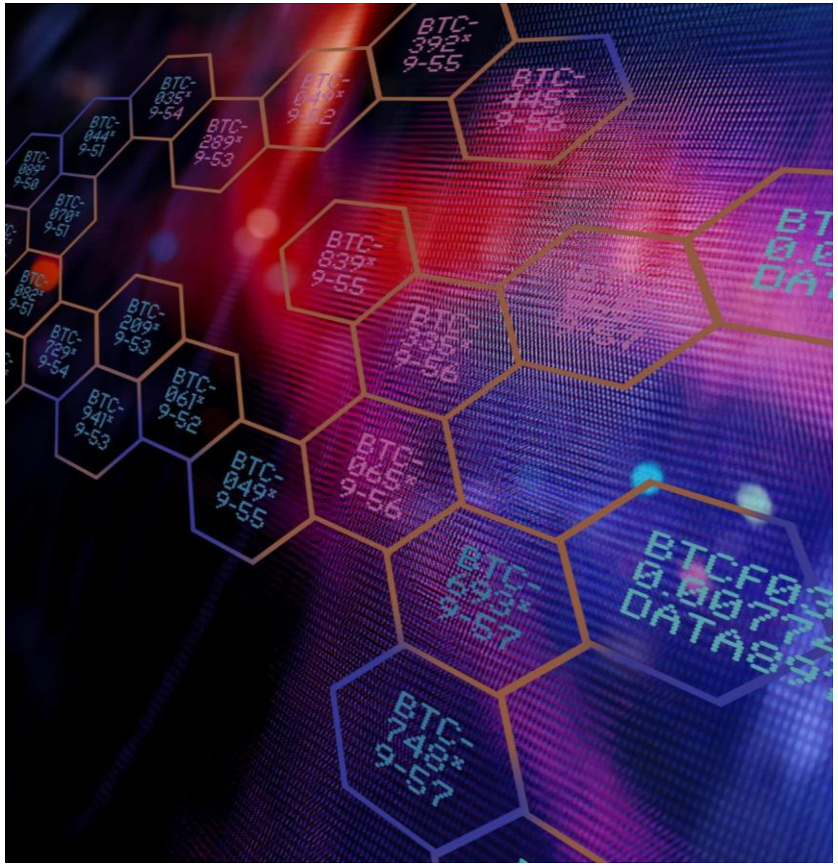
Apache Spark and PySpark

Understanding architecture
and big data processing tools

MC ID: 7124506 | PARVEEN KR | PARVEEN.R@HOTMAIL.COM

Agenda Overview

- Introduction to Apache Spark and Big Data
- Core Components and Execution Flow in Spark
- Data Abstractions: RDDs, DataFrames, and Datasets
- Transformations, Actions, and Lazy Evaluation
- Working with PySpark: Essentials and Interoperability
- PySpark vs Pandas vs Dask: When to Use Each
- Schema Management: Inference and Explicit Schemas
- Reading and Writing Data in PySpark
- Creating DataFrames from Multiple Sources
- Essential DataFrame Operations in PySpark



Introduction to Apache Spark and Big Data

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Understanding Apache Spark and Its Role in Big Data

Apache Spark Defined

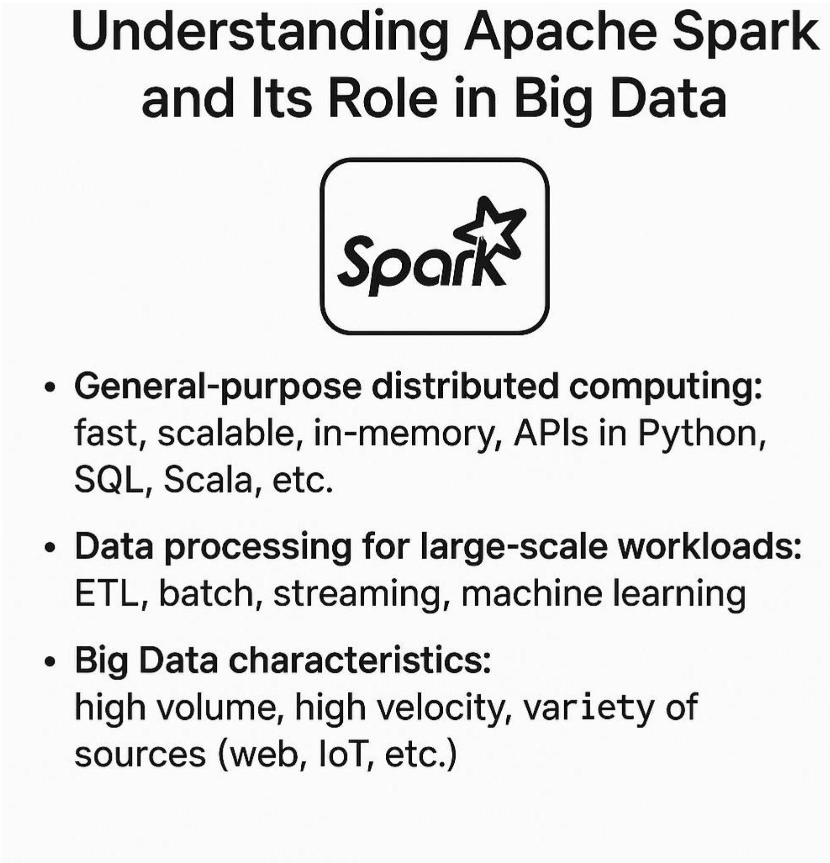
Spark is a distributed compute engine designed for processing large-scale data efficiently across multiple machines.

