Introduction:

* Python + Spark = Pyspark
* PySpark encompass a wide range of modules and functionalities designed for processing data.
* PySpark is a Python API for Apache Spark.
* Apache Spark is one of the most popular cluster computing frameworks.

## Core Components of Pyspark:

1. SparkSession: The main entry point for PySpark. It provides access to all of the other components of PySpark.
2. DataFrames: A distributed data structure that represents a tabular dataset.
3. SQL: A declarative language for querying data.
4. MLlib: A machine learning library for PySpark.
5. GraphX: A graph processing library for PySpark.
6. SparkR: A R API for Spark.

Spark Session:

Apache Spark 1 – Spark Context is the entry point to interact with SQL/Hive

Apache Spark 2 – Spark Session is the entry point to interact with SQL/Hive

It will automatically initialize and include Spark Context in spark session as well.

# PYSPARK FUNCTION:

In PySpark, Spark functions refer to a set of built-in functions that can be applied to DataFrame or columns to perform various operations during data processing. These functions are part of the `pyspark.sql.functions` module.

1. Aggregate Function:

* Count () - Returns the number of rows in a DataFrame.
* Sum (): Computes the sum of values in a column.
* Avg (): Computes the average of values in a column.
* Min () and max (): Find the minimum or maximum value in a column.
* First () and last (): Returns the first or last value in a column.

1. Mathematical Functions:

* Abs (): Computes the absolute value of a numeric column.
* Sqrt (): Computes the square root of a numeric column.
* Round (): Rounds a numeric column to the specified number of decimal places.

1. String Functions:

* Concat (): Concatenates multiple string columns.
* Length (): Computes the length of a string column.
* Substring (): Extracts a substring from a string column.
* Trim (), ltrim (), and rtrim (): Trim whitespace from strings.

1. Date and Time Functions:

* Current \_date () : Returns the current date.
* Current \_timestamp (): Returns the current timestamp.
* Date \_add (), date\_sub (): Add or subtract days from a date column.
* Datediff (): Computes the difference between two dates.

1. Conditional Functions:

* When (): Allows for conditional expressions.
* Otherwise (): Specifies a default value when used with when ().

1. Statistical Functions:

* Corr (): Computes the correlation between two columns.
* Covar\_pop () and covar\_samp(): Compute population and sample covariance.
* Var\_pop () and var\_samp(): Compute population and sample variance.

1. Window Functions:

* Row \_number (): Assigns a unique number to each row within a partition.
* Rank () and dense \_rank (): Assign ranks to rows based on a specified column.

1. Type Conversion Functions:

* Cast (): Converts a column to a specified data type.

1. Collection Functions:

* Array (): Creates an array column.
* Map (): Creates a map column.
* Explode (): Explodes an array or map into multiple rows.

=====================================================================

Abstractions for Data Processing (Apache Spark 2.X):

1. RDD
2. Data Frame (DF)
3. Dataset

Optimization Techniques:

Performance:

1. Data Caching:

Spark provides its own caching mechanisms, there are 2 types of mechanisms.

1. Cache
2. Persist

* Cache (): Caching mechanism is to store an RDD or DataFrame in memory for faster access. It keeps the data in deserialized form in the Java Virtual Machine (JVM) of the Spark executor nodes.

#Command: rdd.cache()

dataframe.cache()

* Persistence (): Persistence is a more flexible version of caching that allows you to store the data not only in memory but also on disk or a combination of both.

Storage Parameter - Memory/Disk/both

#Command: rdd.persist(storage\_level)

dataframe.persist(storage\_level)

1. Logical Plan Optimization:

Logical plan optimizations in PySpark involve transformations and improvements to the logical representation of the query plan before its physical execution. (i.e., executes at compile time but not at run time)

* **Predicate Pushdown:**

Predicate pushdown involves optimizing data retrieval by pushing filter conditions closer to the data source.

Example:

* + A large Parquet file Without predicate pushdown, the entire file is read, and then filtering is applied. With predicate pushdown, the filter is pushed to the Parquet reader, reducing the amount of data read.

