, and recommendation systems.

Conclusion

Databricks represents a comprehensive solution for modern data-driven organizations. By unifying disparate data and AI workflows, it reduces complexity and accelerates the delivery of insights. With components such as Delta Lake, MLflow, and Unity Catalog, and with seamless integration across cloud providers, Databricks provides a powerful and flexible environment for building scalable, secure, and collaborative data solutions.

Databricks Free Acount



What options exist for "free" Databricks access

- **Databricks Free Edition** (replacing Community Edition) — good for students, hobbyists, learning / prototyping.)
- **Databricks Free Trial** — gives you temporary credits and access to more features. Valid for 14 days.

Steps to create a free Databricks account / workspace

Here are the general steps. Depending on your region or cloud provider, some details might differ.

1. Go to the Databricks website and find **Databricks Free Edition** or "Try Databricks for Free".
2. Click **Sign up** for Free Edition, or start the Free Trial.
3. Choose your signup method:
   - You can often sign up just with an email ("Express signup").
   - Or use your existing cloud provider account (e.g. AWS) if you want to link storage / compute resources.
4. Fill in registration details (name, email, password, etc.). If needed, verify your email.
5. (If Free Trial) You might need to enter payment method or link a cloud account so that after the trial expires, there is a way to transition if you decide to continue.
6. A workspace is created for you. In Free Edition, it's serverless and quota-limited.
7. Once you're inside the workspace, you can:
   - Create notebooks
   - Run code
   - Explore tools like SQL editor, dashboards
   - Work with data stored or uploaded into the workspace.

Limitations / things to watch out for

- Free Edition has **quotas** and **limitations** (e.g. on compute, storage, features).
- Free Trial gives you more access, but only for ~14 days. After that, if you don't upgrade, some services might become unavailable.
- In some signup flows, especially via cloud providers, you may incur charges for cloud resources (storage, VM use) if those are outside what the credit allows. Always check what is free vs what might cost.

---

2. Getting Started with PySpark

1. Setting Up PySpark You can run PySpark on a local machine or on a distributed cluster. For classroom purposes, we'll focus on setting up PySpark locally with a Jupyter Notebook environment.

Installation Steps:

1. Install Java (required for Spark):
   default-jre

2. Install PySpark:
   pip install pyspark
3. Running Your First PySpark Program
   You can check if PySpark is installed correctly by launching a Jupyter Notebook and running this code to initialize Spark:

Creating a spark session

SparkSession is the entry point to programming with Spark. It allows you to interact with Spark, load and process data, and manage resources in a cluster.

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("Introduction to Spark").getOrCreate()

# Display Spark version
print("Spark version:", spark.version)
```

```
Spark version: 3.5.3
```

```
# The command above is equivalent to this command which explicitly uses all the availabe cores:
# SparkSession.builder.master("local[*]").appName("...").getOrCreate()
# SparkSession.builder.master("local[2]").appName("Limited Spark").getOrCreate() // limits to 2 cores
```

Check the number of available cores

```
import os
os.cpu_count()

```

- SparkSession.builder: This starts the construction of a Spark session.
- .appName("Introduction to Spark"): This sets the name of your Spark application. It is used for identification in Spark's UI and logs.
- .getOrCreate(): This either retrieves the existing Spark session (if one already exists) or creates a new one if none exists. This creates a Spark session object named spark, which you will use to interact with Spark.

This initializes Spark and shows the version number, confirming that your environment is ready.

Resilient Distributed Datasets (RDDs)

What is an RDD?

RDDs, or Resilient Distributed Datasets, are the foundational data structure in Spark. They represent an immutable distributed collection of objects that can be processed in parallel across the nodes in a Spark cluster. Key properties of RDDs:

- Resilient: RDDs automatically recover from node failures.
- Distributed: Data is spread across multiple nodes.
- Immutable: Once created, an RDD cannot be changed.

Creating RDDs

There are two main ways to create RDDs:

- Parallelizing a collection: Creating an RDD from an existing list or array.
- Reading from an external data source: Creating an RDD from a file or dataset (like a CSV file).

Examples:

1. Parallelize a Collection:

```
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)

```

```
# Collect the RDD data and print it
print(rdd.collect())
```

```
[1, 2, 3, 4, 5]
```

2. Read from a Text File:

```
rdd = spark.sparkContext.textFile("path/to/file.txt")
```

---

3. Tansformations and Actions

In Apache Spark, **Transformations** and **Actions** are the two main types of operations used to process and analyze data. Understanding the difference between them is crucial for mastering Spark.

Transformations are functions executed on demand to produce a new RDD. All transformations are followed by actions. Some examples of transformations include map, filter, and reduceByKey.

Actions are the results of RDD computations or transformations. After an action is performed, the data from RDD moves back to the local machine. Some examples of actions include reduce, collect, first, and take.

3.1. Transformations

Transformations are **lazy operations** that define a set of instructions for manipulating data but do not execute them immediately. Instead, they create a new Resilient Distributed Dataset (RDD) or DataFrame, representing a logical plan for execution.

Key Characteristics

- **Lazy Evaluation**: Spark doesn't execute transformations until an action triggers the computation.
- **Immutable Data**: Transformations create new RDDs or DataFrames rather than modifying existing ones.
- **Chaining**: Multiple transformations can be chained together to create complex workflows.

Common Transformations

Transformation	Description	Example
`map()`	Applies a function to each element in the dataset.	Transform each number to its square.
`filter()`	Filters elements based on a condition.	Keep only even numbers.
`flatMap()`	Similar to `map()`, but can produce multiple output elements for each input element.	Split lines of text into words.
`distinct()`	Removes duplicate elements.	Get unique values in a dataset.
`union()`	Combines two datasets into one.	Combine two RDDs.
`groupByKey()`	Groups data by key (key-value RDD).	Group all values by their keys.
`join()`	Performs a join operation on two datasets.	Join two RDDs/DataFrames.

1. map(): Applies a function to each element in the RDD and returns a new RDD.