Big Data Importance

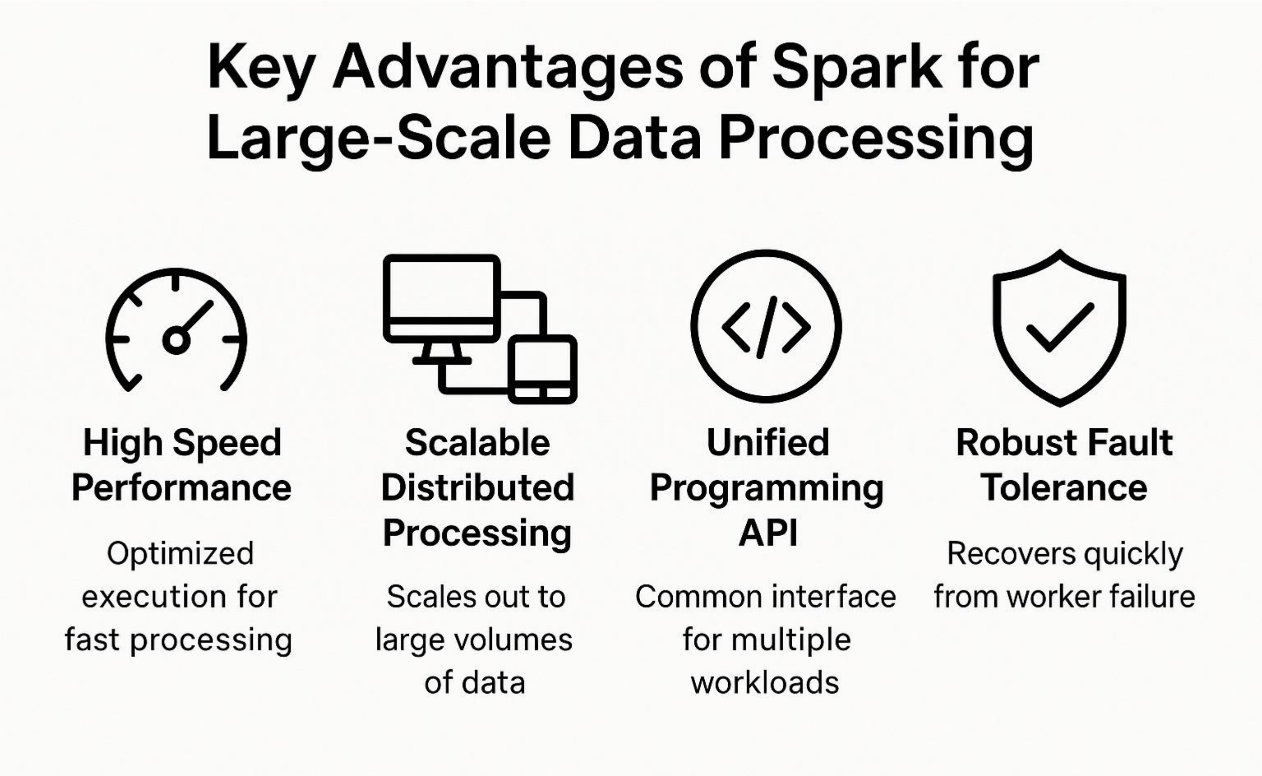
Big Data requires powerful tools like Spark to handle vast volumes of data with speed and scalability.

Cluster Computing

Spark operates across clusters, enabling parallel data processing to accelerate analytics and computations.



Key Advantages of Spark for Large-Scale Data Processing



High Speed Performance

Spark uses in-memory analytics and an optimized engine for fast data processing.

Scalable Distributed Processing

Spark scales across many executors to handle terabyte to petabyte scale datasets efficiently.