# Without Predicate Pushdown

df = spark.read.parquet("data.parquet")

result = df.filter(df["column"] > 10).groupBy("groupColumn").agg({"aggColumn": "sum"})

# With Predicate Pushdown

df = spark.read.parquet("data.parquet")

filtered\_df = df.filter(df["column"] > 10)

result = filtered\_df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Constant Folding:

Constant folding involves evaluating constant expressions at compile-time, optimizing the query plan.

Example:

* + Consider adding a constant value to a column. Without constant folding, the addition operation occurs during execution. With constant folding, the addition is computed at compile-time.

# Without Constant Folding

df = spark.read.parquet("data.parquet")

result = df.withColumn("newColumn", df["column"] + 5)

# With Constant Folding

df = spark.read.parquet("data.parquet")

result = df.withColumn("newColumn", lit(5) + df["column"])

* Boolean Expression Simplification:

Simplifying Boolean expressions involves reducing complex conditions for improved query plan readability.

Example:

* + Simplify logical conditions to enhance readability. For example, combining multiple conditions into separate filter operations.

# Without Boolean Expression Simplification

df = spark.read.parquet("data.parquet")

result = df.filter((df["column1"] > 10) & (df["column2"] < 5))

# With Boolean Expression Simplification

df = spark.read.parquet("data.parquet")

result = df.filter(df["column1"] > 10).filter(df["column2"] < 5)

* Rule Based Transformations:

Rule-based transformations apply predefined rules to the logical plan to optimize specific patterns.

Example:

* + Convert a sequence of groupBy followed by agg into a single agg operation for improved performance.

# Without Rule-Based Transformation

df = spark.read.parquet("data.parquet")

result = df.groupBy("groupColumn").agg({"aggColumn1": "sum", "aggColumn2": "avg"})

# With Rule-Based Transformation

df = spark.read.parquet("data.parquet")

result = df.agg({"aggColumn1": "sum", "aggColumn2": "avg"}).groupBy("groupColumn")

1. Physical Plan Optimizations:

Physical Plan Optimizations in PySpark focus on improving the execution efficiency of data processing tasks.

* Whole Stage Code generation:

Whole-stage code generation involves collapsing a sequence of operations into a single function, reducing the overhead of function calls.

Example:

* + Imagine performing multiple DataFrame transformations. Without whole-stage code generation, each transformation incurs separate function calls. With it, the transformations are compiled into a single function for improved execution speed.

# Without Whole-Stage Code Generation

df = spark.read.parquet("data.parquet")

result = df.filter(df["column"]> 10).groupBy("groupColumn").agg({"aggColumn": "sum"})

# With Whole-Stage Code Generation

df = spark.read.parquet("data.parquet")

filtered\_df = df.filter(df["column"] > 10)

result = filtered\_df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Tungsten Optimizations:

Tungsten optimizations include low-level enhancements to Spark's execution engine, improving memory management and CPU efficiency.

Example:

* + Tungsten optimizations are applied automatically, providing benefits such as improved memory layouts and binary processing. Users don't need to explicitly enable Tungsten; it is part of Spark's underlying architecture.

# Tungsten optimizations are applied automatically in the Spark execution engine

df = spark.read.parquet("data.parquet")

result = df.filter(df["column"] > 10).groupBy("groupColumn").agg({"aggColumn": "sum"})

* Data Skew Handling:

Data skew handling addresses situations where certain keys have significantly more data than others, causing performance imbalances.

Example:

* + When joining two DataFrames with skewed key distributions, PySpark may employ techniques like broadcast joins or dynamic partition pruning to handle the skew and improve performance.

# Data Skew Handling Example (Broadcast Join)

df1 = spark.read.parquet("data1.parquet")

df2 = spark.read.parquet("data2.parquet")

# Broadcast join to handle data skew

result = df1.join(broadcast(df2), "key")

* In Place Mutability:

In-place mutability involves modifying data structures in memory without creating new objects, reducing memory overhead.

