APACHE Spark™

Introduction

- Open-source, distributed computing system.
- Designed for fast and large-scale data processing.
- Provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.
- Written in Scala, but supports Java, Python, and R APIs.
- Components of Spark:
  - Spark Core
  - Spark SQL
  - Spark Streaming
  - MLlib
  - GraphX

Introduction …

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

Why do we need Apache Spark?

- **Challenges in Traditional Data Processing:**
  - Data stored in various systems (Hadoop, NoSQL, etc.) often requires complex integration.
  - Processing can be slow and inefficient due to disk-based storage.
  - Lack of fault tolerance and scalability.
- **How Spark Solves These Issues:**
  - In-memory computing for faster data processing.
  - Fault-tolerant, distributed nature.
  - Handles both batch and stream processing.
  - Scalable from a single server to thousands.

Evolution of Apache Spark

- **Early Days**:
  - Spark was created at UC Berkeley's AMPLab in 2009.
  - Spark was developed to overcome the limitations of MapReduce.
- **Key Milestones**:
  - 2010: Spark introduced as a research project.
  - 2014: Apache Spark graduated from an incubator to a top-level Apache project.
  - 2016: Spark became a leading technology for big data processing.
  - Continuous improvements, like better integration with other tools (Hadoop, Cassandra).

Spark Shell

- Spark Shell is an interactive shell that allows users to interact with a Spark cluster through a command-line interface.
- It’s a powerful tool for quickly testing code, debugging, and learning how Spark works.
- Spark’s shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively.
- It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python or R.

Why use Spark Shell?

- **Learning Tool**:
  - The Spark Shell is a great place for beginners to get hands-on experience with Spark.
  - You can run commands in real time, see outputs immediately, and test different Spark operations.
- **Testing Spark Jobs**:
  - Spark Shell allows users to experiment with small Spark jobs interactively.
  - You can test functions, transformations, and actions on a smaller scale before submitting them to a full Spark cluster.
- **Immediate Feedback**:
  - As an interactive interface, Spark Shell provides immediate feedback and allows for iterative development of your Spark programs.

Starting Spark Shell

- **Scala Shell**:
  - To start the Scala Spark Shell, you typically run the command:

  ```
  ./bin/spark-shell
  ```
  - Once the shell starts, you are connected to the SparkContext (sc) by default.
- **Python Shell (PySpark)**:
  - To use Python in Spark, use the

    ```
    ./bin/pyspark
    ```
  - This launches an interactive Python shell with Spark's context initialized, enabling the use of the Spark API in Python.
- **R Shell (SparkR)**:
  - If you prefer to use R, Spark also provides an R interface

  ```
  ./bin/sparkR
  ```
  - This opens an R console with the Spark session initialized, letting you work with Spark's data structures.

Components of Spark Shell

- **Spark Context (*sc*):**
  - The entry point to Spark is the spark context.
  - The sc object is automatically available when you start the shell.
  - It’s used to connect to the cluster, load data, and create RDDs.
- **Spark Session (*spark*):**
  - Introduced in Spark 2.x, Spark Session is now the entry point for Spark SQL and DataFrame operations.
  - It allows you to work with both structured and unstructured data using DataFrames and Datasets.

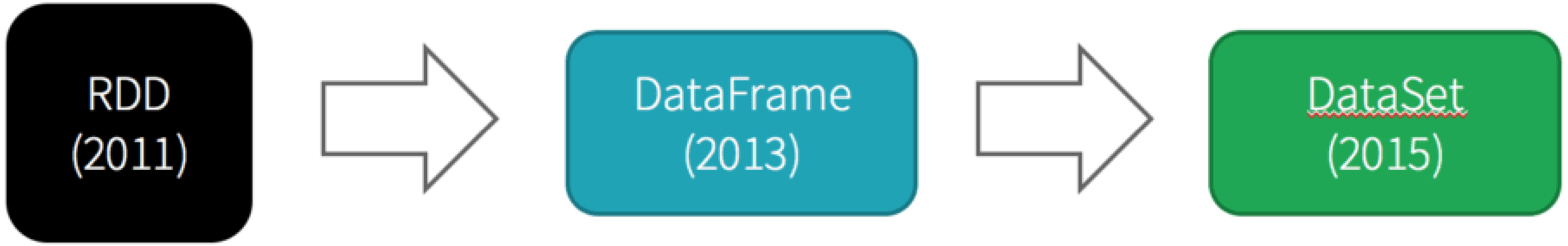
Components of Spark Shell

- **SQLContext (*sqlContext*):**
  - Available in older versions of Spark (pre-2.x), SQL Context is used for Spark SQL operations.
  - With newer versions, Spark Session is preferred over SQL Context.

Data Structures of Spark

- Apache Spark provides several key data structures that allow for efficient distributed computing.
- The primary data structures in Spark are:
    1. Resilient Distributed Dataset (RDD)
    2. DataFrame
    3. Dataset
- Each of these data structures serves a specific purpose in Spark's ecosystem.

History of Spark APIs



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the "Best of both worlds": type safe + fast

But slower than DF
Not as good for interactive analysis, especially Python

1. Resilient Distributed Dataset (RDD)

- RDD is the fundamental data structure in Spark.
- It represents a distributed collection of elements that can be processed in parallel across the nodes of a cluster.
- Resilient Distributed Dataset (RDD) is a collection of data elements that are spread across multiple nodes in a cluster.
- RDDs are the primary API that users interact with in Spark.
- Features of RDDs
    - **Fault-tolerant**: RDDs are fault-tolerant and can automatically recover from system faults.
    - **Parallel processing**: RDDs can be operated on in parallel because they are divided into logical partitions across the cluster.
    - **Immutable**: RDDs are immutable, meaning that once created, their contents cannot be changed.
    - **Low-level API**: RDDs are a low-level API that provides transformations and actions.

RDD…

- RDDs are designed for distributed computing, dividing the dataset into logical partitions.
- This logical partitioning enables efficient and scalable processing by distributing different data segments across different nodes within the cluster.
- RDDs can be created from various data sources, such as Hadoop Distributed File System (HDFS) or local file systems, and can also be derived from existing RDDs through transformations.
- Being the core abstraction in Spark, RDDs encompass a wide range of operations, including transformations (such as map, filter, and reduce) and actions (like count and collect).
- These operations allow users to perform complex data manipulations and computations on RDDs. RDDs provide fault tolerance by keeping track of the lineage information necessary to reconstruct lost partitions.

Reasons on When to use RDDs

- You want low-level transformation and actions and control on your dataset;
- Your data is unstructured, such as media streams or streams of text;
- You want to manipulate your data with functional programming constructs than domain specific expressions;
- You don’t care about imposing a schema, such as columnar format while processing or accessing data attributes by name or column; and
- You can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

Operations of RDD

- Two operations can be applied in RDD. One is transformation. And another one in action.
- **Transformations**
  - Transformations are the processes that you perform on an RDD to get a result which is also an RDD.
  - The example would be applying functions such as filter(), union(), map(), flatMap(), distinct(), reduceByKey(), mapPartitions(), sortBy() that would create an another resultant RDD.
  - Lazy evaluation is applied in the creation of RDD.
- **Actions**
  - Actions return results to the driver program or write it in a storage and kick off a computation. Some examples are count(), first(), collect(), take(), countByKey(), collectAsMap(), and reduce().
  - Transformations will always return RDD whereas actions return some other data type.

DataFrame

- A DataFrame is a distributed collection of data organized into named columns (similar to a table in a relational database).
- It is part of Spark's SQL module and provides an abstraction over RDDs with additional optimizations.
- **Key Features of DataFrame:**
  - **Schema**: DataFrames are structured and have a schema, which defines the columns and their data types.
  - **Optimized**: Spark SQL optimizes queries on DataFrames through the Catalyst Query Optimizer.
  - **Interoperable**: Can be created from RDDs, external data sources (e.g., CSV, JSON, Parquet), and can work seamlessly with Spark SQL.

Dataset

- A **Dataset** is a strongly typed version of a DataFrame. It combines the benefits of RDDs and DataFrames by providing both type safety and optimizations.
- Datasets are available in Scala and Java, but not in Python.
- Key Features of Dataset:
  - **Type-Safe**: Provides compile-time type safety, unlike DataFrames which are dynamically typed.
  - **Optimized**: Like DataFrames, Datasets benefit from the Catalyst Query Optimizer and Tungsten execution engine.
  - **Interoperable**: You can convert between Datasets and DataFrames.