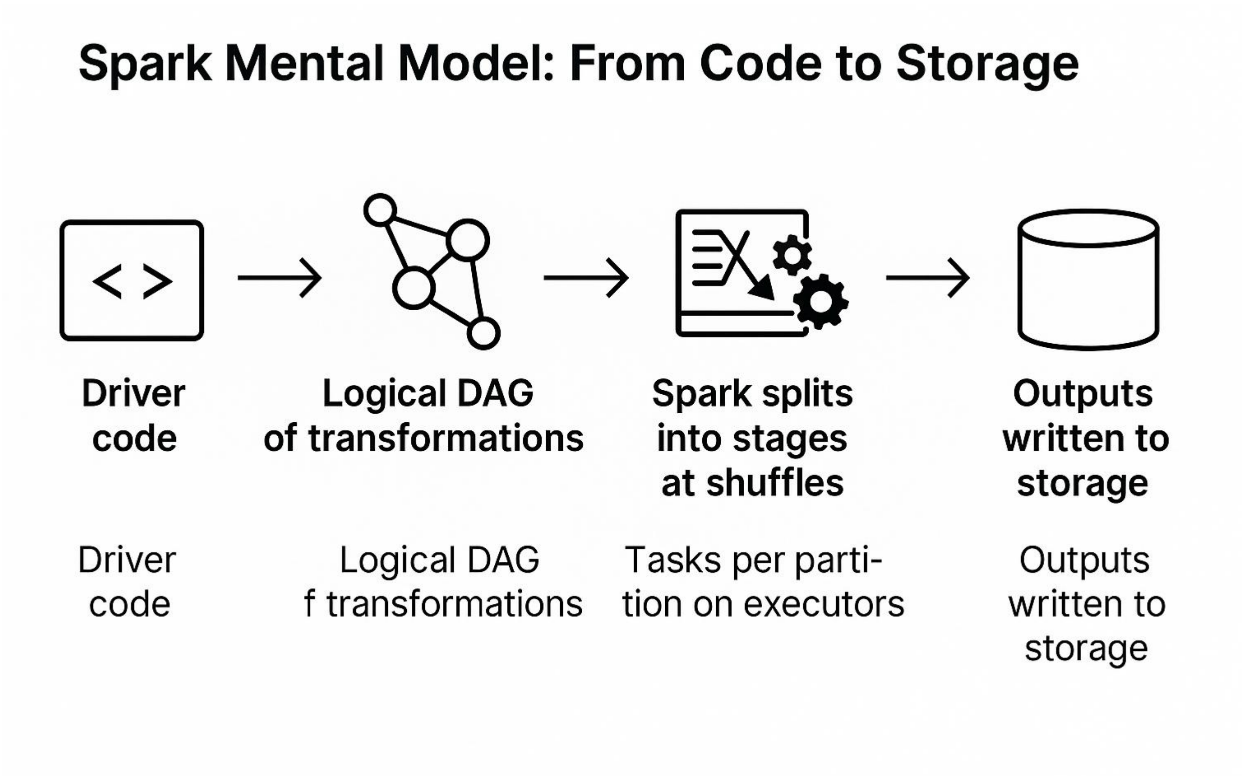
Unified Programming APIs

Provides unified APIs for SQL, streaming, machine learning, and graph processing.

Robust Fault Tolerance

Maintains lineage for RDDs and DataFrames enabling automatic task re-execution on failure.

Spark's Execution Model and Mental Model Overview



Driver and Logical DAG

The driver program generates a logical Directed Acyclic Graph of transformations from the user code.

Stages and Shuffle Boundaries

Spark divides the DAG into stages at shuffle boundaries to organize task execution efficiently.