Example:

* + When performing transformations on large DataFrames, in-place mutability helps minimize the memory footprint by reusing existing memory buffers.

# In-Place Mutability Example

df = spark.read.parquet("data.parquet")

# Performing transformations with in-place mutability

df = df.withColumn("newColumn", df["column"] \* 2)

* Filter & Projection Pushdown:

Filter and projection pushdown involves pushing filtering and projection operations closer to the data source, reducing the amount of data processed.

Example:

* + When reading data from a source like Parquet, pushdown operations can significantly reduce the amount of data read from disk.

# Filter and Projection Pushdown Example

df = spark.read.parquet("data.parquet")

# Pushing filter and projection closer to the data source

result = df.select("column1", "column2").filter(df["column3"] > 10)

1. Shuffling Optimizations:

Shuffling optimizations in PySpark are crucial for improving the performance of operations involving data redistribution across the cluster.

* Shuffle Algorithm Selection:

Shuffling involves redistributing data across the cluster, and selecting the appropriate shuffle algorithm can impact performance.

Example:

* + Depending on the characteristics of the data and the nature of the operation, choosing between Sort-Based Shuffle and Hash-Based Shuffle can optimize shuffling.

# Shuffle Algorithm Selection Example

df = spark.read.parquet("data.parquet")

# Configuring shuffle to use Sort-Based Shuffle

spark.conf.set("spark.shuffle.manager", "sort")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Broadcast Hash Joins:

Broadcast hash joins involve broadcasting small tables to all worker nodes, reducing the need for data shuffling.

Example:

* + When joining a small lookup table with a larger table, broadcasting the smaller table can significantly improve join performance.

# Broadcast Hash Join Example

df1 = spark.read.parquet("large\_data.parquet")

df2 = spark.read.parquet("small\_lookup\_table.parquet")

# Performing a broadcast hash join

result = df1.join(broadcast(df2), "joinColumn")

* Map Side Joins:

Map-side joins involve optimizing join operations by performing parts of the join on the mappers before shuffling.

Example:

* + In scenarios where data for the join condition is already co-located on the same partition, map-side joins can be more efficient than shuffling.

# Map-Side Join Example

df1 = spark.read.parquet("data1.parquet")

df2 = spark.read.parquet("data2.parquet")

# Performing a map-side join

result = df1.join(df2, "joinColumn", "mapside")

* Skew Join Handling:

Skew join handling addresses situations where certain keys have significantly more data than others, causing performance imbalances during joins.

Example:

* + When dealing with skewed data distributions in join operations, PySpark may employ techniques like dynamic partition pruning to handle the skew and improve performance.

# Skew Join Handling Example

df1 = spark.read.parquet("data1.parquet")

df2 = spark.read.parquet("data2.parquet")

# Handling skew with dynamic partition pruning

result = df1.join(df2, "joinColumn").filter(df1["skewedColumn"] == "specificValue")

* Adaptive Query Execution:

Adaptive query execution dynamically adjusts execution plans based on runtime statistics, improving performance.

Example:

* + When dealing with varying data distributions or conditions, adaptive query execution can adapt and optimize the execution plan during runtime.

# Adaptive Query Execution Example

df = spark.read.parquet("data.parquet")

# Enabling adaptive query execution

spark.conf.set("spark.sql.adaptive.enabled", "true")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})

1. Optimizing DF & SQL:

Optimizing DataFrame and SQL operations in PySpark is essential for achieving efficient and scalable data processing.

* Catalyst Optimizer:

Catalyst is PySpark's query optimizer that leverages a rule-based system to optimize DataFrame operations.

Example:

* + Catalyst optimizations automatically analyze and improve query plans, such as reordering operations for better performance.

# Catalyst Optimizations Example

df = spark.read.parquet("data.parquet")

result = df.filter(df["column"] > 10).groupBy("groupColumn").agg({"aggColumn": "sum"})

* DataFrame Transformations:

DataFrame transformations involve high-level operations like select, filter, and groupBy, optimized by Catalyst.