```
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)

# Square each number
squared_rdd = rdd.map(lambda x: x**2)

print(squared_rdd.collect())  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

Parallelization: The `spark.sparkContext.parallelize(data)` function distributes the data across multiple cores or nodes in the cluster. Spark divides the dataset into partitions, and each partition can be processed independently.

Transformations and Actions:

- The `filter` and `map` operations are **transformations**. They are lazy and only define the computation to be performed.
- The `collect` operation is an **action**. It triggers execution (map in this case) and aggregates the results back to the driver (local machine) and returns them as a Python list.

Scaling:

- If you run this code in a Spark cluster with multiple cores or nodes, Spark will distribute the transformations (`filter` and `map`) across all available resources.

- Each core or executor will work on a subset of the data, speeding up the computation.

2. filter(): Returns a new RDD containing only the elements that satisfy a given condition.

```
even_rdd = rdd.filter(lambda x: x % 2 == 0)
print(even_rdd.collect())
```

```
[2, 4]
```

3. flatMap(): Similar to map(), but flattens the results. map() transformation is applied to each row in a dataset to return a new dataset. flatMap() transformation is also used for each dataset row, but a new flattened dataset is returned. In the case of flatMap, if a record is nested (e.g., a column that is in itself made up of a list or array), the data within that record gets extracted and is returned as a new row of the returned dataset.

Both map() and flatMap() transformations are narrow, meaning they do not result in the shuffling of data in Spark.

- flatMap() is a one-to-many transformation function that returns more rows than the current DataFrame. Map() returns the same number of records as in the input DataFrame.
- flatMap() can give a result that contains redundant data in some columns.
- flatMap() can flatten a column that contains arrays or lists. It can be used to flatten any other nested collection too.

```
lines = spark.sparkContext.parallelize(["hello world", "how are you"])
words = lines.flatMap(lambda line: line.split(" "))
print(words.collect())
```

```
['hello', 'world', 'how', 'are', 'you']
```

4. distinct(): Removes duplicate elements.

```
rdd_with_duplicates = spark.sparkContext.parallelize([1, 2, 2, 3, 4])
distinct_rdd = rdd_with_duplicates.distinct()
print(distinct_rdd.collect())
```

```
[1, 2, 3, 4]
```

5. union(): Combines two RDDs into one.

```
rdd1 = spark.sparkContext.parallelize([1, 2, 3])
rdd2 = spark.sparkContext.parallelize([4, 5, 6])
combined_rdd = rdd1.union(rdd2)
print(combined_rdd.collect())
```

```
[1, 2, 3, 4, 5, 6]
```

Excercise

Write Python code to do the following:

- Create an Array that has 10 million numbers.
- Convert that array into an RDD using PySpark
- Write code to compute the average of the numbers in the RDD using PySpark

3.2. Actions

Actions are **eager operations** that trigger the execution of transformations. They perform computations and return a result to the driver program or write the output to an external storage.

Key Characteristics

- **Trigger Execution**: Actions force Spark to evaluate the transformations and perform the computation.
- **Return Results**: They either return a value to the driver or save the result to a file system.
- **Irreversible**: Actions mark the end of a computation chain.

Common Actions

Action	Description	Example
`collect()`	Returns all elements of the dataset as a list.	Collect results from an RDD.
`count()`	Counts the number of elements in the dataset.	Find the total number of rows.
`first()`	Returns the first element of the dataset.	Get the first line in a file.
`take(n)`	Returns the first `n` elements of the dataset.	Get the first 5 rows.
`reduce()`	Aggregates data using a specified function.	Find the sum of all numbers.
`saveAsTextFile()`	Saves the dataset to a text file.	Save results to HDFS or local storage.
`show()`	Displays the first few rows of a DataFrame.	Show data in tabular format.