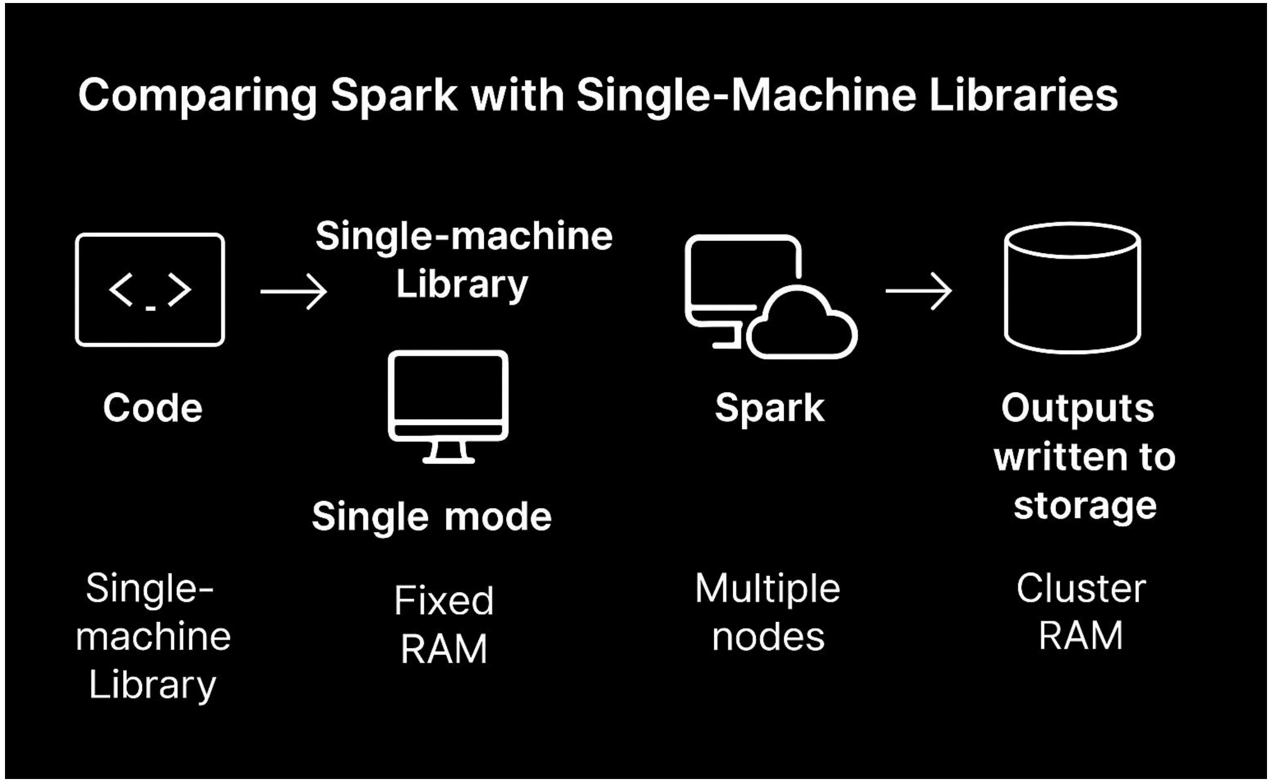
Executors and Task Execution

Tasks are executed per data partition on executors distributed across the cluster.

Output Storage

Task outputs are written back to reliable storage systems after execution completes.

Comparing Spark with Single-Machine Libraries



Single-Machine Libraries

Single-machine libraries like pandas process data on one computer, limiting scalability and memory size.

Apache Spark

Spark distributes data processing across multiple machines, enabling scalable and faster computations on large datasets.

Core Components and Execution Flow in Spark

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Main Components: Driver, Cluster Manager, Executors, and Partitions

Driver Role

The driver runs the main application, manages SparkSession, and builds the DAG for execution.

Cluster Manager Function

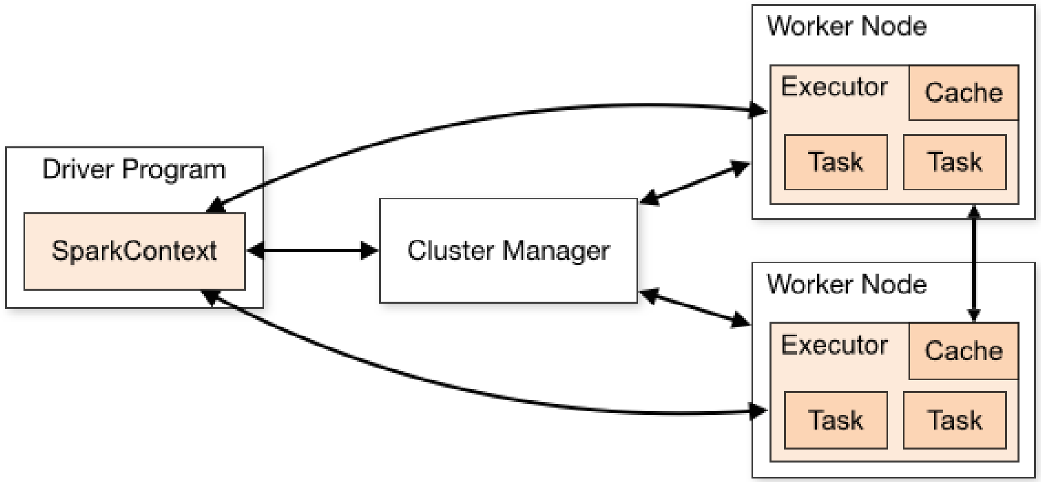
Cluster managers like YARN or Kubernetes allocate executors to run distributed tasks efficiently.

Executors Execution

Executors are JVM processes that run tasks, manage memory, slots, and cache during execution.

Partitions and Parallelism

Partitions are units of parallelism where each task processes one partition of data simultaneously.



Stages, Tasks, and Transformations in Spark

Narrow Transformations

Narrow transformations like map, filter, and select do not cause shuffles and stay within the same stage.

Examples: map, filter, selectData stays in the same partition, no data shuffle happens between nodes.Result: Runs within the same stage (faster and cheaper).