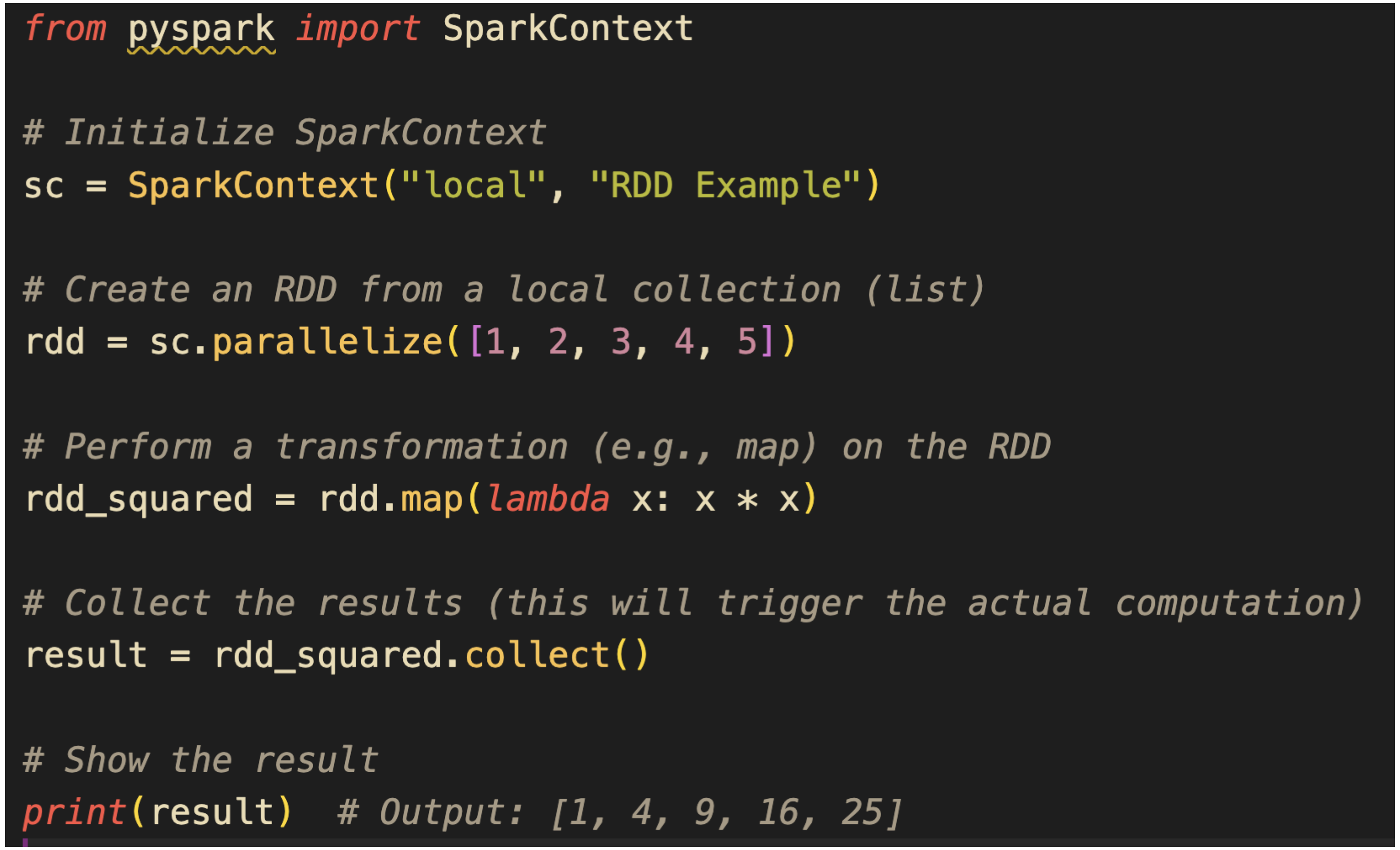
Comparision

Feature	RDD	DataFrame	DataSet
Type Safety	No (dynamically typed)	No (dynamically typed)	Yes (compile-time)
Abstraction Level	Low (raw data)	High (structured data)	High (strongly typed data)
Performance	Basic (no optimization)	Optimized (Catalyst Tungsten)	Optimized (Catalyst Tungsten)
Ease of Use	Requires manual handling	Simple API for SQL operations	Simple API with type safety
Compatibility	Works with Java, Scala, Python	Works with Java, Scala, Python	Works with Java, Scala
API	Functional, low-level	Declarative, SQL-Like	Functional, type-safe

When to use?

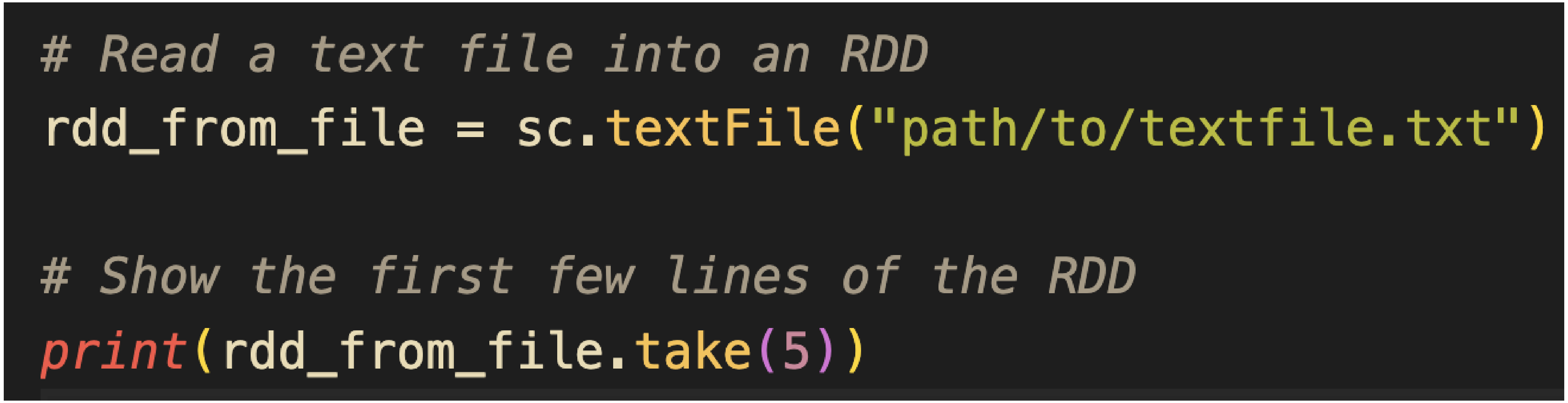
- **RDD**:
  - Use when you need fine-grained control over your data processing.
  - Ideal for low-level transformations or if working with unstructured data.
  - **Less optimized** than DataFrames or Datasets.
- **DataFrame**:
  - Use when you need structured data and optimization.
  - Great for SQL-like operations, querying, and working with large, structured datasets.
  - Automatically optimized via Catalyst and Tungsten engines.
- **Dataset**:
  - Use when you need **type safety** and want the benefits of both RDDs and DataFrames.
  - Ideal for strongly typed operations on str
  - uctured data in **Scala** or **Java**.

Creating RDD



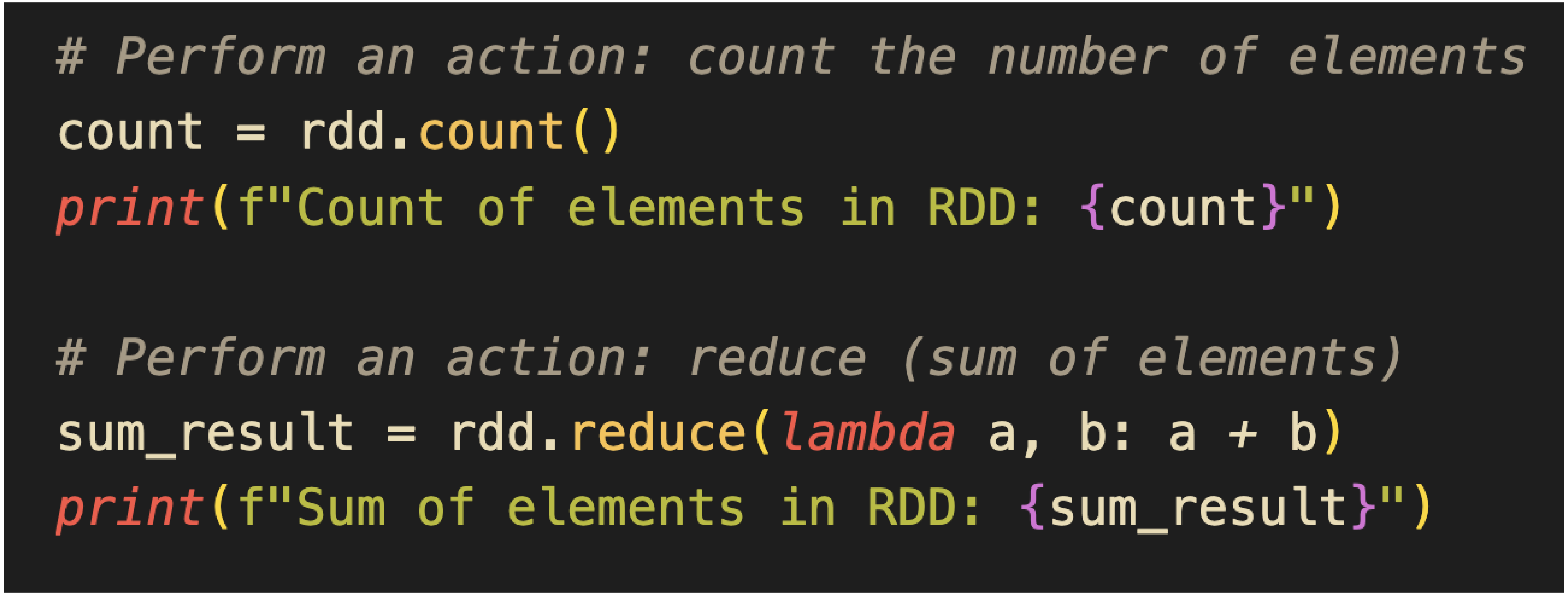
```python
from pyspark import SparkContext

# Initialize SparkContext
sc = SparkContext("local", "RDD Example")

# Create an RDD from a local collection (list)
rdd = sc.parallelize([1, 2, 3, 4, 5])

# Perform a transformation (e.g., map) on the RDD
rdd_squared = rdd.map(lambda x: x * x)

# Collect the results (this will trigger the actual computation)
result = rdd_squared.collect()

# Show the result
print(result)  # Output: [1, 4, 9, 16, 25]
```