Some common actions include: 1. collect(): Returns all elements of the RDD as a list (use sparingly with large datasets).

```
print("Collected elements:", rdd.collect())
```

```
Collected elements: [1, 2, 3, 4, 5]
```

2. count(): Counts the number of elements in the RDD.

```
print("Count of elements:", rdd.count())
```

```
Count of elements: 5
```

3. first(): Returns the first element in the RDD.

```
print("First element:", rdd.first())
```

```
First element: 1
```

4. take(n): Returns the first n elements.

```
print("First three elements:", rdd.take(3))
```

```
First three elements: [1, 2, 3]
```

5. reduce(): Aggregates the elements of the RDD using a specified function.

```
# Sum all elements
sum_of_elements = rdd.reduce(lambda x, y: x + y)
print("Sum of elements:", sum_of_elements)
```

```
Sum of elements: 15
```

6. countByValue(): Returns a dictionary of each unique value and its count.

```
value_counts = rdd.countByValue()
print("Value counts:", value_counts)
```

```
Value counts: defaultdict(<class 'int'>, {1: 1, 2: 1, 3: 1, 4: 1, 5: 1})
```

Transformations vs. Actions

Feature	Transformations	Actions
Execution	Lazy: Build a logical execution plan.	Eager: Trigger computation.
Output	Produces a new RDD/DataFrame.	Returns a value or writes to storage.
Examples	`map()`, `filter()`, `flatMap()`	`collect()`, `count()`, `show()`

Why Lazy Evaluation Matters

Lazy evaluation allows Spark to optimize the execution plan:

1. **Minimizing Data Movement**: Spark analyzes the entire computation chain to reduce shuffling.
2. **Combining Operations**: Spark can merge multiple transformations into a single stage.

Key Takeaway

- Use **transformations** to define how data should be manipulated.
- Use **actions** to trigger execution and extract results.
- Understanding their roles helps you write efficient and optimized Spark applications.

Practical Example: Word Count with RDDs

Problem: Count the occurrences of each word in a text file.

```
# Read the text file
text_rdd = spark.sparkContext.textFile("datasets\social_media_comments\sentimentdataset.txt")

# Split each line into words and flatten
words_rdd = text_rdd.flatMap(lambda line: line.split(" "))

# Map each word to a (word, 1) pair
word_pairs = words_rdd.map(lambda word: (word, 1))

# Reduce by key (word) to count occurrences
word_counts = word_pairs.reduceByKey(lambda x, y: x + y)

# Collect and display results
for word, count in word_counts.collect():
    print(f"{word}: {count}")
```

Code Explaination

```
# Read the text file
text_rdd = spark.sparkContext.textFile("datasets/social_media_comments/sentimentdataset.txt")
```

Reads the file and divides it into chunks that are distributed across worker nodes. Each worker is responsible for processing its assigned lines, allowing for parallel reading.

```
# Split each line into words and flatten
words_rdd = text_rdd.flatMap(lambda line: line.split(" "))
```

Splits each line into words within each worker. Since the data was already distributed in the previous step, flatMap simply applies the splitting operation on each worker's assigned lines independently.

```
# Map each word to a (word, 1) pair
word_pairs = words_rdd.map(lambda word: (word, 1))
```

Converts each word to a `(word, 1)` pair on each worker. The `map` function sends this transformation to each worker, where it operates on its data independently.

```
<>:2: SyntaxWarning: invalid escape sequence '\s'
<>:2: SyntaxWarning: invalid escape sequence '\s'
/tmp/ipykernel_78360/2155919677.py:2: SyntaxWarning: invalid escape sequence '\s'
```

```
# Reduce by key (word) to count occurrences
word_counts = word_pairs.reduceByKey(lambda x, y: x + y)
```

```
Py4JJavaError                             Traceback (most recent call last)
      8 word_pairs = words_rdd.map(lambda word: (word, 1))
     10 # Reduce by key (word) to count occurrences
```

Aggregates the counts by word. Initially, each worker performs a local aggregation for the words it holds. Then, a shuffle occurs, redistributing data so all occurrences of the same word go to the same worker for final aggregation.

```
# Collect and display results
for word, count in word_counts.collect():
    print(f"{word}: {count}")
```

```
   3505 def reduceByKey(
   3506     self: "RDD[Tuple[K, V]]",
   3507     func: Callable[[V, V], V],
   3508     numPartitions: Optional[int] = None,
   3509     partitionFunc: Callable[[K], int] = portable_hash,
   3510 ) -> "RDD[Tuple[K, V]]":
   3511     """
   3512     Merge the values for each key using an associative and commutative reduce function.
```

Collects the final counts from all workers to the driver program. Each worker sends its results to the driver, where the data is combined and displayed.

sorting the counts in descending order

```
# Read the text file
text_rdd = spark.sparkContext.textFile("datasets/social_media_comments/sentimentdataset.txt")

# Split each line into words and flatten
words_rdd = text_rdd.flatMap(lambda line: line.split(" "))

# Map each word to a (word, 1) pair
word_pairs = words_rdd.map(lambda word: (word, 1))

# Reduce by key (word) to count occurrences
word_counts = word_pairs.reduceByKey(lambda x, y: x + y)

# Sort by count in descending order and take the top 10
top_10_words = word_counts.sortBy(lambda x: x[1], ascending=False).take(10)

# Display the top 10 results
for word, count in top_10_words:
    print(f"{word}: {count}")