Example:

* + Performing transformations using DataFrame operations allows Catalyst to optimize the query plan automatically.

# DataFrame Transformations Example

df = spark.read.parquet("data.parquet")

result = df.select("column1", "column2").filter(df["column3"] > 10)

* UDF Optimizations:

PySpark supports User-Defined Functions (UDFs), and Catalyst optimizes the execution of these custom functions.

Example:

* + Applying UDFs on DataFrame columns is optimized by Catalyst, ensuring efficient execution.

# UDF Optimizations Example

from pyspark.sql.functions import udf

from pyspark.sql.types import IntegerType

square\_udf = udf(lambda x: x\*\*2, IntegerType())

df = spark.read.parquet("data.parquet")

result = df.withColumn("squaredColumn", square\_udf(df["column"]))

* Query Plan Visualization:

PySpark allows users to visualize the query plan, helping understand and optimize the execution flow.

Example:

* + Visualizing the query plan can reveal potential bottlenecks or areas for improvement in complex DataFrame operations.

# Query Plan Visualization Example

df = spark.read.parquet("data.parquet")

df.explain(extended=True)

* Broadcast Variables in SQL:

In SQL queries, using broadcast variables for small tables can optimize join operations.

Example:

* + When joining a large table with a small lookup table, broadcasting the smaller table can improve join performance.

# Broadcast Variables in SQL Example

df1 = spark.read.parquet("large\_data.parquet")

df2 = spark.read.parquet("small\_lookup\_table.parquet")

# Performing a broadcast join in SQL

spark.sql("SELECT \* FROM df1 JOIN BROADCAST(df2) ON df1.joinColumn = df2.joinColumn")

1. Memory Management Optimizations:

Efficient memory management is crucial for optimizing the performance of PySpark applications. By configuring memory parameters, leveraging Tungsten optimizations, and making informed choices about memory serialization and off-heap usage, users can achieve better memory efficiency in their PySpark workflows.

* Memory Fraction Configuration:

PySpark allows users to configure the memory fraction allocated to execution and storage.

Example:

* + Adjusting the memory fraction can optimize the balance between execution and storage memory based on the nature of the workload.

# Memory Fraction Configuration Example

from pyspark import SparkConf

from pyspark.sql import SparkSession

conf = SparkConf().set("spark.memory.fraction", "0.7")

spark = SparkSession.builder.config(conf=conf).getOrCreate()

df = spark.read.parquet("data.parquet")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Tungsten Memory Management:

Tungsten memory management optimizations include efficient memory layouts and binary processing.

Example:

* + Tungsten optimizations automatically enhance memory usage for DataFrame operations, improving CPU efficiency.

# Tungsten Memory Management Example

df = spark.read.parquet("data.parquet")

result = df.filter(df["column"] > 10).groupBy("groupColumn").agg({"aggColumn": "sum"})

* Memory Serialization:

PySpark provides options for controlling the serialization format used for caching and shuffling data.

Example:

* + Choosing an optimized serialization format can impact the efficiency of caching and shuffling operations.

# Memory Serialization Example

from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer").getOrCreate()

df = spark.read.parquet("data.parquet")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Off Heap Memory:

PySpark allows users to allocate off-heap memory for certain operations, reducing the impact on the Java Garbage Collector.

Example:

* + Off-heap memory usage can be beneficial for operations with large in-memory data.

# Off-Heap Memory Usage Example

from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.memory.offHeap.enabled", "true").config("spark.memory.offHeap.size", "1g").getOrCreate()

df = spark.read.parquet("data.parquet")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})

* Memory Monitoring & Tunning:

PySpark provides tools for monitoring and tuning memory usage during application execution.

Example:

* + Monitoring and tuning memory parameters based on the observed usage patterns can optimize the application's performance.

# Memory Monitoring and Tuning Example

from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.memory.useLegacyMode", "true").getOrCreate()

df = spark.read.parquet("data.parquet")

result = df.groupBy("groupColumn").agg({"aggColumn": "sum"})