Wide Transformations

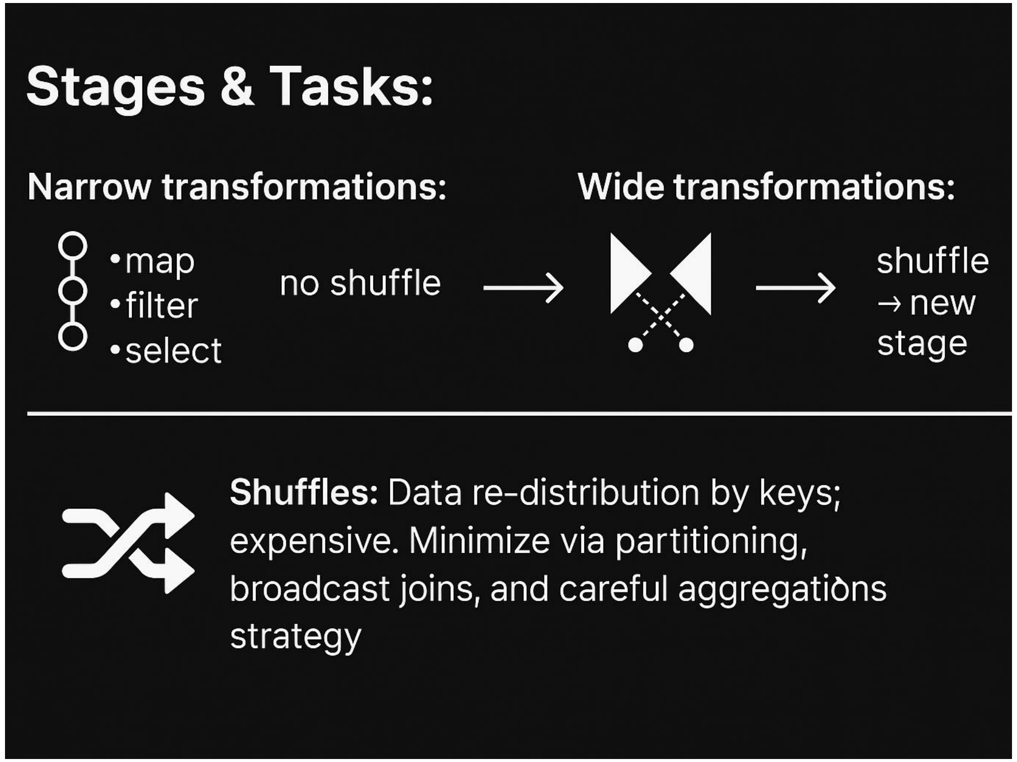
Wide transformations such as groupBy, join, and distinct cause shuffles and trigger new stages in Spark.

Examples: groupBy, join, distinctData must be re-distributed across partitions (shuffle) so that rows with the same key end up together.Result: Triggers a new stage (more expensive).

Shuffle Optimization

Shuffles are expensive due to data redistribution; minimize using partitioning, broadcast joins, and careful key selection.

Definition: Data re-distribution between nodes/partitions based on keys.Costly because it involves network I/O and disk I/O.Optimization strategies:Use partitioning wisely.Apply broadcast joins when one dataset is small.Aggregate carefully to reduce shuffle size.



MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Optimizing Execution Plans and Shuffle Management

Using df.explain

Use df.explain("formatted") to visualize detailed execution plans for dataframes.

Identifying Exchange Nodes

Exchange nodes indicate shuffle operations that may impact performance in data processing.

Managing Shuffles

Careful management of shuffle operations improves execution efficiency and resource utilization.

Data Abstractions: RDDs, DataFrames, and Datasets

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Comparing RDDs, DataFrames, and Datasets

RDD Characteristics

RDDs offer low-level, JVM-typed control with weak optimization, ideal for custom partitioning and unstructured data.

DataFrame Features

DataFrames represent distributed tables with schemas and use Catalyst optimization, making them the primary PySpark API.

Dataset Overview

Datasets are typed DataFrames in Scala/Java, not available as typed in Python, where DataFrame equals Dataset of Row.

Use Case Comparison

RDDs are ideal for custom log parsing, whereas DataFrames excel in analyzing structured sales data efficiently.

Best Practices for Choosing Data Abstractions

Primary Data Abstraction

DataFrames are preferred for most data workloads due to their optimizations and ease of use.

Use of RDDs

Resilient Distributed Datasets (RDDs) should be used only when DataFrames cannot achieve the desired functionality.

Transformations, Actions, and Lazy Evaluation

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Understanding Transformations and Actions in Spark

Lazy Transformations

Transformations are lazy operations that create new DataFrames or RDDs without executing jobs immediately.

Triggering Actions

Actions trigger execution and return results or write data, starting the computation in Spark.

Examples of Transformations

Common transformations include select, filter, withColumn, map, groupBy, and join operations.

Examples of Actions

Typical actions are count, collect, show, take, foreach, and write operations that trigger computation.

Transformations

map	join	union	distinct	repartition
mapPartitions	flatMap	intersection	pipe	coalesce
cartesian	cogroup	filter	sample	
sortByKey	groupByKey	reduceByKey	aggregateByKey	
mapPartitionswithIndex		repartitionAndSortWithinPartitions		

Actions

reduce	take	collect	takeSample	count
takeOrdered	countByKey	first	foreach	saveAsTextFile
saveAsSequenceFile		saveAsObjectFile		

Benefits of Lazy Evaluation and Common Pitfalls

Advantages of Lazy Evaluation

Lazy evaluation allows Spark to optimize query plans before execution, improving efficiency and performance.