```

```
: 2321
the: 808
of: 623
a: 621
in: 259
to: 133
and: 111
with: 107
for: 99
on: 91
```

```
-> 4867     return self.getNumPartitions()

File ~/myenv/lib/python3.12/site-packages/pyspark/rdd.py:5453, in PipelinedRDD.getNumPartitions(self)
   5452 def getNumPartitions(self) -> int:
-> 5453     return self._prev_jrdd.partitions().size()

File ~/myenv/lib/python3.12/site-packages/py4j/java_gateway.py:1322, in JavaMember.__call__(self, *args)
   1316 command = proto.CALL_COMMAND_NAME +\
   1317     self.command_header +\
   1318     args_command +\
   1319     proto.END_COMMAND_PART
   1321 answer = self.gateway_client.send_command(command)
-> 1322 return_value = get_return_value(
   1323     answer, self.gateway_client, self.target_id, self.name)
   1325 for temp_arg in temp_args:
   1326     if hasattr(temp_arg, "_detach"):
```

This word count example demonstrates several core RDD concepts, including transformations (flatMap, map, reduceByKey) and actions (collect).

```
File ~/myenv/lib/python3.12/site-packages/pyspark/errors/exceptions/captured.py:179, in capture_sql_exception.<locals>.deco(*a, **kw)
    177 def deco(*a: Any, **kw: Any) -> Any:
    178     try:
--> 179         return f(*a, **kw)
    180     except Py4JJavaError as e:
    181         converted = convert_exception(e.ja
```

Double-click (or enter) to edit

4. Introducing Spark SQL Tutorial: Using `apartment_prices.csv`

```
File ~/myenv/lib/python3.12/site-packages/py4j/protocol.py:326, in get_return_value(answer, gateway_client, target_id, name)
    324 value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    325 if answer[1] == REFERENCE_TYPE:
--> 326     raise Py4JJavaError(
    327         "An error occurred while calling {0}{1}{2}.\n".
    328         format(target_id, ".", name), value)
    329 else:
    330     raise Py4JError(
    331         "An error occurred while calling {0}{1}{2}. Trace:\n{3}\n".
    332         format(target_id, ".", name, value))
```

Spark SQL is a powerful module in Apache Spark for processing structured data. It enables SQL-like querying of data and integrates seamlessly with Spark's core APIs.

Key capabilities:

- Query structured data using SQL.
- Work with various data formats (CSV, JSON, Parquet).
- Combine SQL with Spark's DataFrame API for powerful analytics.

```
Py4JJavaError: An error occurred while calling o426.partitions.
: org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: file:/media/me/Disk1-Repo
```

```
1/my_code/my_courses/307401-Big-Data/Apache Spark/datasets\social_media_comments\sentimentdataset.txt
    at org.apache.hadoop.mapred.FileInputFormat.singleThreadedListStatus(FileInputFormat.java:304)
    at org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:244)
    at org.apache.hadoop.mapred.FileInputFormat.getSplits(FileInputFormat.java:332)
    at org.apache.spark.rdd.HadoopRDD.getPartitions(HadoopRDD.scala:208)
    at org.apache.spark.rdd.RDD.$anonfun$partitions$2(RDD.scala:294)
    at scala.Option.getOrElse(Option.scala:189)
    at org.apache.spark.rdd.RDD.partitions(RDD.scala:290)
```

Setting Up Spark SQL

Create a SparkSession

The `SparkSession` is the entry point for working with Spark SQL.

```python
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder.appName("Apartment Prices Analysis").getOrCreate()
```

```
    at org.apache.spark.api.java.AbstractJavaRDDLike.partitions(JavaRDDLike.scala:45)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:75)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:52)
    at java.base/java.lang.reflect.Method.invoke(Method.java:580)
    at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
    at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:374)
    at py4j.Gateway.invoke(Gateway.java:282)
    at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
```

Loading the Dataset

We'll load the provided `apartment_prices.csv` into a Spark DataFrame for analysis.

Load the Dataset

```python
# Load CSV file into a DataFrame
df = spark.read.csv("datasets/apartment_prices.csv", header=True, inferSchema=True)

# Show the schema and a few rows of the dataset
df.printSchema()
df.show(5)
```

```
root
 |-- Square_Area: integer (nullable = true)
 |-- Num_Rooms: integer (nullable = true)
 |-- Age_of_Building: integer (nullable = true)
 |-- Floor_Level: integer (nullable = true)
 |-- City: string (nullable = true)
 |-- Price: double (nullable = true)

+-----------+---------+---------------+-----------+-----+-------+
|Square_Area|Num_Rooms|Age_of_Building|Floor_Level| City|  Price|
+-----------+---------+---------------+-----------+-----+-------+
|        162|        1|             15|         12|Amman|74900.0|
|        152|        5|              8|          8|Aqaba|79720.0|
|         74|        3|              2|          8|Irbid|43200.0|
|        166|        1|              3|         18|Irbid|69800.0|
|        131|        3|             14|         15|Aqaba|63160.0|
+-----------+---------+---------------+-----------+-----+-------+
only showing top 5 rows
```

Registering the DataFrame as a SQL Table

To query the dataset using SQL, we register the DataFrame as a temporary table.

```python
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("apartments")
```