Reading datafile into RDD



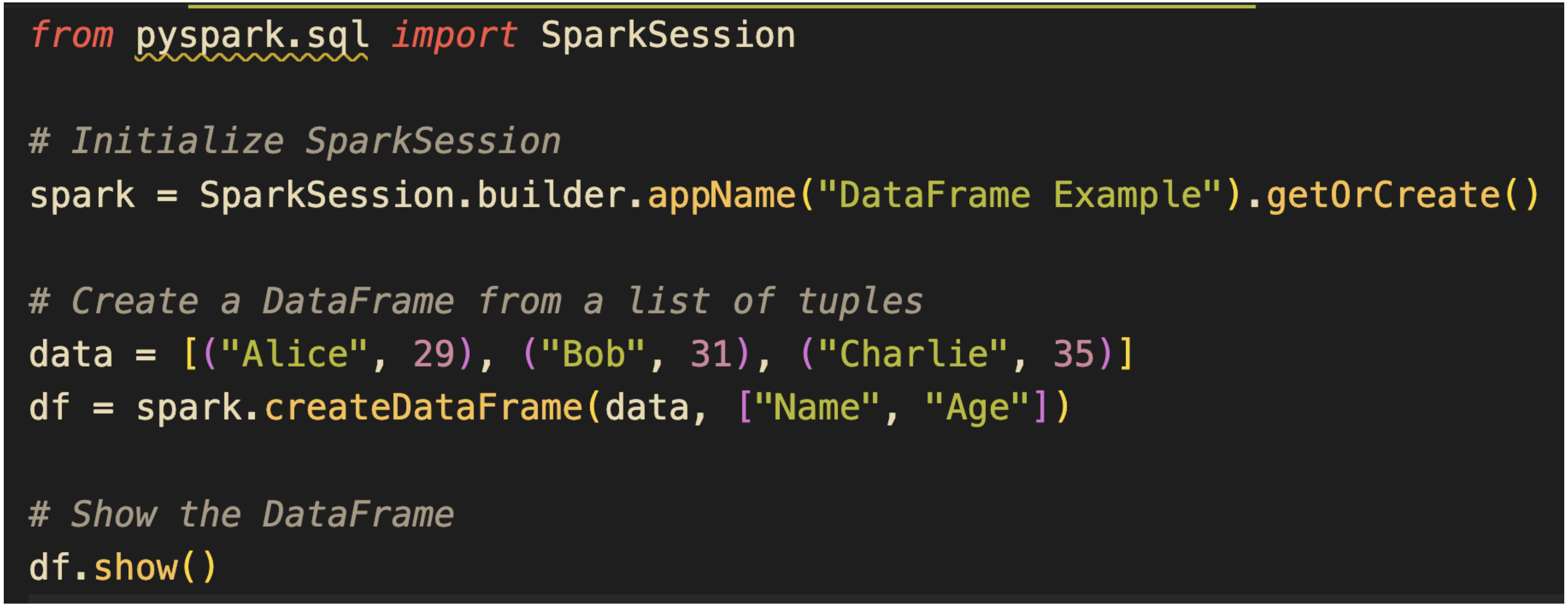
```
# Read a text file into an RDD
rdd_from_file = sc.textFile("path/to/textfile.txt")

# Show the first few lines of the RDD
print(rdd_from_file.take(5))
```

RDD Action



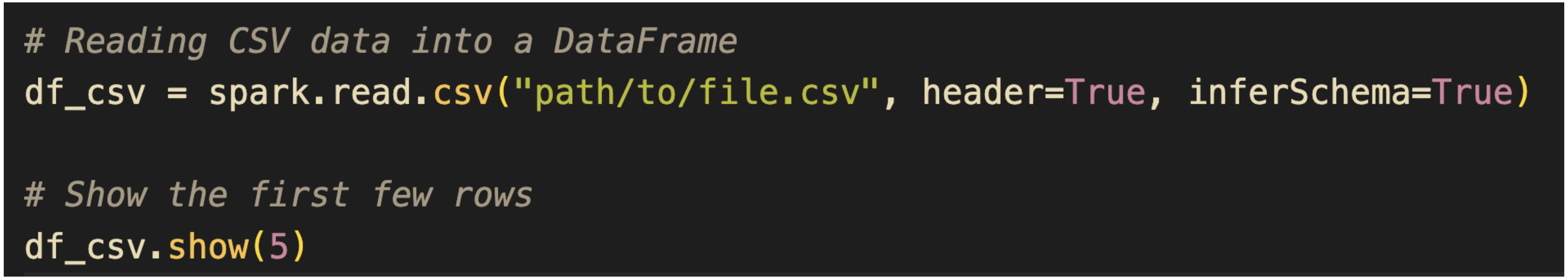
```python
# Perform an action: count the number of elements
count = rdd.count()
print(f"Count of elements in RDD: {count}")

# Perform an action: reduce (sum of elements)
sum_result = rdd.reduce(lambda a, b: a + b)
print(f"Sum of elements in RDD: {sum_result}")
```

Creating a DataFrame



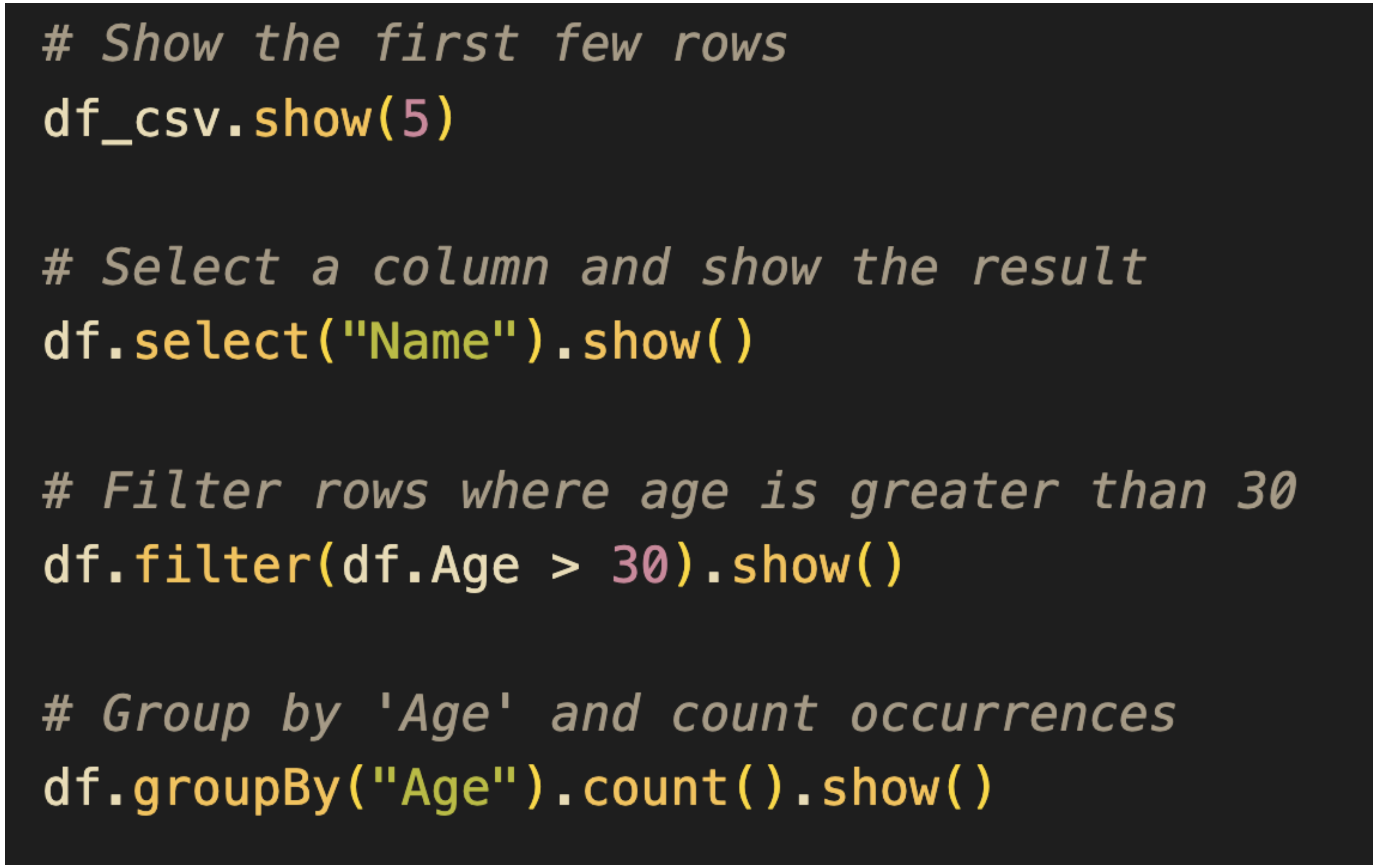
```python
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder.appName("DataFrame Example").getOrCreate()

# Create a DataFrame from a list of tuples
data = [("Alice", 29), ("Bob", 31), ("Charlie", 35)]
df = spark.createDataFrame(data, ["Name", "Age"])

# Show the DataFrame
df.show()
```