Optimization Techniques

Techniques like predicate pushdown, column pruning, and stage fusion help reduce data processed and speed up jobs.

Common Pitfall: collect() Usage

Calling collect() on large datasets can cause driver out-of-memory errors; safer alternatives include show, take, or writing data out.

Working with PySpark: Essentials and Interoperability

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

PySpark Entry Points and Function Types

SparkSession Entry Point

SparkSession is the main entry point in PySpark, created using builder pattern for managing Spark application.

Built-in SQL Functions

Built-in SQL functions are preferred over UDFs for better optimization and efficient JVM-level execution.

Pandas UDFs for Python Logic

Use Pandas UDFs to perform Python-heavy logic with vectorized operations and Apache Arrow for speed.

Pandas-on-Spark and Data Interoperability

Pandas-like API on Spark

pandas-on-Spark provides a familiar Pandas API powered by Spark, enabling easy migration and scalability for large datasets.

Data Interoperability Methods

Supports interoperability between Spark DataFrames and Pandas DataFrames using df.toPandas() and spark.createDataFrame().

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

PySpark vs Pandas vs Dask: When to Use Each

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Feature Comparison Table: PySpark, Pandas, and Dask

FEATURE	PYSPARK	PANDAS	DASK
Scale	Cluster-scale	Single-machine memory	Scales across cores/nodes
API	SQL/DataFrame + Spark SQL	Pythonic dataframe	Pandas-like
Best for	Huge datasets; pipelines; BI/ETL	Small–medium, EDA	Medium/large; Python ecosystem
I/O	Parquet/Delta/Kafka/JDBC	Files, DB APIs	Similar to pandas + distributed

Guidance for Selecting the Right Tool

Data Size Consideration

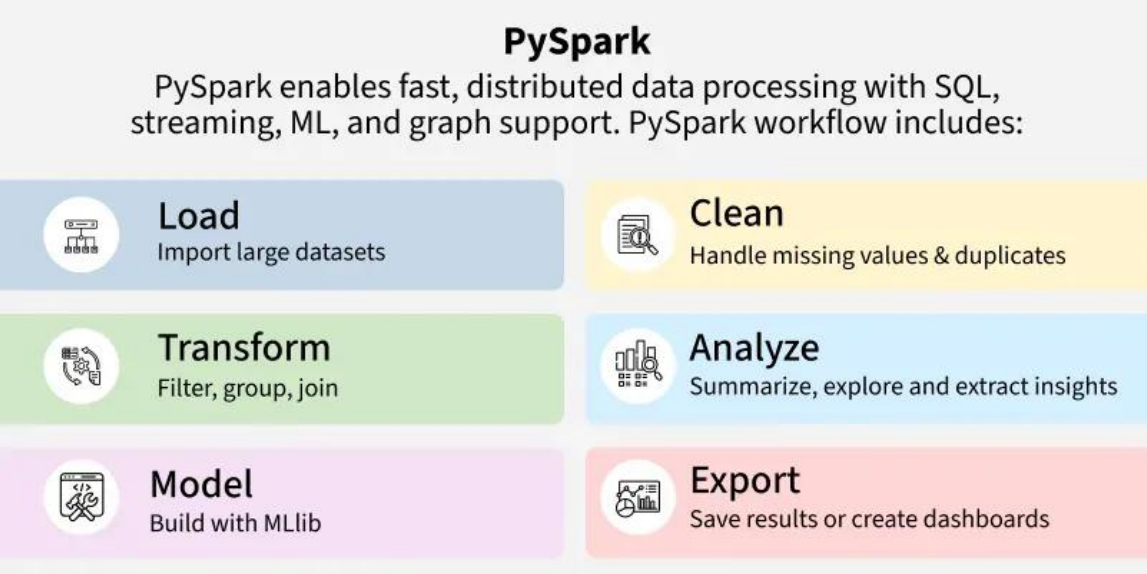
Use PySpark when data exceeds single machine RAM capacity for efficient processing.

Pipeline Scheduling Needs

PySpark supports scheduled data pipelines for automated and timely data processing.

Streaming Data Handling

PySpark enables real-time streaming data processing for immediate insights.



Schema Management: Inference and Explicit Schemas

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Schema Inference vs Explicit Schema: Pros and Cons

Inference (inferSchema=True): convenient, but can be wrong (e.g., numeric vs string; date parsing). Good for prototyping.
Explicit schema (StructType): safer, faster, and consistent across runs.

Schema Inference Convenience

Schema inference automatically detects data types, providing convenience for quick prototyping and development.

Inference Limitations

Inference can be inaccurate, confusing numeric and string types or misparsing dates, leading to data errors.

Explicit Schema Advantages

Explicit schemas ensure safety, consistency, and faster processing by defining exact data types upfront.