SQL Operations on the Dataset

a. Basic SELECT Query**

Retrieve all apartments located in "Amman."

```python
result = spark.sql("SELECT * FROM apartments WHERE City = 'Amman'")
result.show()
```

```
+-----------+---------+---------------+-----------+-----+--------+
|Square_Area|Num_Rooms|Age_of_Building|Floor_Level| City|   Price|
+-----------+---------+---------------+-----------+-----+--------+
|        162|        1|             15|         12|Amman| 74900.0|
|        134|        4|              4|          4|Amman| 80300.0|
|        163|        4|             18|         10|Amman| 85350.0|
|         97|        4|             19|         12|Amman| 56650.0|
|        117|        1|              4|         19|Amman| 72650.0|
|        108|        1|              1|          4|Amman| 56600.0|
|         74|        1|              5|          8|Amman| 41300.0|
|        110|        5|             11|         19|Amman| 82500.0|
```

```
|        110|        5|              5|         19|Amman| 88500.0|
|         80|        5|              6|          9|Amman| 64000.0|
|        132|        2|              9|          3|Amman| 63400.0|
|        191|        5|             12|         19|Amman|117950.0|
|        149|        3|             14|         12|Amman| 80050.0|
|        143|        1|             19|          4|Amman| 54350.0|
|        163|        1|              7|          2|Amman| 73350.0|
|        191|        3|              9|          4|Amman| 95950.0|
|        193|        1|             14|         15|Amman| 92850.0|
|         73|        1|              6|         19|Amman| 50850.0|
|         99|        5|              9|         13|Amman| 73550.0|
|        183|        5|              6|          3|Amman|104350.0|
+-----------+---------+---------------+-----------+-----+--------+
only showing top 20 rows
```

b. Aggregations

Calculate the average price of apartments grouped by the number of bedrooms.

```
result = spark.sql("SELECT Num_Rooms, AVG(Price) AS avg_price FROM apartments GROUP BY Num_Rooms")
result.show()
```

```
+---------+------------------+
|Num_Rooms|         avg_price|
+---------+------------------+
|        1| 56153.36363636364|
|        3|61384.903846153844|
|        5| 76516.07843137255|
|        4|  66747.1264367816|
|        2| 57040.51546391752|
+---------+------------------+
```

c. Sorting Data

List the top 5 most expensive apartments.

```
result = spark.sql("SELECT * FROM apartments ORDER BY price DESC LIMIT 5")
result.show()
```

```
+-----------+---------+---------------+-----------+-----+--------+
|Square_Area|Num_Rooms|Age_of_Building|Floor_Level| City|   Price|
+-----------+---------+---------------+-----------+-----+--------+
|        199|        4|              2|         16|Amman|123550.0|
|        183|        5|              4|         19|Amman|122350.0|
|        191|        5|             12|         19|Amman|117950.0|
|        187|        4|              7|         19|Amman|116150.0|
|        160|        5|              1|         15|Amman|111000.0|
+-----------+---------+---------------+-----------+-----+--------+
```

d. Filtering and Conditions

Find apartments with more than 3 bedrooms and priced below 200,000.

```
result = spark.sql("""
    SELECT *
    FROM apartments
    WHERE Num_Rooms > 3 AND Price < 200000
""")
result.show()
```

```
+-----------+---------+---------------+-----------+-----+--------+
|Square_Area|Num_Rooms|Age_of_Building|Floor_Level| City|   Price|
+-----------+---------+---------------+-----------+-----+--------+
|        152|        5|              8|          8|Aqaba| 79720.0|
|         80|        4|             14|          7|Aqaba| 41800.0|
|        181|        4|             16|         16|Aqaba| 85160.0|
|        134|        4|              4|          4|Amman| 80300.0|
|        147|        5|              5|          6|Aqaba| 78920.0|
|        159|        4|             16|          9|Irbid| 60700.0|
|        163|        4|             18|         10|Amman| 85350.0|
|         61|        4|              7|         18|Aqaba| 52960.0|
|         97|        4|             19|         12|Amman| 56650.0|
|        189|        4|             16|         13|Aqaba| 85040.0|
```

```
|         80|        5|             19|         13|Aqaba| 47800.0|
|         81|        4|             18|         19|Aqaba| 50160.0|
|        110|        5|             11|         19|Amman| 82500.0|
|        167|        4|             10|         12|Irbid| 72100.0|
|        114|        5|              2|         18|Irbid| 75200.0|
|        123|        4|              3|          6|Aqaba| 67280.0|
|        190|        5|             12|         18|Aqaba| 99400.0|
|        110|        5|              5|         19|Amman| 88500.0|
|         80|        5|              6|          9|Amman| 64000.0|
|        191|        5|             12|         19|Amman|117950.0|
+-----------+---------+---------------+-----------+-----+--------+
only showing top 20 rows
```

Writing Query Results to a File

Save the filtered data (apartments in "Amman") to a new CSV file.

```
result = spark.sql("SELECT * FROM apartments WHERE location = 'Amman'")
result.write.csv("/mnt/data/amman_apartments.csv", header=True)
```