Reading file into DataFrame



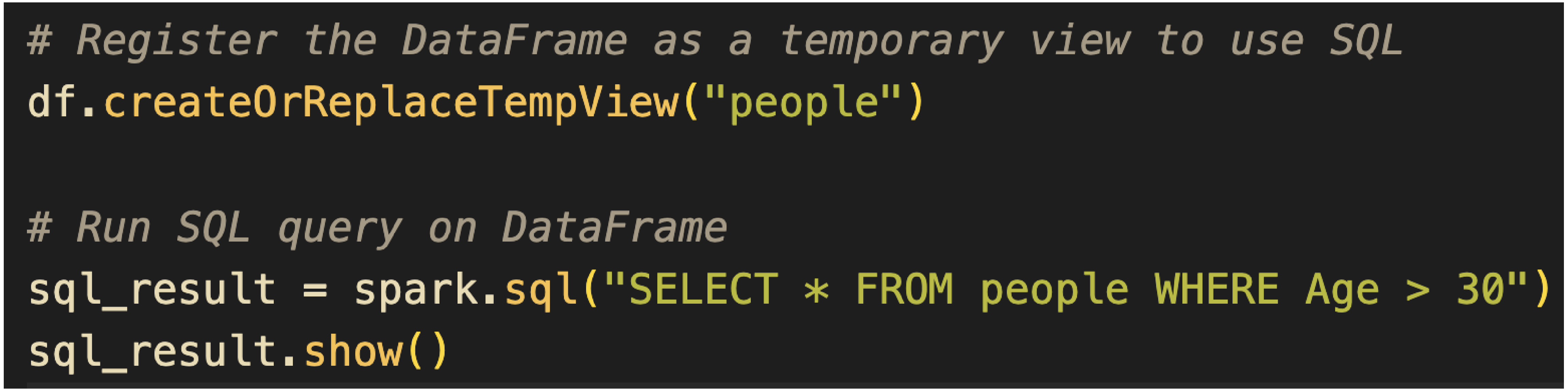
```python
# Reading CSV data into a DataFrame
df_csv = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)

# Show the first few rows
df_csv.show(5)
```

Performing Operations on DataFrame



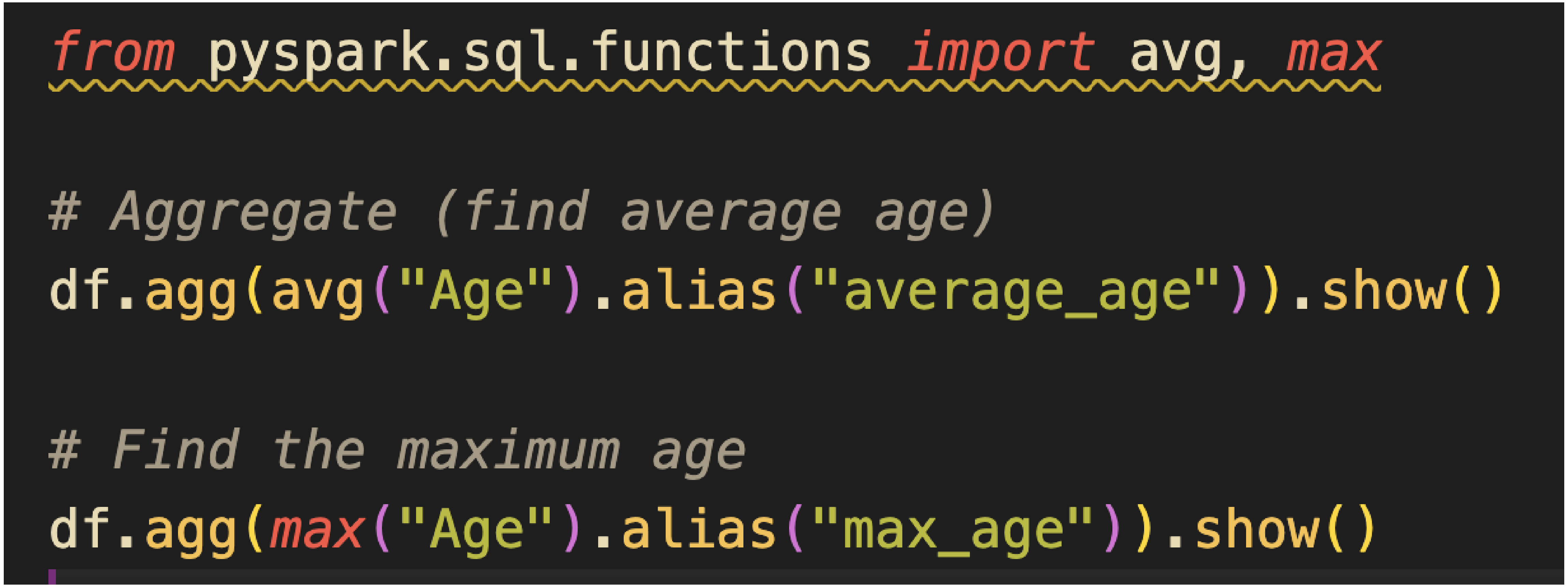
```python
# Show the first few rows
df_csv.show(5)

# Select a column and show the result
df.select("Name").show()

# Filter rows where age is greater than 30
df.filter(df.Age > 30).show()

# Group by 'Age' and count occurrences
df.groupBy("Age").count().show()
```

SQL Query on DataFrame



```
# Register the DataFrame as a temporary view to use SQL
df.createOrReplaceTempView("people")

# Run SQL query on DataFrame
sql_result = spark.sql("SELECT * FROM people WHERE Age > 30")
sql_result.show()
```

SQL Query – Aggregation



```
from pyspark.sql.functions import avg, max

# Aggregate (find average age)
df.agg(avg("Age").alias("average_age")).show()

# Find the maximum age
df.agg(max("Age").alias("max_age")).show()
```

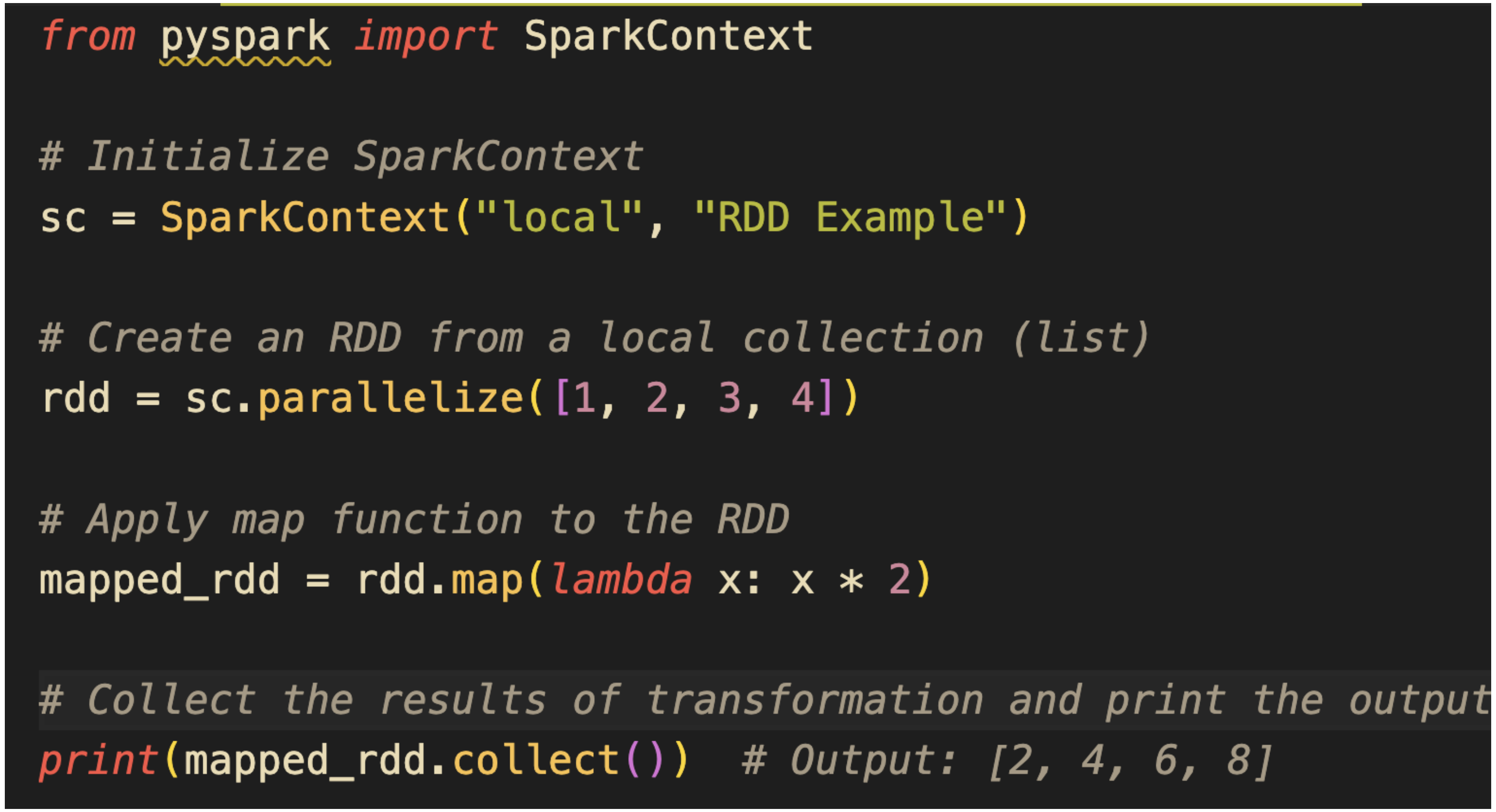
Transformations in RDD

- **Transformations** in Spark are **lazy operations** that create a new RDD from an existing one. These operations are not executed immediately but are instead recorded in the RDD's **lineage**.
- They define a **transformation pipeline**, which Spark executes only when an **action** is triggered (e.g., collect(), count()).Transformations allow for the parallel processing of data and can be applied to large datasets distributed across a cluster.
- **Transformations** in Spark are **lazy operations** that create a new RDD from an existing one.