Defining Explicit Schemas: Code Examples

```
from pyspark.sql.types import StructType, StructField,
StringType, IntegerType, DoubleType, TimestampType
sales_schema = StructType([
StructField("SalesOrderNumber", StringType(), False),
StructField("SalesOrderLineNumber", IntegerType(), False),
StructField("OrderDate", TimestampType(), True),
StructField("CustomerName", StringType(), True),
StructField("EmailAddress", StringType(), True),
StructField("Item", StringType(), True),
StructField("Quantity", IntegerType(), True),
StructField("UnitPrice", DoubleType(), True),
StructField("TaxAmount", DoubleType(), True)
])
df = (spark.read
.option("header", True)
.schema(sales_schema)
.csv("dbfs:/FileStore/data/sales.csv"))
```

Explicit Schema Definition

Use StructType and StructField to define explicit schemas for structured data in PySpark.

Data Types Specification

Explicitly specify data types such as StringType, IntegerType, and DoubleType for each column.

Reading CSV with Schema

Load CSV data using Spark read option with the defined explicit schema for better data consistency.

Using DDL Strings for Quick Schema Definitions

```
ddl = """
SalesOrderNumber STRING, SalesOrderLineNumber INT,
OrderDate TIMESTAMP,
CustomerName STRING, EmailAddress STRING, Item STRING,
Quantity INT, UnitPrice DOUBLE, TaxAmount DOUBLE
"""
df = spark.read.option("header",
True).schema(ddl).csv("dbfs:/FileStore/data/sales.csv")
```

DDL String Usage

DDL strings allow quick and readable schema definitions for data processing frameworks like Spark.

Schema Definition Benefits

Using DDL strings simplifies schema creation and helps avoid errors in defining data types manually.

Reading CSV with Schema

Applying schema to CSV reading improves performance and ensures data integrity in Spark applications.

Reading and Writing Data in PySpark

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Reading and Writing CSV, JSON, Parquet, ORC, Avro, and Delta

CSV (read):
df = (spark.read.format("csv")
.option("header", True).option("inferSchema", True)
.load("dbfs:/FileStore/data/sales.csv"))
CSV (write):
(df.write.format("csv")
.mode("overwrite")
.option("header", True)
.save("dbfs:/tmp/out/csv/sales/"))

JSON:
spark.read.json("dbfs:/FileStore/data/events/.json")*
Parquet (recommended for analytics):
df_parq = spark.read.parquet("dbfs:/data/in/parquet/")
df.write.mode("overwrite").parquet("dbfs:/data/out/parquet/sales/")

ORC:
df_orc = spark.read.orc("dbfs:/data/in/orc/")
Avro (built-in on Databricks/Spark 2.4+):
df_avro = spark.read.format("avro").load("dbfs:/data/in/avro/")

Delta (preferred for lakehouse tables):
write
df.write.format("delta").mode("overwrite").saveAsTable("bronze.sales")
read
spark.read.table("bronze.sales")

CSV Format Handling

PySpark reads and writes CSV files with options to include headers and infer schemas automatically.

JSON and Avro Support

Spark supports JSON and Avro formats natively, enabling efficient handling of semi-structured data.

Efficient Analytical Formats

Parquet and ORC are optimized formats recommended for analytics with efficient compression and performance.

Delta Lake Integration

Delta format is preferred for lakehouse tables supporting ACID transactions and scalable data management.

Advanced Options for Data I/O

.option("compression", "snappy"), .partitionBy("OrderDate"), .mode("append")

Compression Option

Using compression like Snappy optimizes storage and speeds up data processing.

Partitioning Data

Partitioning by fields such as OrderDate improves query efficiency and data organization.

Write Mode Append

Append mode allows adding new data without overwriting existing datasets.

Creating DataFrames from Multiple Sources

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

From Python Lists, RDDs, and Relational Databases

```
from Python lists / dicts
rows = [
("SO1001", 1, "2025-08-01 10:00:00", "Alice", "a@x.com", "Pencil", 5,
1.0, 0.05),
("SO1001", 2, "2025-08-01 10:00:00", "Alice", "a@x.com", "Notebook", 1,
3.0, 0.18)
]
df_small = spark.createDataFrame(rows, schema=ddl)
From RDDs
rdd = spark.sparkContext.parallelize(rows, 2)
df_rdd = spark.createDataFrame(rdd, schema=sales_schema)
From relational DB (JDBC)

jdbc_url =
"jdbc:sqlserver://<server>.database.windows.net:1433;databaseNam
e=sales"
props =
{"user":"<u>","password":"<p>","driver":"com.microsoft.sqlserver.jdbc.
SQLServerDriver"}
df_jdbc = (spark.read.format("jdbc")
.option("url", jdbc_url)
.option("dbtable", "dbo.Orders")
.options(**props).load())
```

DataFrames from Python Lists

Create Spark DataFrames directly from Python lists or dictionaries using a specified schema.

DataFrames from RDDs

Spark DataFrames can be created from RDDs, enabling distributed data processing and transformation.