Using Built-in SQL Functions

a. String Manipulation

Convert all location names to uppercase.

```
1 result = spark.sql("SELECT UPPER(City) AS location_upper, Square_Area, Price FROM apartments")
2 result.show()
```

```
+--------------+-----------+-------+
|location_upper|Square_Area|  Price|
+--------------+-----------+-------+
|         AMMAN|        162|74900.0|
|         AQABA|        152|79720.0|
|         IRBID|         74|43200.0|
|         IRBID|        166|69800.0|
|         AQABA|        131|63160.0|
|         AQABA|         80|41800.0|
|         AQABA|        162|68320.0|
|         AQABA|        181|85160.0|
|         AMMAN|        134|80300.0|
|         AQABA|        147|78920.0|
|         IRBID|        176|51800.0|
|         IRBID|        159|60700.0|
|         AMMAN|        163|85350.0|
|         IRBID|        190|74000.0|
|         IRBID|        112|44600.0|
|         AQABA|         61|52960.0|
|         IRBID|        147|65100.0|
|         AMMAN|         97|56650.0|
|         AQABA|        189|85040.0|
|         AQABA|         80|47800.0|
+--------------+-----------+-------+
only showing top 20 rows
```

b. Numeric Functions

Calculate the price per square foot for each apartment.

```
1 result = spark.sql("SELECT City, Square_Area, Price, (Price / Square_Area) AS price_per_sqft FROM apartments")
2 result.show()
```

```
+-----+-----------+-------+------------------+
| City|Square_Area|  Price|    price_per_sqft|
+-----+-----------+-------+------------------+
|Amman|        162|74900.0|462.34567901234567|
|Aqaba|        152|79720.0| 524.4736842105264|
|Irbid|         74|43200.0| 583.7837837837837|
|Irbid|        166|69800.0|420.48192771084337|
|Aqaba|        131|63160.0| 482.1374045801527|
|Aqaba|         80|41800.0|             522.5|
|Aqaba|        162|68320.0| 421.7283950617284|
```

```
|Aqaba|        181|85160.0|470.49723756906076|
|Amman|        134|80300.0| 599.2537313432836|
|Aqaba|        147|78920.0| 536.8707482993198|
|Irbid|        176|51800.0| 294.3181818181818|
|Irbid|        159|60700.0|381.76100628930817|
|Amman|        163|85350.0| 523.6196319018405|
|Irbid|        190|74000.0| 389.4736842105263|
|Irbid|        112|44600.0| 398.2142857142857|
|Aqaba|         61|52960.0| 868.1967213114754|
|Irbid|        147|65100.0|442.85714285714283|
|Amman|         97|56650.0|  584.020618556701|
|Aqaba|        189|85040.0| 449.9470899470899|
|Aqaba|         80|47800.0|             597.5|
+-----+-----------+-------+------------------+
only showing top 20 rows
```

c. Statistical Analysis

Find the minimum, maximum, and average apartment prices.

```
result = spark.sql("""
    SELECT
        MIN(price) AS min_price,
        MAX(price) AS max_price,
        AVG(price) AS avg_price
    FROM apartments
""")
result.show()
```

```
+---------+---------+---------+
|min_price|max_price|avg_price|
+---------+---------+---------+
|  15900.0| 123550.0| 63410.94|
+---------+---------+---------+
```

End-to-End Example

1. Load the dataset.
2. Filter apartments with at least 2 bedrooms and priced below 150,000.
3. Group them by location and calculate the average price.
4. Save the results.

```
# Step 1: Filter data
filtered_data = spark.sql("""
    SELECT *
    FROM apartments
    WHERE Num_Rooms >= 2 AND price < 150000
""")

# Step 2: Group and aggregate
aggregated_data = spark.sql("""
    SELECT City, AVG(price) AS avg_price
    FROM apartments
    WHERE Num_Rooms >= 2 AND price < 150000
    GROUP BY City
""")

# Step 3: Save results to a file
# aggregated_data.write.csv("/mnt/data/filtered_apartments.csv", header=True)
```

Machine Learning with Apache Spark: Predicting Apartment Prices

In this notebook, we'll explore the basics of machine learning in Apache Spark using the MLlib library. Specifically, we'll build a regression model to predict apartment prices based on features like square area, number of rooms, age of the building, and floor level.

Step 1: Setting Up the Spark Environment

First, we need to set up a `SparkSession`, which is the main entry point for using Spark's DataFrame and MLlib capabilities. The `SparkSession` allows us to create and manipulate DataFrames and to access Spark's machine learning library.

```python
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder.appName("Apartment Price Prediction").getOrCreate()
```

```
24/11/15 21:02:14 WARN SparkSession: Using an existing Spark session; only runtime SQL configurations will take effect.
```