Key Characteristics of RDD Transformations

- **Lazy Evaluation**:
  - Transformations are not executed immediately. They are only executed when an action is invoked (e.g., collect(), count()).
  - This helps optimize the execution plan.
- **Immutable**:
  - RDDs are immutable, meaning each transformation results in a new RDD, leaving the original RDD unchanged.
- **Wide vs. Narrow Transformations**:
  - **Narrow** transformations: Data from one partition is mapped to another (e.g., map(), filter()).
  - **Wide** transformations: Data from multiple partitions is shuffled to generate the resulting RDD (e.g., groupBy(), join()).

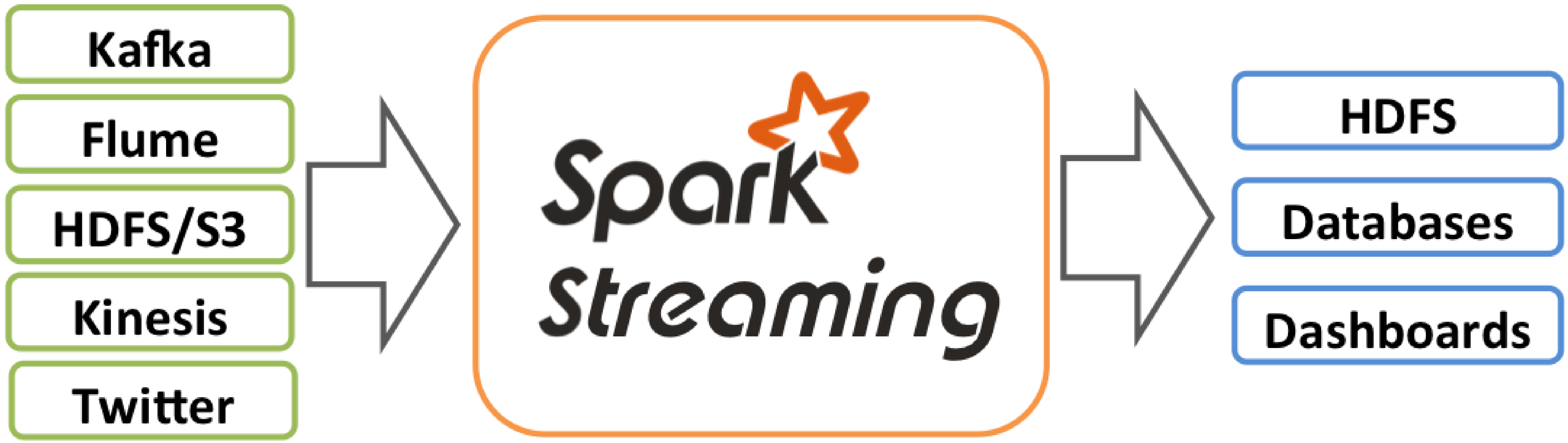
Example of RDD Transformation



```python
from pyspark import SparkContext

# Initialize SparkContext
sc = SparkContext("local", "RDD Example")

# Create an RDD from a local collection (list)
rdd = sc.parallelize([1, 2, 3, 4])

# Apply map function to the RDD
mapped_rdd = rdd.map(lambda x: x * 2)

# Collect the results of transformation and print the output
print(mapped_rdd.collect())  # Output: [2, 4, 6, 8]
```

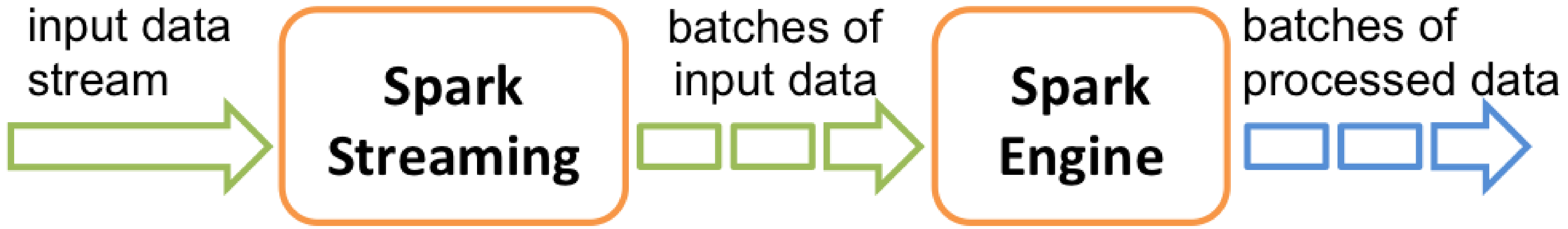
Spark Streaming

- It provides real-time stream process engine.
- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Data can be ingested from many sources like Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to filesystems, databases, and live dashboards.



Spark Streaming

- Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



Spark Streaming

- Spark Streaming is the previous generation of Spark’s streaming engine.
- There are no longer updates to Spark Streaming and it’s a legacy project. There is a newer and easier to use streaming engine in Spark called **Structured Streaming**.

Structured Streaming

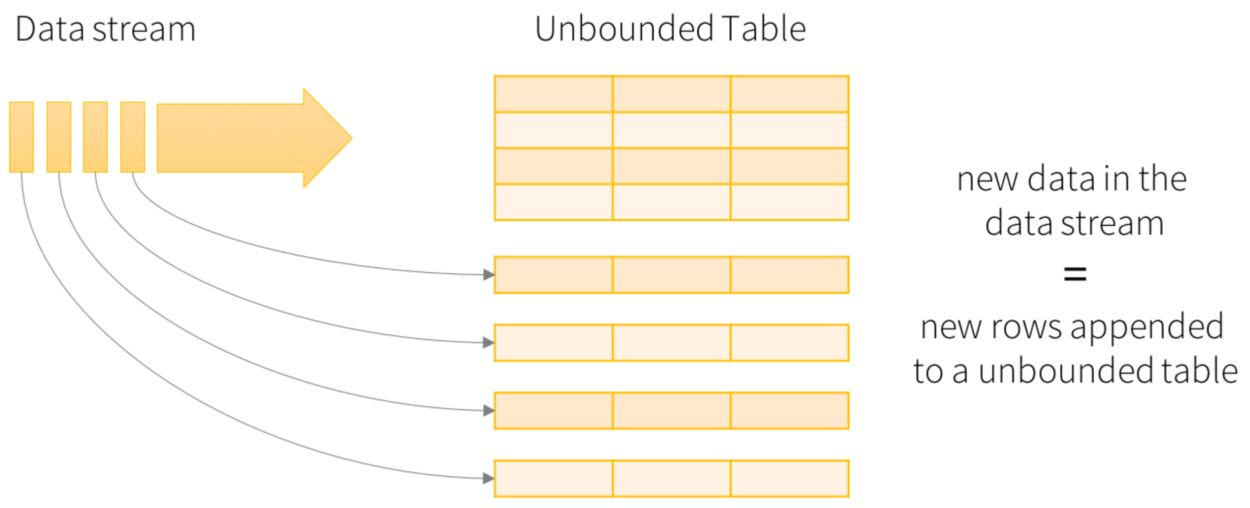
- Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2.
- Structured Streaming allows you to take the same operations that you perform in batch mode using Spark's structured APIs, and run them in a streaming fashion.
- This can reduce latency and allow for incremental processing.
- The best thing about Structured Streaming is that it allows you to rapidly and quickly get value out of streaming systems with virtually no code changes.
- It also makes it easy to reason about because you can write your batch job as a way to prototype it and then you can convert it to a streaming job.
- The way all of this works is by incrementally processing that data.

Structured Streaming…

- The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended.
- This leads to a new stream processing model that is very similar to a batch processing model.
- You will express your streaming computation as standard batch-like query as on a static table, and Spark runs it as an incremental query on the unbounded input table.

Structured Streaming...

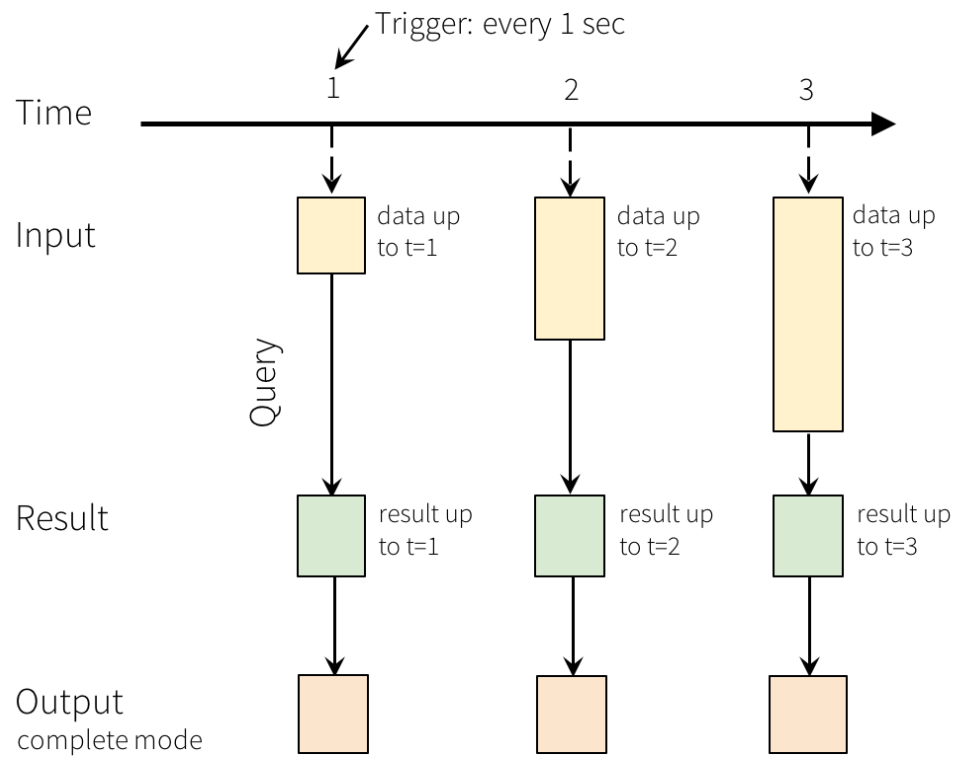
- Consider the input data stream as the “Input Table”. Every data item that is arriving on the stream is like a new row being appended to the Input Table.


Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

Structured Streaming…

- A query on the input will generate the "Result Table".
- Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table.
- Whenever the result table gets updated, we would want to write the changed result rows to an external sink.



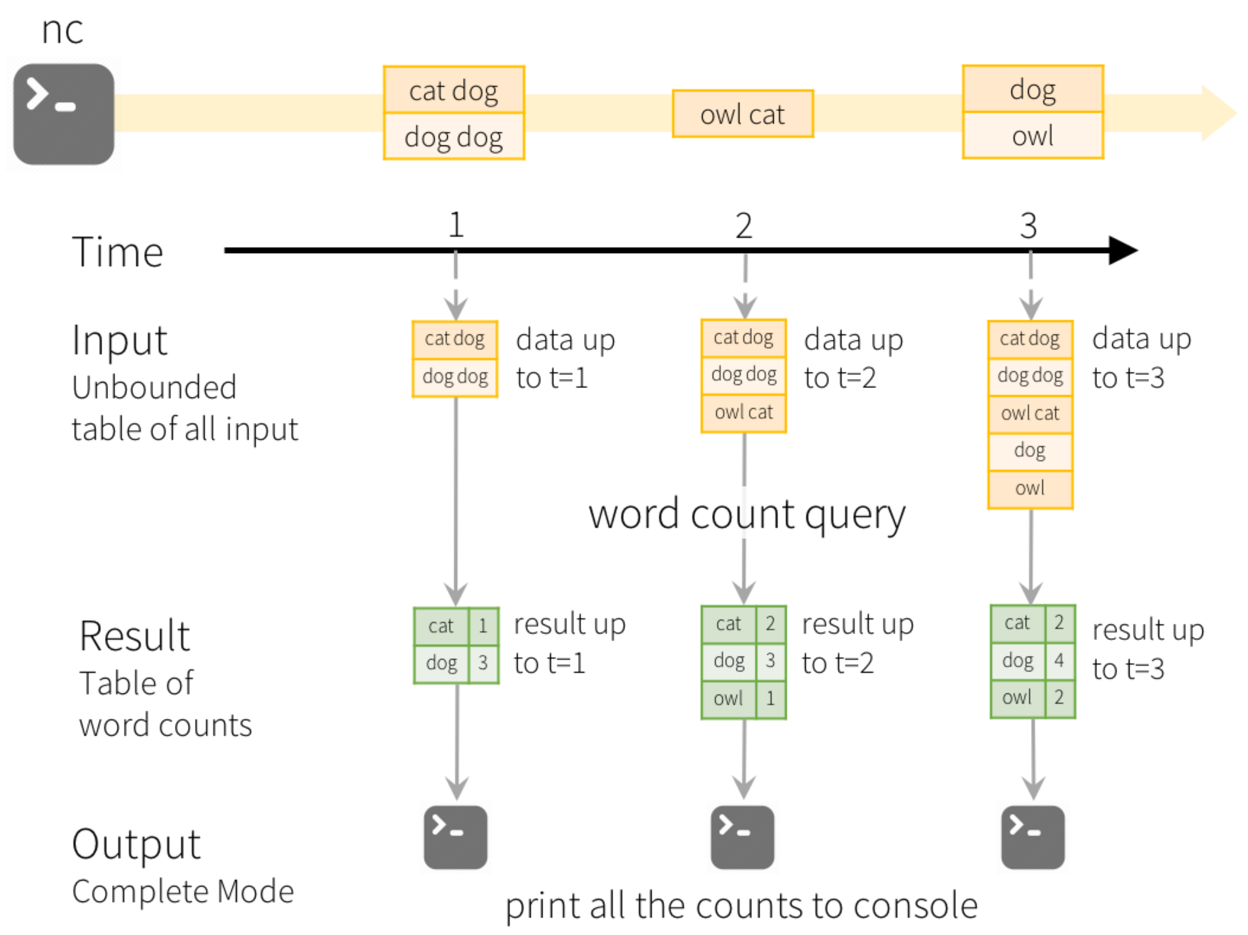
Programming Model for Structured Streaming

Structured Streaming

- The “Output” is defined as what gets written out to the external storage. The output can be defined in a different mode:
    - **Complete Mode** - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
    - **Append Mode** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.
    - **Update Mode** - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1). Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn’t contain aggregations, it will be equivalent to Append mode.

Structured Streaming

- Note that Structured Streaming does not materialize the entire table.
- It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data.
- It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).



Model of the Quick Example

GraphX

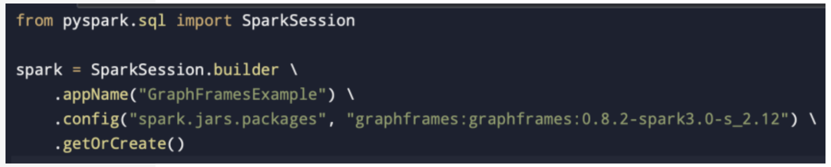
- GraphX is a new component in Spark for graphs and graph-parallel computation.
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
- To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API.
- In addition, GraphX includes a growing collection of graph algorithms and buildersto simplify graph analytics tasks.

GraphX

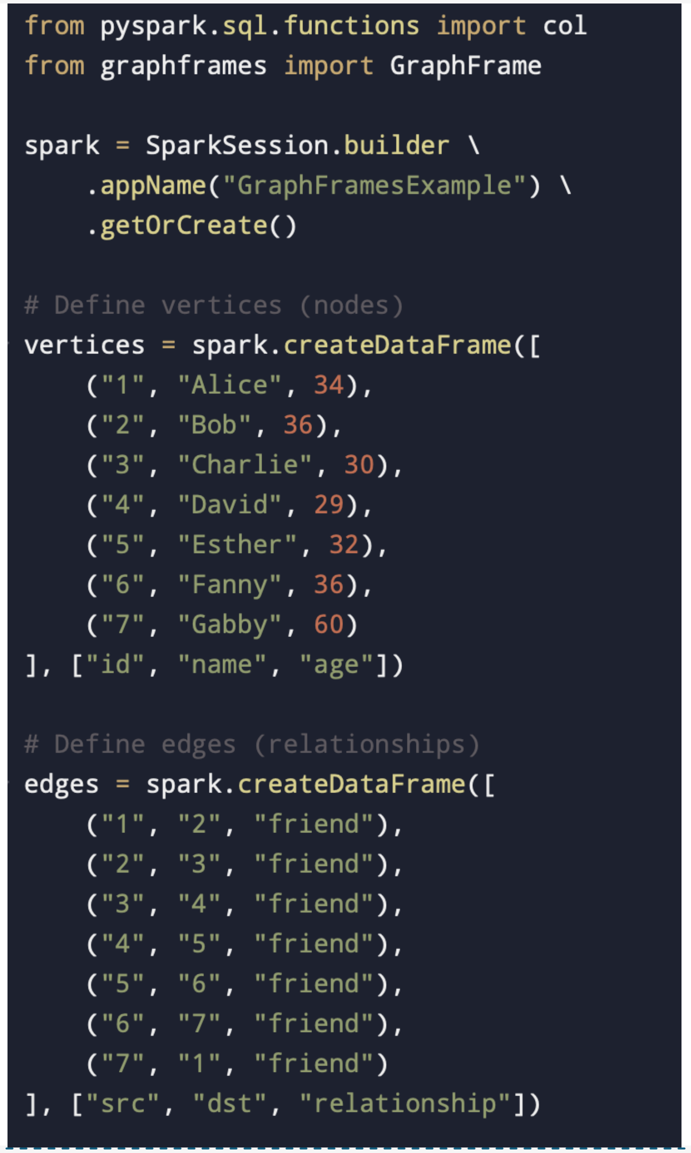
- GraphX introduces the concept of graphs as a directed multigraph of vertex and edges with attached user-defined properties.
- A directed multigraph is a graph which may have multiple directed edges between two nodes.
- GraphX allows us to seamlessly work with graphs and collections and efficiently transform and join graphs with RDDs.
- GraphX unifies various aspects of graph processing:
    1. GraphX helps perform the **ETL process** on graphs by allowing us to create, manipulate and join graphs with other data sources.
    2. GraphX helps to perform **Exploratory analysis** by providing different methods to filter and query graph data.
    3. GraphX offers different methods for **Iterative graph computations**. For example, PageRank is an algorithm in GraphX which iteratively computes a node's rank based on its neighbours' rank.

GraphX

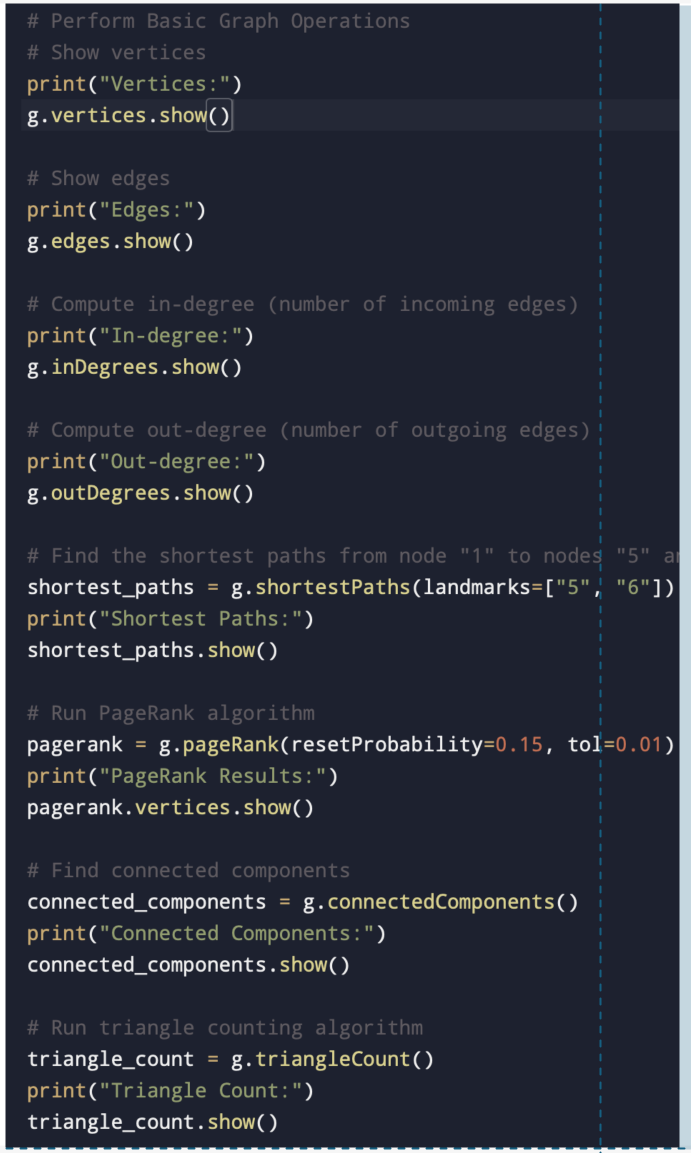
- Apache Spark GraphX is primarily available for Scala and Java, but in PySpark, you can use **GraphFrames**, which provides similar graph processing capabilities using DataFrames.



```python
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("GraphFramesExample") \
    .config("spark.jars.packages", "graphframes:graphframes:0.8.2-spark3.0-s_2.12") \
    .getOrCreate()
```



```python
from pyspark.sql.functions import col
from graphframes import GraphFrame

spark = SparkSession.builder \
    .appName("GraphFramesExample") \
    .getOrCreate()

# Define vertices (nodes)
vertices = spark.createDataFrame([
    ("1", "Alice", 34),
    ("2", "Bob", 36),
    ("3", "Charlie", 30),
    ("4", "David", 29),
    ("5", "Esther", 32),
    ("6", "Fanny", 36),
    ("7", "Gabby", 60)
], ["id", "name", "age"])

# Define edges (relationships)
edges = spark.createDataFrame([
    ("1", "2", "friend"),
    ("2", "3", "friend"),
    ("3", "4", "friend"),
    ("4", "5", "friend"),
    ("5", "6", "friend"),
    ("6", "7", "friend"),
    ("7", "1", "friend")
], ["src", "dst", "relationship"])
```



```python
# Perform Basic Graph Operations
# Show vertices
print("Vertices:")
g.vertices.show()

# Show edges
print("Edges:")
g.edges.show()

# Compute in-degree (number of incoming edges)
print("In-degree:")
g.inDegrees.show()

# Compute out-degree (number of outgoing edges)
print("Out-degree:")
g.outDegrees.show()

# Find the shortest paths from node "1" to nodes "5" ar
shortest_paths = g.shortestPaths(landmarks=["5", "6"])
print("Shortest Paths:")
shortest_paths.show()

# Run PageRank algorithm
pagerank = g.pageRank(resetProbability=0.15, tol=0.01)
print("PageRank Results:")
pagerank.vertices.show()

# Find connected components
connected_components = g.connectedComponents()
print("Connected Components:")
connected_components.show()

# Run triangle counting algorithm
triangle_count = g.triangleCount()
print("Triangle Count:")
triangle_count.show()
```

Use of GraphX

- Social Network Analysis
- Recommendation Systems
- Fraud Detection
- Transportation and Route Optimization
- Analyze Protein interaction networks in Genomics etc.

MLlib

- MLlib is Spark’s machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:
    - **ML Algorithms**: common learning algorithms such as classification, regression, clustering, and collaborative filtering
    - **Featurization**: feature extraction, transformation, dimensionality reduction, and selection
    - **Pipelines**: tools for constructing, evaluating, and tuning ML Pipelines
    - **Persistence**: saving and load algorithms, models, and Pipelines
    - **Utilities**: linear algebra, statistics, data handling, etc.

Features of MLlib

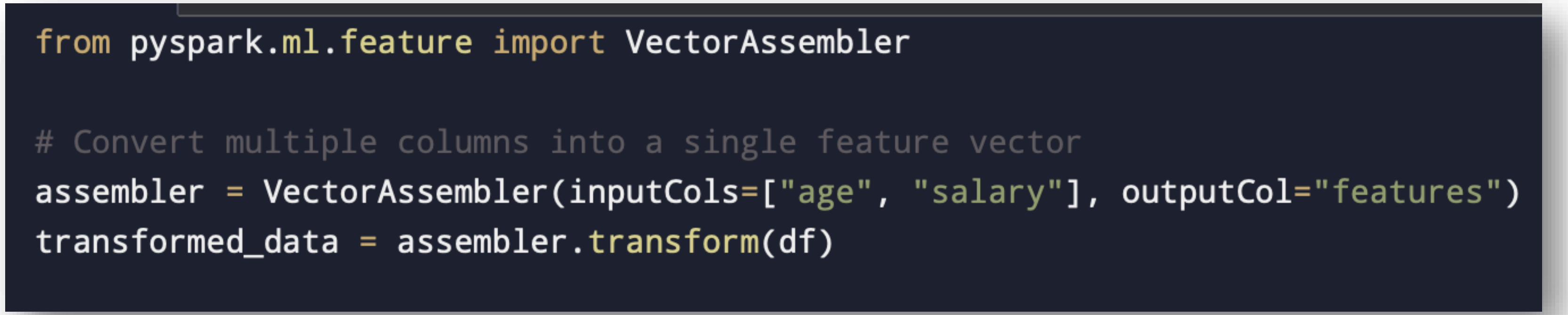
- **Scalability** – Designed to run on large-scale datasets using Spark’s distributed framework.
- **Ease of Use** – Provides APIs in Python (PySpark), Scala, Java, and R.
- **Performance** – Optimized algorithms outperform traditional ML libraries on large datasets.
- **Integration with Spark** – Works seamlessly with Spark DataFrames, GraphX, and Streaming.
- **ML Pipelines** – Supports feature extraction, transformation, and model tuning.

Components of MLlib

1. Feature Engineering
2. Supervised Learning
3. Unsupervised Learning
4. Recommendation System
5. Model Evaluation and Hyperparameter Tuning

MLlib – Feature Engineering

- **Feature Extraction & Transformation:** Convert raw data into structured features.
- **Dimensionality Reduction:** PCA, SVD, and Feature Selection.
- **Standardization & Normalization:** MinMaxScaler, StandardScaler.
- **One-Hot Encoding & String Indexing:** Convert categorical variables into numeric representations.



```python
from pyspark.ml.feature import VectorAssembler

# Convert multiple columns into a single feature vector
assembler = VectorAssembler(inputCols=["age", "salary"], outputCol="features")
transformed_data = assembler.transform(df)
```

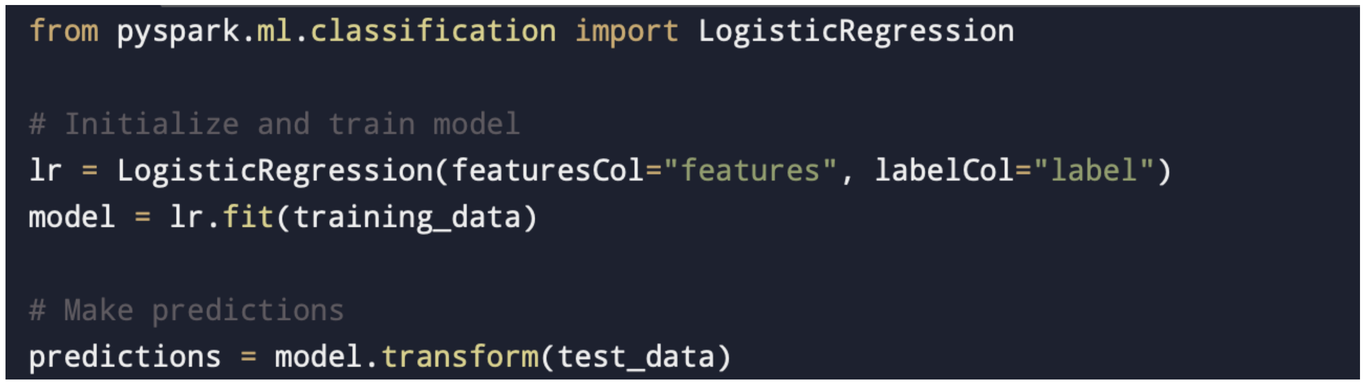
MLlib – Supervised Learning

- **Classification Algorithms**

✓ Logistic Regression
✓ Decision Trees
✓ Random Forest
✓ Gradient-Boosted Trees (GBT)
✓ Support Vector Machine (SVM)

- **Regression Algorithms**

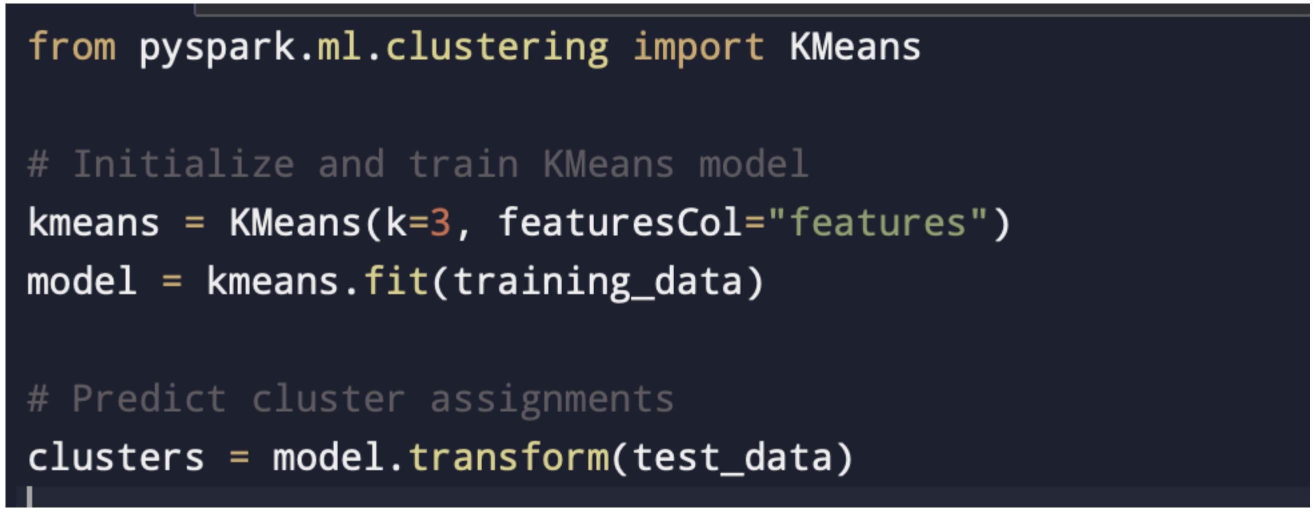
✓ Linear Regression
✓ Decision Tree Regression
✓ Random Forest Regression
✓ Generalized Linear Regression



```python
from pyspark.ml.classification import LogisticRegression

# Initialize and train model
lr = LogisticRegression(featuresCol="features", labelCol="label")
model = lr.fit(training_data)

# Make predictions
predictions = model.transform(test_data)
```

MLlib – Unsupervised

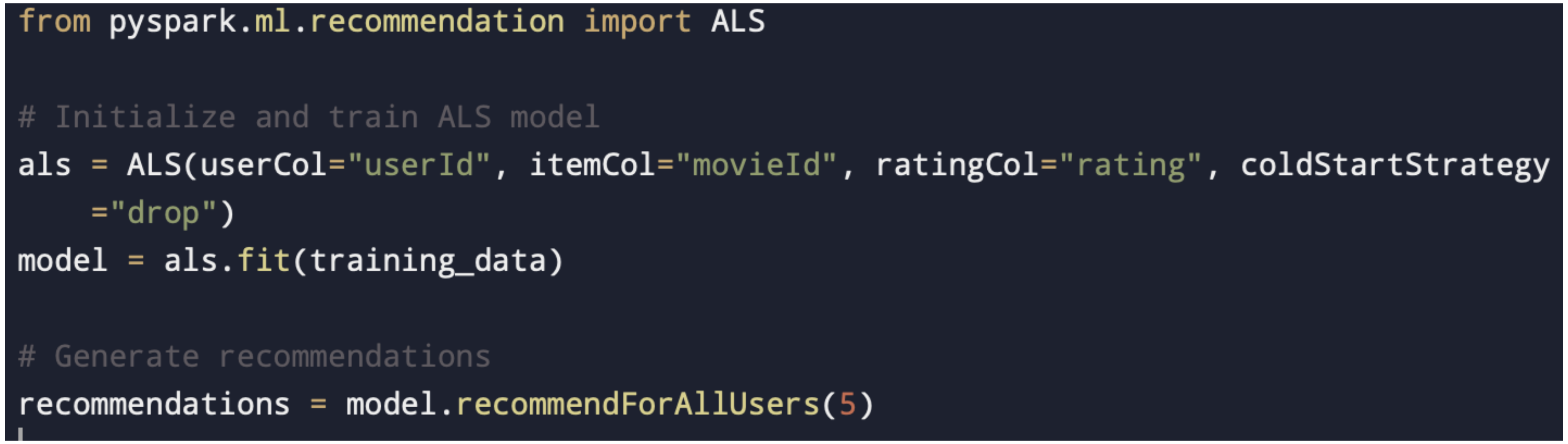
- ✓ K-Means
- ✓ Gaussian Mixture Model (GMM)
- ✓ Latent Dirichlet Allocation (LDA)



```python
from pyspark.ml.clustering import KMeans

# Initialize and train KMeans model
kmeans = KMeans(k=3, featuresCol="features")
model = kmeans.fit(training_data)

# Predict cluster assignments
clusters = model.transform(test_data)
```

MLlib – Recommendation Engine

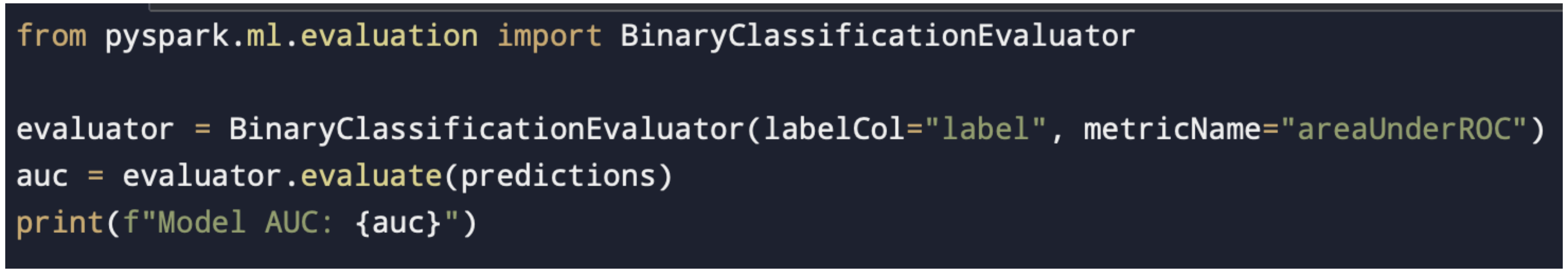
- MLLib provides collaborative filtering using **ALS (Alternating Least Squares)** for **recommendation engines**.



```python
from pyspark.ml.recommendation import ALS

# Initialize and train ALS model
als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy
    ="drop")
model = als.fit(training_data)

# Generate recommendations
recommendations = model.recommendForAllUsers(5)
```

Mllib - Model Evaluation and Hyperparameter Tuning

✅ **Evaluation Metrics** – Accuracy, Precision, Recall, RMSE, MSE.
✅ **Hyperparameter Tuning** – Cross-validation and Grid Search using TrainValidationSplit.



```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator(labelCol="label", metricName="areaUnderROC")
auc = evaluator.evaluate(predictions)
print(f"Model AUC: {auc}")
```

Spark Core

- **Apache Spark Core** is the fundamental component of Apache Spark. It provides the **basic functionalities** needed for distributed computing, including:
  - **Task Scheduling**
  - **Memory Management**
  - **Fault Tolerance**
  - **Distributed Data Processing**
- Everything in Spark (e.g., Spark SQL, MLlib, GraphX, and Streaming) is built on top of **Spark Core**.

Feature of Spark Core

1. Fault Tolerance
    - If a node fails, Spark can **recompute lost partitions** using **RDD lineage**.
    - Uses **DAG (Directed Acyclic Graph)** to track transformations and actions.
2. Distributed Task Scheduling
    - Spark automatically distributes computations across multiple nodes.
    - Uses DAG Scheduler and Task Scheduler to optimize execution.
3. In-Memory Processing
    - Spark keeps data in-memory (RAM) instead of writing to disk (HDFS).
    - This makes it 100x faster than Hadoop MapReduce.