DataFrames from Relational DB

Load data into Spark DataFrames from relational databases using JDBC connections and configuration properties.

Integrating with Google BigQuery and Snowflake

```
from Google BigQuery (connector required)
bq = (spark.read.format("bigquery")
.option("table", "project.dataset.table")
.load())
From Snowflake (spark-snowflake connector required)
sfOptions = {
"sfURL": "<account>.snowflakecomputing.com",
"sfUser": "<user>",
"sfPassword": "<pwd>",
"sfDatabase": "<DB>",
"sfSchema": "<SCHEMA>",
"sfWarehouse": "<WH>"
}
df_sf = (spark.read.format("snowflake")
.options(**sfOptions)
.option("dbtable","ORDERS")
.load())
```

Google BigQuery Integration

Use Spark BigQuery connector to load data from Google BigQuery for analytics and processing.

Snowflake Integration Setup

Configure spark-snowflake connector with credentials to access Snowflake tables in Spark.

Connector Requirements and Security Considerations

External connectors need the proper JARs/cluster libraries and secrets in a secure store (Databricks secret scopes / Key Vault).

External Connector Essentials

External connectors must include proper JAR files and cluster libraries to function correctly.

Secret Management

Secrets must be stored securely using secret scopes or key vaults to protect sensitive information.

Essential DataFrame Operations in PySpark

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Selecting, Renaming, and Filtering Data

Selecting & renaming

```
from pyspark.sql.functions import col, expr
df_sel = df.select("SalesOrderNumber",
"OrderDate", col("Quantity").alias("qty"))
```

Filtering

```
df_f = df.filter((col("Quantity") > 0) &
(col("UnitPrice") > 0))
```

Selecting Data Columns

Select specific columns from a DataFrame to focus on relevant data for analysis or processing.

Renaming Columns

Rename DataFrame columns for clarity or to meet schema requirements during data transformation.

Filtering Data Rows

Filter rows based on conditions to exclude unwanted or invalid data from analysis.

Derived Columns and Handling Nulls

*df2 = df.withColumn("Revenue", expr("Quantity * UnitPrice"))*
Handling nulls
df_na = df.fillna({"EmailAddress": "unknown@example.com"})
 # or
df.dropna(subset=["SalesOrderNumber", "Quantity"])

Creating Derived Columns

Derived columns are calculated from existing data using expressions or transformations for enhanced analysis.

Handling Null Values

Null values are managed by filling missing data or dropping rows to ensure data quality in analysis.

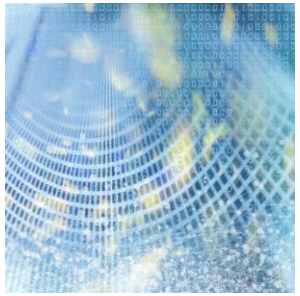
Aggregations, Joins, and Broadcast Joins

Data Aggregations

PySpark aggregation groups data and computes sums and distinct counts for insightful analysis.

Standard Joins

Joins combine datasets on common keys, enabling richer data analysis and insights integration.



Broadcast Joins

Broadcast joins optimize join performance by broadcasting small tables to all nodes in a cluster.

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Deduplication, Unions, and Window Functions

df_dedup = df.dropDuplicates(["SalesOrderNumber", "SalesOrderLineNumber"])
Unions
df_union = df.unionByName(df2, allowMissingColumns=True)
Window functions
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
w = Window.partitionBy("CustomerName").orderBy(col("OrderDate").desc())
df_ranked = df.withColumn("rn", row_number().over(w))

Deduplication of Data

Removing duplicate records ensures data accuracy by keeping unique entries based on key columns.

Union of DataFrames

Combining datasets by aligning columns, allowing missing columns for flexible data merging.

Window Functions for Ranking

Using window functions to rank records within partitions, ordered by specific criteria like date.

Repartitioning, Caching, and Performance Debugging

df_rep = df.repartition(24, "OrderDate")   # more parallelism
df_small = df.coalesce(1)   # fewer files (avoid on big data)
Cache / persist
df_cached = df.cache()
df_cached.count()   # materialize
Explain (debug & perf)
df.explain("formatted")

Repartitioning and Coalesce

Repartitioning increases parallelism by redistributing data across partitions, improving processing speed. Coalesce reduces the number of partitions to optimize file handling, especially on smaller datasets.

Caching and Persisting Data

Caching stores data in memory to accelerate repeated access and computation. Persisting materializes cached data to ensure it is retained during processing.

Performance Debugging

Using explain methods helps analyze query plans and diagnose performance issues to optimize data processing workflows.

Conclusion

Powerful Big Data Frameworks

Apache Spark and PySpark offer robust frameworks enabling scalable and distributed big data processing across clusters.

Architectural Understanding

Grasping Spark's architecture and data abstractions is crucial for optimizing performance and resource management.

Effective Data Operations

Practical knowledge of Spark operations enables efficient data engineering and advanced analytics tasks.