Step 2: Loading the Dataset

Next, we load the dataset containing apartment information and prices. Spark can read various file formats; here, we're loading a CSV file with headers and inferring the data types for each column. Once loaded, we display the schema and some sample rows to understand the data structure.

```python
# Load the dataset
data_path = "datasets/apartment_prices.csv"  # Adjust the path if needed
df = spark.read.csv(data_path, header=True, inferSchema=True)

# Show the schema and data
df.printSchema()
df.show()
```

```
root
 |-- Square_Area: integer (nullable = true)
 |-- Num_Rooms: integer (nullable = true)
 |-- Age_of_Building: integer (nullable = true)
 |-- Floor_Level: integer (nullable = true)
 |-- City: string (nullable = true)
 |-- Price: double (nullable = true)

+-----------+---------+---------------+-----------+-----+-------+
|Square_Area|Num_Rooms|Age_of_Building|Floor_Level| City|  Price|
+-----------+---------+---------------+-----------+-----+-------+
|        162|        1|             15|         12|Amman|74900.0|
|        152|        5|              8|          8|Aqaba|79720.0|
|         74|        3|              2|          8|Irbid|43200.0|
|        166|        1|              3|         18|Irbid|69800.0|
|        131|        3|             14|         15|Aqaba|63160.0|
|         80|        4|             14|          7|Aqaba|41800.0|
|        162|        2|             11|         11|Aqaba|68320.0|
|        181|        4|             16|         16|Aqaba|85160.0|
|        134|        4|              4|          4|Amman|80300.0|
|        147|        5|              5|          6|Aqaba|78920.0|
|        176|        2|             14|          3|Irbid|51800.0|
|        159|        4|             16|          9|Irbid|60700.0|
|        163|        4|             18|         10|Amman|85350.0|
|        190|        2|              7|         14|Irbid|74000.0|
|        112|        2|             10|         11|Irbid|44600.0|
|         61|        4|              7|         18|Aqaba|52960.0|
|        147|        2|              1|         12|Irbid|65100.0|
|         97|        4|             19|         12|Amman|56650.0|
|        189|        4|             16|         13|Aqaba|85040.0|
|         80|        5|             19|         13|Aqaba|47800.0|
+-----------+---------+---------------+-----------+-----+-------+
only showing top 20 rows
```

Step 3: Data Preprocessing – Handling Categorical Data

In machine learning, we need to convert categorical data into numerical representations. Here, the `City` column is a categorical feature that we need to transform. We use `StringIndexer` to assign a numeric index to each unique city, and then we apply `OneHotEncoder` to convert these indices into a one-hot encoded vector. This helps the model process categorical data effectively.

```python
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# Convert the 'City' column to a numeric index
indexer = StringIndexer(inputCol="City", outputCol="CityIndex")
df = indexer.fit(df).transform(df)

# Convert the numeric index to one-hot encoding
encoder = OneHotEncoder(inputCol="CityIndex", outputCol="CityVec")
df = encoder.fit(df).transform(df)
```

Step 4: Feature Engineering – Assembling Features

Spark's MLlib expects the features for each data point to be in a single vector column. We use the `VectorAssembler` to combine `Square_Area`, `Num_Rooms`, `Age_of_Building`, `Floor_Level`, and the one-hot encoded `CityVec` column into a single `features` column. We also rename the `Price` column to `label`, as MLlib expects the target variable to be named `label`.

```
from pyspark.ml.feature import VectorAssembler

# Assemble features into a single vector
assembler = VectorAssembler(inputCols=["Square_Area", "Num_Rooms", "Age_of_Building", "Floor_Level", "CityVec"], outputCol=
df = assembler.transform(df)

# Select the final columns for modeling
df = df.select("features", df["Price"].alias("label"))
df.show()
```

```
+--------------------+-------+
|            features|  label|
+--------------------+-------+
|[162.0,1.0,15.0,1...|74900.0|
|[152.0,5.0,8.0,8....|79720.0|
|[74.0,3.0,2.0,8.0...|43200.0|
|[166.0,1.0,3.0,18...|69800.0|
|[131.0,3.0,14.0,1...|63160.0|
|[80.0,4.0,14.0,7....|41800.0|
|[162.0,2.0,11.0,1...|68320.0|
|[181.0,4.0,16.0,1...|85160.0|
|[134.0,4.0,4.0,4....|80300.0|
|[147.0,5.0,5.0,6....|78920.0|
|[176.0,2.0,14.0,3...|51800.0|
|[159.0,4.0,16.0,9...|60700.0|
|[163.0,4.0,18.0,1...|85350.0|
|[190.0,2.0,7.0,14...|74000.0|
|[112.0,2.0,10.0,1...|44600.0|
|[61.0,4.0,7.0,18....|52960.0|
|[147.0,2.0,1.0,12...|65100.0|
|[97.0,4.0,19.0,12...|56650.0|
|[189.0,4.0,16.0,1...|85040.0|
|[80.0,5.0,19.0,13...|47800.0|
+--------------------+-------+
only showing top 20 rows
```

Step 5: Splitting the Dataset

To evaluate our model, we need to split the data into training and test sets. Typically, 80% of the data is used for training, and 20% is used for testing. This allows us to train the model on one portion of the data and then test its performance on unseen data.

```
# Split data into training and test sets
train_data, test_data = df.randomSplit([0.8, 0.2], seed=42)
```

Step 6: Building and Training a Linear Regression Model

Now, we initialize and train a **Linear Regression** model. Linear regression is a supervised learning algorithm commonly used for predicting numerical values. Here, it will help us predict apartment prices based on the features provided.

```
from pyspark.ml.regression import LinearRegression

# Initialize Linear Regression model
lr = LinearRegression(featuresCol="features", labelCol="label")

# Train the model on the training data
lr_model = lr.fit(train_data)

# Print model coefficients and intercept
print(f"Coefficients: {lr_model.coefficients}")
print(f"Intercept: {lr_model.intercept}")
```

```
24/11/15 21:02:37 WARN Instrumentation: [03494bb4] regParam is zero, which might cause numerical instability and overfitting.
24/11/15 21:02:37 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlib.blas.VectorBLAS
Coefficients: [371.418044269336,4978.707796345987,-1014.4774362162049,1035.5111204683565,11925.43813852181,-8114.944798383919]
Intercept: -1623.4506090574075
```

The output will show the coefficients (weights) for each feature, indicating how each feature impacts the apartment price, as well as the intercept term.

```
from pyspark.ml.evaluation import RegressionEvaluator

# Make predictions on the test data
predictions = lr_model.transform(test_data)

# Show predictions
predictions.select("features", "label", "prediction").show()

# Evaluate model using RMSE
evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE): {rmse}")


# Initialize RegressionEvaluator with R2 metric
evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction", metricName="r2")

# Evaluate model using R2
r2 = evaluator.evaluate(predictions)
print(f"R-squared: {r2}")
```

```
+--------------------+-------+------------------+
|            features|  label|        prediction|
+--------------------+-------+------------------+
|[60.0,3.0,14.0,3....|22000.0|16386.659892134998|
|[61.0,3.0,11.0,2....|24300.0| 18765.99912458459|
|[61.0,5.0,15.0,18...|55450.0|61274.065836811176|
|[63.0,4.0,14.0,12...|46350.0| 51839.60484240993|
|[65.0,3.0,19.0,16...|35400.0|34747.952296873205|
|[67.0,2.0,14.0,8....|28120.0| 27300.37880640007|
|[68.0,1.0,13.0,14...|36600.0|41846.071351871564|
|[71.0,2.0,4.0,3.0...|30300.0| 25638.32494491376|
|[73.0,2.0,17.0,1....|15900.0|11121.932121705046|
|[74.0,1.0,4.0,3.0...|30640.0| 29888.81607975969|
|[74.0,1.0,5.0,8.0...|41300.0| 45977.33238440708|
|[74.0,2.0,13.0,10...|40300.0|44911.242931960136|
|[74.0,5.0,19.0,10...|49300.0|53760.501703700866|
|[76.0,2.0,11.0,4....|37200.0| 41469.96717012108|
|[78.0,2.0,9.0,2.0...|31080.0|30245.297751633643|
|[80.0,5.0,6.0,9.0...|64000.0|  68141.7055196592|
|[83.0,1.0,10.0,18...|50350.0| 54602.81880643364|
|[89.0,1.0,8.0,6.0...|29700.0| 26393.76556195607|
|[93.0,4.0,2.0,11....|70850.0|  74120.3642846161|
|[94.0,3.0,10.0,5....|38200.0| 35143.80538309395|
+--------------------+-------+------------------+
only showing top 20 rows

Root Mean Squared Error (RMSE): 2875.771955223112
```

The predictions DataFrame shows the actual price (label) alongside the model's predicted price (prediction).

The RMSE (Root Mean Squared Error) provides a quantitative measure of the model's accuracy on the test data, where lower values indicate better model performance.

The R-squared (R^2) value represents the proportion of the variance in the target variable (e.g., price) that is explained by the model. An R^2 value closer to 1 indicates a better model fit, while a value closer to 0 indicates a poor fit.

Summary

In this notebook, we demonstrated how to use Spark MLlib to build a regression model for predicting apartment prices. The workflow included data preprocessing, feature engineering, model training, and evaluation. This example highlights Spark's ability to handle machine learning tasks on large datasets in a distributed environment, making it an excellent tool for scalable data processing and analysis.

Installing pyspark on local windows machine:

- install python 3.9
- add python to the path you can create a virtual environment and add it to the path

- install pyspark, pip install pyspark
- install java 11
- add JAVA_HOME = java folder to environment variables
- Under System Variables, click New to add new variables: Variable name: PYSPARK_PYTHON Variable value: python
- Repeat to add another variable: Variable name: PYSPARK_DRIVER_PYTHON Variable value: python

---

code-based comparison that demonstrates how **PySpark** outperforms **pandas** in processing larger datasets—even on a single machine like Google Colab or your local environment.

⚖️ Objective:

We'll compare PySpark and pandas on:

- Generating and processing a large dataset (e.g., 10 million rows).
- Performing a **group-by aggregation**, which is CPU-intensive.

🧪 Benchmark Setup

🐼 Pandas Benchmark

```
import time
import numpy as np
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg

# Generate large dataset
n = 10_000_000
np.random.seed(42)
df_pandas = pd.DataFrame({
    'category': np.random.randint(0, 1000, size=n),
    'value': np.random.rand(n)
})

# Time groupby operation
start = time.time()
result_pandas = df_pandas.groupby('category')['value'].mean()
end = time.time()

print(f"Pandas time: {end - start:.2f} seconds")
```

🔥 PySpark Benchmark

```
# Create Spark session
spark = SparkSession.builder \
    .appName("Pandas vs Spark Benchmark") \
    .master("local[*]") \
    .getOrCreate()

# Create Spark DataFrame
df_spark = spark.createDataFrame(df_pandas)

# Time groupby operation
start = time.time()
result_spark = df_spark.groupBy("category").agg(avg("value").alias("mean_value"))
result_spark.collect()  # Trigger computation
end = time.time()

print(f"PySpark time: {end - start:.2f} seconds")
```

🧾 Expected Output

On a system with 2–4 cores: