ALL ABOUT DATA ENGINEERING

COMPREHENSIVE PYSPARK LESSONS, NOTES & INTERVIEW QUESTIONS & ANSWERS

All about Data Engineering

APACHE SPARK INTERVIEW QUESTIONS

ABHINANDAN PATRA

DATA ENGINEER

1. What is Apache Spark?

Apache Spark is an open-source, distributed computing system that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It is designed to process large-scale data efficiently.

2. Why Apache Spark?

Apache Spark is used because it is faster than traditional big data tools like Hadoop MapReduce due to its in-memory processing capabilities, supports multiple languages (Scala, Python, R, Java), provides libraries for various tasks (SQL, machine learning, graph processing, etc.), and has robust fault tolerance.

3. What are the components of the Apache Spark Ecosystem?

The main components are:

- **Spark Core**: The foundational engine for large-scale parallel and distributed data processing.
- **Spark SQL**: For structured data processing.
- **Spark Streaming**: For real-time data processing.
- **MLlib**: A library for scalable machine learning.
- **GraphX**: For graph and graph-parallel computation.

4. What is Spark Core?

Spark Core is the general execution engine for the Spark platform, responsible for tasks such as scheduling, distributing, and monitoring applications.

5. Which languages does Apache Spark support?

Apache Spark supports:

- Scala
- Python
- Java
- R
- SQL

6. How is Apache Spark better than Hadoop?

Spark is better in several ways, including faster processing due to in-memory computation, ease of use with APIs for various programming languages, flexibility with built-in libraries for diverse tasks, and a rich set of APIs for transformations and actions.

7. What are the different methods to run Spark over Apache Hadoop?

Spark can run on Hadoop in the following modes:

- **Standalone**: Using its cluster manager.
- **YARN**: Hadoop's cluster manager.
- **Mesos**: Another cluster manager.

8. What is SparkContext in Apache Spark?

`SparkContext` is the entry point for any Spark application. It acts as a connection to the Spark cluster, allowing Spark jobs to be executed.

9. What is SparkSession in Apache Spark?

`SparkSession` is the unified entry point to work with DataFrames, Datasets, and SQL in Apache Spark. It replaces `SQLContext` and `HiveContext`.

10. SparkSession vs. SparkContext in Apache Spark

`SparkSession` is a combination of `SQLContext`, `HiveContext`, and `SparkContext` to provide a single point of entry to interact with Spark.

11. What are the abstractions of Apache Spark?

The primary abstractions are:

- **RDD (Resilient Distributed Dataset)**
- **DataFrames**
- **Datasets**

12. How can we create RDD in Apache Spark?

RDDs can be created in three ways:

- **Parallelizing a collection** in your program.
- **Referencing a dataset** in an external storage system (e.g., HDFS, S3, etc.).
- **Transforming** an existing RDD.

13. Why is Spark RDD immutable?

RDDs are immutable to provide fault tolerance and support functional programming principles, allowing Spark to rebuild lost data from the lineage information.

14. Explain the term paired RDD in Apache Spark.

Paired RDDs are RDDs where each element is a pair (key-value). They are used for operations like aggregation, grouping, and joins.

15. How is RDD in Spark different from Distributed Storage Management?

RDD is an in-memory data structure optimized for processing, while Distributed Storage (like HDFS) focuses on data storage and retrieval.

16. Explain transformation and action in RDD in Apache Spark.

- **Transformation**: Lazy operations that define a new RDD without executing until an action is called (e.g., `map`, `filter`).
- **Action**: Triggers the execution of transformations (e.g., `count`, `collect`).

17. What are the types of Apache Spark transformations?

Transformations can be narrow (e.g., `map`, `filter`) or wide (e.g., `groupByKey`, `reduceByKey`).

18. Explain the RDD properties.

RDD properties include:

- **Immutability**: Once created, RDDs cannot be changed.
- **Partitioned**: Distributed across various nodes in the cluster.
- **Lazy evaluation**: Operations are computed when an action is called.
- **Fault tolerance**: Recomputed using lineage information.

19. What is a lineage graph in Apache Spark?

A lineage graph tracks the sequence of transformations that created an RDD, used for recomputing lost data due to node failures.

20. Explain the terms Spark Partitions and Partitioners.

- **Partitions**: Logical division of data in RDDs, physically stored across nodes.
- **Partitioner**: Determines how data is distributed across partitions (e.g., `HashPartitioner`, `RangePartitioner`).

21. By default, how many partitions are created in RDD in Apache Spark?

By default, Spark creates partitions based on the number of cores available or the input file's HDFS block size.

22. What is Spark DataFrames?

DataFrames are distributed collections of data organized into named columns, similar to tables in a relational database.

23. What are the benefits of DataFrame in Spark?

Benefits include optimizations (Catalyst query optimizer), improved performance, and easier manipulation using SQL-like syntax.

24. What is Spark Dataset?

A Dataset is a distributed collection of data that provides type safety and object-oriented programming interfaces.

25. What are the advantages of datasets in Spark?

Advantages include compile-time type safety, optimizations through Tungsten, and the ability to leverage JVM object serialization.

26. What is Directed Acyclic Graph (DAG) in Apache Spark?

A DAG in Spark represents a sequence of computations performed on data, where each node is an RDD and edges represent transformations. It's used to optimize execution plans.

27. What is the need for Spark DAG?

The DAG allows Spark to optimize execution by scheduling tasks efficiently, minimizing data shuffling, and managing dependencies.

28. What is the difference between DAG and Lineage?

- **DAG**: Represents the entire execution plan of a Spark application.
- **Lineage**: Tracks transformations on a particular RDD, useful for fault recovery.

29. What is the difference between Caching and Persistence in Apache Spark?

- **Caching**: Default storage level is in-memory (`MEMORY_ONLY`).
- **Persistence**: Allows choosing different storage levels (disk, memory, etc.) for storing RDDs.

30. What are the limitations of Apache Spark?

Limitations include high memory consumption, limited built-in libraries compared to Hadoop, not suitable for small data or real-time streaming without specific tools.

31. Different Running Modes of Apache Spark

Spark can run in:

- **Local mode**: Single machine.
- **Standalone mode**: Using its cluster manager.
- **YARN mode**: On Hadoop’s cluster manager.
- **Mesos mode**: On Mesos cluster manager.
- **Kubernetes mode**: On Kubernetes.

32. What are the different ways of representing data in Spark?

Data can be represented as:

- **RDDs (Resilient Distributed Datasets)**

- **DataFrames**
- **Datasets**

33. What is Write-Ahead Log (WAL) in Spark?

Write-Ahead Log is a fault-tolerance mechanism where every received data is first written to a log file (disk) before processing, ensuring no data loss.

34. Explain Catalyst Query Optimizer in Apache Spark.

Catalyst is Spark SQL's query optimizer that uses rule-based and cost-based optimization techniques to generate efficient execution plans.

35. What are shared variables in Apache Spark?

Shared variables are variables that can be used by tasks running on different nodes:

- **Broadcast variables**: Efficiently share read-only data across nodes.
- **Accumulators**: Used for aggregating information (e.g., sums) across tasks.

36. How does Apache Spark handle accumulated metadata?

Spark stores metadata like lineage information, partition data, and task details in the driver and worker nodes, managing it using its DAG scheduler.

37. What is Apache Spark's Machine Learning Library?

MLlib is Spark's scalable machine learning library, which provides algorithms and utilities for classification, regression, clustering, collaborative filtering, and more.

38. List commonly used Machine Learning Algorithms.

Common algorithms in Spark MLlib include:

- Linear Regression
- Logistic Regression
- Decision Trees
- Random Forests
- Gradient-Boosted Trees
- K-Means Clustering

39. What is the difference between DSM and RDD?

- **DSM (Distributed Storage Management)**: Focuses on data storage across clusters.
- **RDD (Resilient Distributed Dataset)**: Focuses on distributed data processing with fault tolerance.

40. List the advantage of Parquet file in Apache Spark.

Advantages of Parquet files:

- Columnar storage format, optimized for read-heavy workloads.
- Efficient compression and encoding schemes.
- Schema evolution support.

41. What is lazy evaluation in Spark?

Lazy evaluation defers execution until an action is performed, optimizing the execution plan by reducing redundant computations.

42. What are the benefits of Spark lazy evaluation?

Benefits include:

- Reducing the number of passes over data.
- Optimizing the computation process.
- Decreasing execution time.

43. How much faster is Apache Spark than Hadoop?

Apache Spark is generally up to 100x faster than Hadoop for in-memory processing and up to 10x faster for on-disk data.

44. What are the ways to launch Apache Spark over YARN?

Spark can be launched over YARN in:

- **Client mode**: Driver runs on the client machine.
- **Cluster mode**: Driver runs inside YARN cluster.

45. Explain various cluster managers in Apache Spark.

Spark supports:

- **Standalone Cluster Manager**: Default cluster manager.
- **Apache Mesos**: A general-purpose cluster manager.
- **Hadoop YARN**: A resource manager for Hadoop clusters.
- **Kubernetes**: For container orchestration.

46. What is Speculative Execution in Apache Spark?

Speculative execution is a mechanism to detect slow-running tasks and run duplicates on other nodes to speed up the process.

47. How can data transfer be minimized when working with Apache Spark?

Data transfer can be minimized by:

- Reducing shuffling and repartitioning.
- Using broadcast variables.
- Efficient data partitioning.

48. What are the cases where Apache Spark surpasses Hadoop?

Apache Spark outperforms Hadoop in scenarios involving iterative algorithms, in-memory computations, real-time analytics, and complex data processing workflows.

49. What is an action, and how does it process data in Apache Spark?

An action is an operation that triggers the execution of transformations (e.g., `count`, `collect`), performing computations and returning a result.

50. How is fault tolerance achieved in Apache Spark?

Fault tolerance is achieved through lineage information, allowing RDDs to be recomputed from scratch if a partition is lost.

51. What is the role of the Spark Driver in Spark applications?

The Spark Driver is responsible for converting the user's code into tasks, scheduling them on executors, and collecting the results.

52. What is a worker node in an Apache Spark cluster?

A worker node is a machine in a Spark cluster where the actual data processing tasks are executed.

53. Why is Transformation lazy in Spark?

Transformations are lazy to build an optimized execution plan (DAG) and to avoid unnecessary computation.

54. Can I run Apache Spark without Hadoop?

Yes, Spark can run independently using its built-in cluster manager or other managers like Mesos and Kubernetes.

55. Explain Accumulator in Spark.

An accumulator is a variable used for aggregating information across executors, like counters in MapReduce.

56. What is the role of the Driver program in a Spark Application?

The Driver program coordinates the execution of tasks, maintains the SparkContext, and communicates with the cluster manager.

57. How to identify that a given operation is a Transformation or Action in your program?

Transformations return RDDs (e.g., `map`, `filter`), while actions return non-RDD values (e.g., `collect`, `count`).

58. Name the two types of shared variables available in Apache Spark.

- **Broadcast Variables**
- **Accumulators**

59. What are the common faults of developers while using Apache Spark?

Common faults include:

- Inefficient data partitioning.
- Excessive shuffling and data movement.
- Inappropriate use of transformations and actions.
- Not leveraging caching and persistence properly.

60. By Default, how many partitions are created in RDD in Apache Spark?

The default number of partitions is based on the number of cores available in the cluster or the HDFS block size.

61. Why do we need compression, and what are the different compression formats supported?

Compression reduces the storage size of data and speeds up data transfer. Spark supports several compression formats:

- **Snappy**
- **Gzip**
- **Bzip2**
- **LZ4**
- **Zstandard (Zstd)**

62. Explain the filter transformation.

The `filter` transformation creates a new RDD by selecting only elements that satisfy a given predicate function.

63. How to start and stop Spark in the interactive shell?

To start Spark in the interactive shell:

- Use `spark-shell` for Scala or `pyspark` for Python. To stop Spark:
- Use `:quit` or `Ctrl + D` in the shell.

64. Explain the `sortByKey()` operation.

`sortByKey()` sorts an RDD of key-value pairs by the key in ascending or descending order.

65. Explain `distinct()`, `union()`, `intersection()`, and `subtract()` transformations in Spark.

- **distinct()**: Returns an RDD with duplicate elements removed.
- **union()**: Combines two RDDs into one.
- **intersection()**: Returns an RDD with elements common to both RDDs.
- **subtract()**: Returns an RDD with elements in one RDD but not in another.

66. Explain `foreach()` operation in Apache Spark.

`foreach()` applies a function to each element in the RDD, typically used for side effects like updating an external data store.

67. `groupByKey` vs `reduceByKey` in Apache Spark.

- **groupByKey**: Groups values by key and shuffles all data across the network, which can be less efficient.
- **reduceByKey**: Combines values for each key locally before shuffling, reducing network traffic.

68. Explain `mapPartitions()` and `mapPartitionsWithIndex()`.

- **mapPartitions()**: Applies a function to each partition of the RDD.
- **mapPartitionsWithIndex()**: Applies a function to each partition, providing the partition index.

69. What is `map` in Apache Spark?

`map` is a transformation that applies a function to each element in the RDD, resulting in a new RDD.

70. What is `flatMap` in Apache Spark?

`flatMap` is a transformation that applies a function to each element, resulting in multiple elements (a flat structure) for each input.

71. Explain `fold()` operation in Spark.

`fold()` aggregates the elements of an RDD using an associative function and a "zero value" (an initial value).

72. Explain `createOrReplaceTempView()` API.

`createOrReplaceTempView()` registers a DataFrame as a temporary table in Spark SQL, allowing it to be queried using SQL.

73. Explain `values()` operation in Apache Spark.

`values()` returns an RDD containing only the values of key-value pairs.

74. Explain `keys()` operation in Apache Spark.

`keys()` returns an RDD containing only the keys of key-value pairs.

75. Explain `textFile` vs `wholeTextFiles` in Spark.

- **textFile()**: Reads a text file and creates an RDD of strings, each representing a line.
- **wholeTextFiles()**: Reads entire files and creates an RDD of (filename, content) pairs.

76. Explain `cogroup()` operation in Spark.

`cogroup()` groups data from two or more RDDs sharing the same key.

77. Explain `pipe()` operation in Apache Spark.

`pipe()` passes each partition of an RDD to an external script or program and returns the output as an RDD.

78. Explain Spark `coalesce()` operation.

`coalesce()` reduces the number of partitions in an RDD, useful for minimizing shuffling when reducing the data size.

79. Explain the `repartition()` operation in Spark.

`repartition()` reshuffles data across partitions, increasing or decreasing the number of partitions, involving a full shuffle of data.

80. Explain `fullOuterJoin()` operation in Apache Spark.

`fullOuterJoin()` returns an RDD with all pairs of elements for matching keys and `null` for non-matching keys from both RDDs.

81. Explain Spark `leftOuterJoin()` and `rightOuterJoin()` operations.

- **leftOuterJoin()**: Returns all key-value pairs from the left RDD and matching pairs from the right, filling with `null` where no match is found.
- **rightOuterJoin()**: Returns all key-value pairs from the right RDD and matching pairs from the left, filling with `null` where no match is found.

82. Explain Spark `join()` operation.

`join()` returns an RDD with all pairs of elements with matching keys from both RDDs.

83. Explain `top()` and `takeOrdered()` operations.

- **top()**: Returns the top `n` elements from an RDD in descending order.
- **takeOrdered()**: Returns the top `n` elements from an RDD in ascending order.

84. Explain `first()` operation in Spark.

`first()` returns the first element of an RDD.

85. Explain `sum()`, `max()`, `min()` operations in Apache Spark.

These operations compute the sum, maximum, and minimum of elements in an RDD, respectively.

86. Explain `countByValue()` operation in Apache Spark RDD.

`countByValue()` returns a map of the counts of each unique value in the RDD.

87. Explain the `lookup()` operation in Spark.

`lookup()` returns the list of values associated with a given key in a paired RDD.

88. Explain Spark `countByKey()` operation.

`countByKey()` returns a map of the counts of each key in a paired RDD.

89. Explain Spark `saveAsTextFile()` operation.

`saveAsTextFile()` saves the RDD content as a text file or set of text files.

90. Explain `reduceByKey()` Spark operation.

`reduceByKey()` applies a reducing function to the elements with the same key, reducing them to a single element per key.

91. Explain the operation `reduce()` in Spark.

`reduce()` aggregates the elements of an RDD using an associative and commutative function.

92. Explain the action `count()` in Spark RDD.

`count()` returns the number of elements in an RDD.

93. Explain Spark `map()` transformation.

`map()` applies a function to each element of an RDD, creating a new RDD with the results.

94. Explain the `flatMap()` transformation in Apache Spark.

`flatMap()` applies a function that returns an iterable to each element and flattens the results into a single RDD.

95. What are the limitations of Apache Spark?

Limitations include high memory consumption, not ideal for OLTP (transactional processing), lack of a mature security framework, and dependency on cluster resources.

96. What is Spark SQL?

Spark SQL is a Spark module for structured data processing, providing a DataFrame API and allowing SQL queries to be executed.

97. Explain Spark SQL caching and uncaching.

- **Caching**: Storing DataFrames in memory for faster access.
- **Uncaching**: Removing cached DataFrames to free memory.

98. Explain Spark Streaming.

Spark Streaming is an extension of Spark for processing real-time data streams.

99. What is DStream in Apache Spark Streaming?

DStream (Discretized Stream) is a sequence of RDDs representing a continuous stream of data.

100. Explain different transformations in DStream in Apache Spark Streaming.

Transformations include:

- **map()**, **flatMap()**, **filter()**
- **reduceByKeyAndWindow()**
- **window()**, **countByWindow()**
- **updateStateByKey()**

101. What is the Starvation scenario in Spark Streaming?

Starvation occurs when all tasks are waiting for resources that are occupied by other long-running tasks, leading to delays or deadlocks.

102. Explain the level of parallelism in Spark Streaming.

Parallelism is controlled by the number of partitions in RDDs; increasing partitions increases the level of parallelism.

103. What are the different input sources for Spark Streaming?

Input sources include:

- Kafka
- Flume
- Kinesis
- Socket
- HDFS or S3

104. Explain Spark Streaming with Socket.

Spark Streaming can receive real-time data streams over a socket using `socketTextStream()`.

105. Define the roles of the file system in any framework.

The file system manages data storage, access, and security, ensuring data integrity and availability.

106. How do you parse data in XML? Which kind of class do you use with Java to parse data?

To parse XML data in Java, you can use classes from the `javax.xml.parsers` package, such as:

- **DocumentBuilder**: Used with the Document Object Model (DOM) for in-memory tree representation.
- **SAXParser**: Used with the Simple API for XML (SAX) for event-driven parsing.

107. What is PageRank in Spark?

PageRank is an algorithm used to rank web pages in search engine results, based on the number and quality of links to a page. In Spark, it can be implemented using RDDs or DataFrames to compute the rank of nodes in a graph.

108. What are the roles and responsibilities of worker nodes in the Apache Spark cluster? Is the Worker Node in Spark the same as the Slave Node?

- **Worker Nodes**: Execute tasks assigned by the Spark Driver, manage executors, and store data in memory or disk as required.
- **Slave Nodes**: Worker nodes in Spark are commonly referred to as slave nodes. Both terms are used interchangeably.

109. How to split a single HDFS block into partitions in an RDD?

When reading from HDFS, Spark splits a single block into multiple partitions based on the number of available cores or executors. You can also use the `repartition()` method to explicitly specify the number of partitions.

110. On what basis can you differentiate RDD, DataFrame, and DataSet?

- **RDD**: Low-level, unstructured data; provides functional programming APIs.
- **DataFrame**: Higher-level abstraction with schema; optimized for SQL queries and transformations.
- **Dataset**: Combines features of RDDs and DataFrames; offers type safety and object-oriented programming.

SPARK BASED TOPICS KEYWORDS:

Spark Intro:

1. Spark : In-memory processing engine
2. Why spark is fast: Due to less I/O disc reads and writes
3. RDD: It is a data structure to store data in spark
4. When RDD fails: Using lineage graph we track which RDD failed and reprocess it
5. Why RDD immutable : As it has to be recovered after its failure and to track which RDD failed
6. Operations in spark: Transformation and Action
7. Transformation: Change data from one form to another, are lazy.
8. Action: Operations which processes the tranformations, not lazy. creates DAG to remember sequence of steps.
9. Broadcast Variables: Data which is distributed to all the systems. Similar to map side join in hive
10. Accumulators: Shared copy in driver, executors can update but not read. Similar to counters in MR
11. MR before Yarn: Job tracker (scheduling &monitoring), task manager (manages tasks in its node)
12. Limitations of MR: Unable to add new clusters(scalable), resource under-utilization, only MR jobs handled
13. YARN: Resource manager(scheduling), application master(monitoring & resource negotiation), node manager (manages tasks in its node)
14. Uberization: Tasks run on AM itself if they are very small

15. Spark components: Driver (gives location of executors) and executors(process data in memory)
16. Client Mode: Driver is at client side
17. Cluster Mode: Driver is inside AM in the cluster
18. Types of transformation: Narrow and Wide
19. Narrow: Data shuffling doesn't happen (map, flatMap, filter)
20. Wide: Data shuffling happens (reduceByKey, groupByKey)
21. reduceByKey() is a transformation and reduce() is an action
22. reduceByKey(): Data is processed at each partition, groupByKey() : Data is grouped at each partition and complete processing is done at reducer.
23. Repartition: used to increase/decrease partitions. Use it for INCREASE

    Coalesce: used to decrease partitions and optimized as data shuffling is less

SPARK DATAFRAMES:

1. Cache() : It is used to cache the data only in memory. Rdd.cache()
2. Persist() : it is used to cache the data in different storage levels (memory, disc, memory & disc, off heap). Rdd.persist(StorageLevel.____)
3. Serialization: Process of converting data in object form into bytes, occupies less space

4. De-Serialization: Process of converting data in bytes back to objects, occupies more space.
5. DAG : Created when an action is called, represents tasks, stages of a job
6. Map : performs one-to-one mapping on each line of input
7. mapPartitions: performs map function only once on each partition
8. Driver: converts high level programming constructs to low level to be fed to executors (dataframe to rdd)
9. Executors: Present in memory to process the rdd
10. Spark context: creates entry point into spark cluster for spark appl
11. Spark session: creates unified entry point into spark cluster
12. Data frame: it is a dataset[row] where type error caught only at run time
13. Data set: it is a dataset[object] where type error caught at compile time
14. Modes of dealing with corrupted record: permissive, malformed, fail fast
15. Schema types: implicit, infer, explicit (case class, StructType, DDL string)

SPARK OPTIMIZATIONS

1. Spark optimization:

a. Cluster Configuration : To configure resources to the cluster so that spark jobs can process well.

b. Code configuration: To apply optimization techniques at code level so that processing will be fast.
2. Thin executor: More no. of executors with less no. of resources. Multithreading not possible, too many broadcast variables required. Ex. 1 executor with each 2 cpu cores, 1 GB ram.
3. Fat executor: Less no. of executors with more amount of resources. System performance drops down, garbage collection takes time. Ex 1 executor 16 cpu cores, 32 GB ram.
4. Garbage collection: To remove unused objects from memory.
5. Off heap memory: Memory stored outside of executors/ jvm. It takes less time to clean objects than garbage collector, used for java overheads (extra memory which directly doesn't add to performance but required by system to carry out its operation)
6. Static allocation: Resources are fixed at first and will remain the same till the job ends.
7. Dynamic Allocation: Resources are allocated dynamically based on the job requirement and released during job stages if they are no longer required.
8. Edge node: It is also called as gateway node which is can be accessed by client to enter into hadoop cluster and access name node.
9. How to increase parallelism :

a. Salting : To increase no. of distinct keys so that work can be distributed across many tasks which in turn increase parallelism.
b. Increase no. of shuffle partitions
c. Increase the resources of the cluster (more cpu cores)

10. Execution memory : To perform computations like shuffle, sort, join
11. Storage memory : To store the cache
12. User memory : To store user's data structures, meta data etc.
13. Reserved memory : To run the executors

14. Kyro Serializer: Used to store the data in disk in serialized manner which occupies less space.
15. Broadcast join: Used to send the copies of data to all executors. Used when we have only 1 big table.
16. Optimization on using coalesce() rather than repartition while reducing no. of partitions
17. Join optimizations:

a. To avoid or minimize shuffling of data
b. To increase parallelism

1. How to avoid/minimize shuffling?

a. Filter and aggregate data before shuffling
b. Use optimization methods which require less shuffling (coalesce())

18. How to increase parallelism ?

a. **Min (total cpu cores, total shuffle partitions, total distinct keys)**
b. Use salting to increase no. of distinct keys
c. Increase default no. of shuffle partitions
d. Increase resources to inc total cpu cores

19. Skew partitions : Partitions in which data is unevenly distributed. Bucketing, partitioning, salting can be used to handle it.
20. Sort aggregate: Data is sorted based on keys and then aggregated. More processing time
21. Hash aggregate: Hash table is created and similar keys are added to the same hash value. Less processing time.
22. Stages of execution plan :

a. Parsed logical plan (unresolved logical plan) : To find out syntax errors

b. Analytical logical plan (Resolved logical plan) : Checks for column and table names from the catalog.
c. Optimized logical plan (Catalyst optimization) : Optimization done based on built in rules.
d. Physical plan : Actual execution plan is selected based on cost effective model.
e. Conversion into Rdd : Converted into rdd and sent to executors for processing.

**Note:
1 hdfs block = 1 rdd partition = 128mb
1 hdfs block in local=1 rdd partition in local spark cluster= 32mb
1 rdd ~ can have n partitions in it
1 cluster = 1 machine
N cores = N blocks can run in parallel in each cluster/machine
N stages = N - 1 wide transformations
N tasks in each stage= N partitions in each stage for that rdd/data frame

Must Know Pyspark Coding Before Your Next Databricks Interview

Document by – Siddhartha Subudhi

Visit my LinkedIn profile

1. Find the second highest salary in a DataFrame using PySpark.

Scenario: You have a DataFrame of employee salaries and want to find the second highest salary.

```
from pyspark.sql import Window
from pyspark.sql.functions import col, dense_rank

windowSpec = Window.orderBy(col("salary").desc())
df_with_rank = df.withColumn("rank", dense_rank().over(windowSpec))
second_highest_salary = df_with_rank.filter(col("rank") == 2).select("salary")
second_highest_salary.show()
```

---

2. Count the number of null values in each column of a PySpark DataFrame.

Scenario: Given a DataFrame, identify how many null values each column contains.

```
from pyspark.sql.functions import col, isnan, when, count

df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()
```

---

3. Calculate the moving average over a window of 3 rows.

Scenario: For a stock price dataset, calculate a moving average over the last 3 days.

```
from pyspark.sql import Window
from pyspark.sql.functions import avg

windowSpec = Window.orderBy("date").rowsBetween(-2, 0)
df_with_moving_avg = df.withColumn("moving_avg", avg("price").over(windowSpec))
df_with_moving_avg.show()
```

---

4. Remove duplicate rows based on a subset of columns in a PySpark DataFrame.

Scenario: You need to remove duplicates from a DataFrame based on certain columns.

```
df = df.dropDuplicates(["column1", "column2"])
df.show()
```

5. Split a single column with comma-separated values into multiple columns.

Scenario: Your DataFrame contains a column with comma-separated values. You want to split this into multiple columns.

```
from pyspark.sql.functions import split


df_split = df.withColumn("new_column1", split(df["column"], ",").getItem(0)) \
        .withColumn("new_column2", split(df["column"], ",").getItem(1))
df_split.show()
```

---

6. Group data by a specific column and calculate the sum of another column.

Scenario: Group sales data by "product" and calculate the total sales.

```
df.groupBy("product").sum("sales").show()
```

---

7. Join two DataFrames on a specific condition.

Scenario: You have two DataFrames: one for customer data and one for orders. Join these DataFrames on the customer ID.

```
df_joined = df_customers.join(df_orders, df_customers.customer_id == df_orders.customer_id, "inner")
df_joined.show()
```

---

8. Create a new column based on conditions from existing columns.

Scenario: Add a new column "category" that assigns "high", "medium", or "low" based on the value of the "sales" column.

```
from pyspark.sql.functions import when
df = df.withColumn("category", when(df.sales > 500, "high")
                   .when((df.sales <= 500) & (df.sales > 200), "medium")
                   .otherwise("low"))
df.show()
```

9. Calculate the percentage contribution of each value in a column to the total.

Scenario: For a sales dataset, calculate the percentage contribution of each product's sales to the total sales.

```
from pyspark.sql.functions import sum, col
total_sales = df.agg(sum("sales").alias("total_sales")).collect()[0]["total_sales"]
df = df.withColumn("percentage", (col("sales") / total_sales) * 100)
df.show()
```

---

10. Find the top N records from a DataFrame based on a column.

Scenario: You need to find the top 5 highest-selling products.

```
df.orderBy(col("sales").desc()).limit(5).show()
```

---

11. Write PySpark code to pivot a DataFrame.

Scenario: You have sales data by "year" and "product", and you want to pivot the table to show "product" sales by year.

```
df_pivot = df.groupBy("product").pivot("year").sum("sales")
df_pivot.show()
```

---

12. Add row numbers to a PySpark DataFrame based on a specific ordering.

Scenario: Add row numbers to a DataFrame ordered by "sales" in descending order.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number
windowSpec = Window.orderBy(col("sales").desc())
df_with_row_number = df.withColumn("row_number", row_number().over(windowSpec))
df_with_row_number.show()
```

13. Filter rows based on a condition.

Scenario: You want to filter only those customers who made purchases over ₹1000.

```
df_filtered = df.filter(df.purchase_amount > 1000)
df_filtered.show()
```

---

14. Flatten a JSON column in PySpark.

Scenario: Your DataFrame contains a JSON column, and you want to extract specific fields from it.

```
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StructField, StringType
schema = StructType([
   StructField("name", StringType(), True),
   StructField("age", StringType(), True)
])
df = df.withColumn("json_data", from_json(col("json_column"), schema))
df.select("json_data.name", "json_data.age").show()
```

---

15. Convert a PySpark DataFrame column to a list.

Scenario: Convert a column from your DataFrame into a list for further processing.

```
column_list = df.select("column_name").rdd.flatMap(lambda x: x).collect()
```

---

16. Handle NULL values by replacing them with a default value.

Scenario: Replace all NULL values in the "sales" column with 0.

```
df = df.na.fill({"sales": 0})
df.show()
```

---

17. Perform a self-join on a PySpark DataFrame.

Scenario: You have a hierarchy of employees and want to find each employee's manager.

```
df_self_join = df.alias("e1").join(df.alias("e2"), col("e1.manager_id") == col("e2.employee_id"), "inner") \
        .select(col("e1.employee_name"), col("e2.employee_name").alias("manager_name"))
df_self_join.show()
```


18. Write PySpark code to unpivot a DataFrame.

Scenario: You have a DataFrame with "year" columns and want to convert them to rows.

```
from pyspark.sql.functions import expr
df_unpivot = df.selectExpr("id", "stack(2, '2021', sales_2021, '2022', sales_2022) as (year, sales)")
df_unpivot.show()
```

---

19. Write a PySpark code to group data based on multiple columns and calculate aggregate functions.

Scenario: Group data by "product" and "region" and calculate the average sales for each group.

```
df.groupBy("product", "region").agg({"sales": "avg"}).show()
```

---

20. Write PySpark code to remove rows with duplicate values in any column.

Scenario: You want to remove rows where any column has duplicate values.

```
df_cleaned = df.dropDuplicates()
df_cleaned.show()
```

---

21. Write PySpark code to read a CSV file and infer its schema.

Scenario: You need to load a CSV file into a DataFrame, ensuring the schema is inferred.

```
df = spark.read.option("header", "true").option("inferSchema", "true").csv("path_to_csv")
df.show()
```

---

22. Write PySpark code to merge multiple small files into a single file.

Scenario: You have multiple small files in HDFS, and you want to consolidate them into one large file.

```
df.coalesce(1).write.mode("overwrite").csv("output_path")
```

23. Write PySpark code to calculate the cumulative sum of a column.

Scenario: You want to calculate a cumulative sum of sales in your DataFrame.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum

windowSpec = Window.orderBy("date").rowsBetween(Window.unboundedPreceding, 0)
df_with_cumsum = df.withColumn("cumulative_sum", sum("sales").over(windowSpec))
df_with_cumsum.show()
```

---

24. Write PySpark code to find outliers in a dataset.

Scenario: Detect outliers in the "sales" column based on the 1.5 * IQR rule.

```
from pyspark.sql.functions import expr
q1 = df.approxQuantile("sales", [0.25], 0.01)[0]
q3 = df.approxQuantile("sales", [0.75], 0.01)[0]
iqr = q3 - q1
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr
df_outliers = df.filter((col("sales") < lower_bound) | (col("sales") > upper_bound))
df_outliers.show()
```

25. Write PySpark code to convert a DataFrame to a Pandas DataFrame.

Scenario: Convert your PySpark DataFrame into a Pandas DataFrame for local processing.

```
pandas_df = df.toPandas()
```

Data Cleaning with Apache Spark

Spark Schema

```
.printSchema()
```

```
# Import schema

import pyspark.sql.types
from pyspark.sql.types import *

peopleSchema = StructType([
# Define the name field
StructField('name', StringType(), True), # Define the name field
StructField('age', IntegerType(), True), # Add the age field
StructField('city', StringType(), True) ]) # Add the city field
#if false, means cannot be nullable.

people_df = spark.read.format('csv').load(name='rawdata.csv',
schema=peopleSchema)
```

Immutability and Lazy Processing

A component of functional programming
Defined once
Unable to be directly modied
Re-created if reassigned
Able to be shared efficiently

```
# Immutability Example
# Load the CSV file
aa_dfw_df = spark.read.format('csv').options(Header=True).load('AA_DFW_2018.csv

# Add the airport column using the F.lower() method
aa_dfw_df = aa_dfw_df.withColumn('airport', F.lower(aa_dfw_df['Destination Airpo

# Drop the Destination Airport column
aa_dfw_df = aa_dfw_df.drop(aa_dfw_df['Destination Airport'])

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

Understanding Parquet

Difficulties with CSV files

No defined schema (no data type, no column name beyond a header row)

Nested data requires special (content containing a comma requires escaping, using the escape character within content requires even further escaping)

handling Encoding format limited

for spark: slow to parse, cannot be shared during the import process; if no schema is defined, all data must be read before a schema can be inferred, forcing the code to read the file twice.

for spark: files cannot be filtered (no 'predicate pushdown', ordering tasks to do the least amount of work, filtering data prior to processing is one of the primary optimizations of predicate pushdown.)

for spark: any intermediate use requires redefining schema. Spark process are often multi-step and may utilize an intermediate file representation. These representations allow data to be used later without regenerating the data from source.

The Parquet Format

columnar data format

supported in spark and other data processing frameworks

supports predicate pushdown

automatically stores schema information

binary file format

Parquet is a compressed columnar data format developed for use in any Hadoop based system. Include: Spark, Hadoop, Apache Impala... Perfect for intermediary or on-disk representation of processed data.

predicate pushdown This means Spark will only process the data necessary to complete the operations you define versus reading the entire dataset.

Working with Parquet

```
df = spark.read.format('parquet').load('filename.parquet')
df = spark.read.parquet('filename.parquet')

df.write.format('parquet').save('filename.parquet')
df.write.parquet('filename.parquet') #mode='overwrite'
```

Parquet with SQL

parquet - dataframe - table

```
flight_df = spark.read.parquet('flights.parquet')
flight_df.createOrReplaceTempView('flights')
short_flights_df = spark.sql('SELECT * FROM flights WHERE flightduration <100')

# Run a SQL query of the average flight duration
avg_duration = spark.sql('SELECT avg(flight_duration) from flights').collect()[(
print('The average flight time is: %d' % avg_duration)
```

DataFrame column operations

DataFrames:

Made up of rows & columns

Immutable

Use various transformation operations to modify data

```
# Return rows where name starts with "M"
voter_df.filter(voter_df.name.like('M%'))
# Return name and position only
voters = voter_df.select('name', 'position')
```

Common DataFrame transformations

```
#Filter / Where interchangeable
voter_df.filter(voter_df.date > '1/1/2019') # or voter_df.where(...)

#Select
voter_df.select(voter_df.name)

#withColumn creates new column
voter_df.withColumn('year', voter_df.date.year)

#drop
voter_df.drop('unused_column')
```

Filtering data

Remove nulls

Remove odd entries

Split data from combined sources

Negate with ~

```
voter_df.filter(voter_df['name'].isNotNull())
voter_df.filter(voter_df.date.year > 1800)
voter_df.where(voter_df['_c0'].contains('VOTE'))
voter_df.where(~ voter_df._c1.isNull())
```

```
# Show the distinct VOTER_NAME entries
voter_df.select('VOTER_NAME').distinct().show(40, truncate=False)

# Filter voter_df where the VOTER_NAME is 1-20 characters in length
voter_df = voter_df.filter('length(VOTER_NAME) > 0 and length(VOTER_NAME) < 20')

# Filter out voter_df where the VOTER_NAME contains an underscore
voter_df = voter_df.filter(~ F.col('VOTER_NAME').contains('_'))

# Show the distinct VOTER_NAME entries again
voter_df.select('VOTER_NAME').distinct().show(40, truncate=False)
```

Column string transformations

```
#Contained in pyspark.sql.functions
import pyspark.sql.functions as F

#Applied per column as transformation
voter_df.withColumn('upper', F.upper('name'))

#Can create intermediary columns
voter_df.withColumn('splits', F.split('name', ' '))

#Can cast to other types
voter_df.withColumn('year',voter_df[' c4'].cast(IntegerType()))
```

ArrayType() column functions

Various utility functions / transformations to interact with ArrayType() .

. `size()` - returns length of arrayType() column

. `getItem()` - used to retrieve a specific item at index of list column.

```
# Add a new column called splits separated on whitespace
voter_df = voter_df.withColumn('splits', F.split(voter_df.VOTER_NAME, '\s+'))

# Create a new column called first_name based on the first item in splits
voter_df = voter_df.withColumn('first_name', voter_df.splits.getItem(0))

# Get the last entry of the splits list and create a column called last_name
voter_df = voter_df.withColumn('last_name', voter_df.splits.getItem(F.size('spl

# Drop the splits column
voter_df = voter_df.drop('splits')

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

Conditional DataFrame column operations
Conditional clauses

Conditional Clauses are:

Inline version of if / then / else

`.when()`

`.otherwise()`

```
df.select(df.Name, df.Age,
.when(df.Age >= 18, "Adult")
.when(df.Age < 18, "Minor"))

df.select(df.Name, df.Age,
.when(df.Age >= 18, "Adult")
.otherwise("Minor"))

# Add a column to voter_df for any voter with the title **Councilmember**
voter_df = voter_df.withColumn('random_val',
          when(voter_df.TITLE == 'Councilmember', F.rand()))

# Add a column to voter_df for a voter based on their position
voter_df = voter_df.withColumn('random_val',
    when(voter_df.TITLE == 'Councilmember', F.rand())
    .when(voter_df.TITLE == 'Mayor', 2)
    .otherwise(0))
```

User defined functions

Python method

Wrapped via the `pyspark.sql.functions.udf` method

Stored as a variable

Called like a normal Spark function

```
def getFirstAndMiddle(names):
  # Return a space separated string of names
  return ' '.join(names[:-1])

# Define the method as a UDF
udfFirstAndMiddle = F.udf(getFirstAndMiddle, StringType())

# Create a new column using your UDF
voter_df = voter_df.withColumn('first_and_middle_name', udfFirstAndMiddle(voter_

# Drop the unecessary columns then show the DataFrame
voter_df = voter_df.drop('first_name')
voter_df = voter_df.drop('splits')
voter_df.show()
```

Reverse string UDF

```
#Define a Python method
def reverseString(mystr):
    return mystr[::-1]

#Wrap the function and store as a variable
udfReverseString = udf(reverseString, StringType())

#Use with Spark
user_df = user_df.withColumn('ReverseName', udfReverseString())
```

Argument-less example

```
def sortingCap():
  return random.choice(['G', 'H', 'R', 'S'])
udfSortingCap = udf(sortingCap, StringType())
user_df = user_df.withColumn('Class', udfSortingCap())
```

Partitioning and lazy processing

Partitioning

DataFrames are broken up into partitions

Partition size can vary

Each partition is handled independently

To check the number of partitions, use the method `.rdd.getNumPartitions()` on a DataFrame.

Lazy processing

Transformations are lazy
* .withColumn(...)
* .select(...)
* .cache()

Nothing is actually done until an action is performed
* .count()
* .write(...)
*. show()

Transformations can be re-ordered for best performance

Sometimes causes unexpected behavior

Adding IDs

Normal ID fields:

Common in relational databases

Most usually an integer increasing, sequential and unique

Not very parallel

Monotonically increasing IDs

`pyspark.sql.functions.monotonically_increasing_id()`

Integer (64-bit), increases in value, unique

Not necessarily sequential (gaps exist)

Completely parallel

```
# Select all the unique council voters
voter_df = df.select(df["VOTER NAME"]).distinct()

# Count the rows in voter_df
print("\nThere are %d rows in the voter_df DataFrame.\n" % voter_df.count())

# Add a ROW_ID
voter_df = voter_df.withColumn('ROW_ID', F.monotonically_increasing_id())

# Show the rows with 10 highest IDs in the set
voter_df.orderBy(voter_df.ROW_ID.desc()).show(10)
```

More ID Tricks

Depending on your needs, you may want to start your IDs at a certain value so there isn't overlap with previous runs of the Spark task. This behavior is similar to how IDs would behave in a relational database. Make sure that the IDs output from a monthly Spark task start at the highest value from the previous month.

Caching

Caching in Spark:

Stores DataFrames in memory or on disk

Improves speed on later transformations / actions

Reduces resource usage

Disadvantages of caching

Very large data sets may not t in memory

Local disk based caching may not be a performance improvement

Cached objects may not be available

Caching tips When developing Spark tasks:

Cache only if you need it

Try caching DataFrames at various points and determine if your performance improves

Cache in memory and fast SSD / NVMe storage

Cache to slow local disk if needed

Use intermediate files!

Stop caching objects when finished

Eviction Policy Least Recently Used (LRU)

Caching is a lazy operation. It requires an action to trigger it. eg.

spark.sql("select count(*) from text").show()

partitioned_df.count()

```
df.cache()    #df.persist() df.persist(storageLevel=pyspark.StorageLevel.MEMORY_
df.unpersist()

#Determining whether a dataframe is cached
df.is_cached

#storage level useDisk useMemory useOffHeap deserialized replication
df.storageLevel

#Caching a table
df.createOrReplaceTempView('df')
spark.catalog.cacheTable('df')
spark.catalog.isCached(tableName='df')
spark.catalog.dropTempView('table1')
# List the tables
print("Tables:\n", spark.catalog.listTables())

# Uncaching a table
spark.catalog.uncacheTable('df')
spark.catalog.clearCache()
```

Improve import performance

Spark clusters

Spark Clusters are made of two types of processes

Driver process

Worker processes

Import performance

Important parameters:

Number of objects (Files, Network locations, etc)

More objects better than larger ones

Can import via wildcard

```
airport_df = spark.read.csv('airports-*.txt.gz')
```

General size of objects

Spark performs better if objects are of similar size

It's safe to assume the more import objects available, the better the cluster can divvy up the job.

Schemas

A well-dened schema will drastically improve import performance

Avoids reading the data multiple times

Provides validation on import

How to split objects

Use OS utilities / scripts (split, cut, awk)

```
split -l 10000 -d largefile chunk-
```

> 每个文件100000行，字符，名字叫largefile，生成chunk0000开始

Use custom scripts

Write out to Parquet

```
df_csv = spark.read.csv('singlelargefile.csv')
```

```
df_csv.write.parquet('data.parquet')
df = spark.read.parquet('data.parquet')
```

Explaining the Spark execution plan

```
voter_df = df.select(df['VOTER NAME']).distinct()
voter_df.explain()
```

> == Physical Plan ==
> *(2) HashAggregate(keys=[VOTER NAME#15], functions=[]) +- Exchange hashpartitioning(VOTER NAME#15, 200)
> +- *(1) HashAggregate(keys=[VOTER NAME#15], functions=[])
> +- *(1) FileScan csv [VOTER NAME#15] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/DallasCouncilVotes.csv.gz],
> PartitionFilters: [], PushedFilters: [],
> ReadSchema: struct<VOTER NAME:string>

Shuffling

Shuffling refers to moving data around to various workers to complete a task

Hides complexity from the user

Can be slow to complete

Lowers overall throughput

Is often necessary, but try to minimize

How to limit shuffling

Limit use of `.repartition(num_partitions)`
Use `.coalesce(num_partitions)` instead

Use care when calling `.join()` Use `.broadcast()`

May not need to limit it

Broadcasting

Provides a copy of an object to each worker

Prevents undue / excess communication between nodes

Can drastically speed up `.join()` operations

A couple tips:

Broadcast the smaller DataFrame. The larger the DataFrame, the more time required to transfer to the worker nodes.

On small DataFrames, it may be better skip broadcasting and let Spark figure out any optimization on its own.

If you look at the query execution plan, a broadcastHashJoin indicates you've successfully configured broadcasting.

Use the `.broadcast(<DataFrame>)` method

```
from pyspark.sql.functions import broadcast
combined_df = df_1.join(broadcast(df_2))
```

Cluster Configurations

Configuration options

Spark contains many configuration settings

These can be modied to match needs

Reading configuration settings: `spark.conf.get(<configuration name>)`

Writing configuration settings `spark.conf.set(<configuration name>)`

Configure Spark to use 500 partitions
`spark.conf.set('spark.sql.shuffle.partitions', 500)`

```
# Name of the Spark application instance
app_name = spark.conf.get('spark.app.name')

# Driver TCP port
driver_tcp_port = spark.conf.get('spark.driver.port')

# Number of join partitions
num_partitions = spark.conf.get('spark.sql.shuffle.partitions')

# Store the number of partitions in variable
before = departures_df.rdd.getNumPartitions()

# Configure Spark to use 500 partitions
spark.conf.set('spark.sql.shuffle.partitions', 500)

# Recreate the DataFrame using the departures data file
departures_df = spark.read.csv('departures.txt.gz').distinct()
```

Cluster Types

Spark deployment options:

Single node (deploying all components on a single system, can be physical/VM/container)

Standalone (dedicated machines as the driver and workers)

Managed (cluster components are handled by a third party cluster manager)

YARN

Mesos

Kubernetes

Driver

Task assignment

Result consolidation

Shared data access

Tips:

Driver node should have double the memory of the worker

Fast local storage helpful

Worker

Runs actual tasks

Ideally has all code, data, and resources for a given task

Recommendations:

More worker nodes is often better than larger workers

Test to find the balance

Fast local storage extremely useful

Data Pipelines

Input(s)
CSV, JSON, web services, databases

Transformations
`withColumn()` ,
`.filter()` ,
`.drop()`

Output(s)
CSV, Parquet, database Validation

Analysis

Pipeline details

Not formally dened in Spark

Typically all normal Spark code required for task

```
schema = StructType([
 StructField('name', StringType(), False),
 StructField('age', StringType(), False)
 ])
 df = spark.read.format('csv').load('datafile').schema(schema) df = df.withColur
 ...
 df.write.parquet('outdata.parquet')
 df.write.json('outdata.json')
```

```
 # Import the data to a DataFrame
departures_df = spark.read.csv('2015-departures.csv.gz', header=True)

# Remove any duration of 0
departures_df = departures_df.filter(~departures_df[3].contains('0'))

# Add an ID column
departures_df = departures_df.withColumn('id', F.monotonically_increasing_id())

# Write the file out to JSON format
departures_df.write.json('output.json')
```

Data Handling

Parsing

Incorrect data
* Empty rows
* Commented lines
* Headers

Nested structures
* Multiple delimiters

Non-regular data
 * Differing numbers of columns per row

Removing blank lines, headers, and comments

Spark's CSV parser:

Automatically removes blank lines

Can remove comments using an optional argument

```
df1 = spark.read.csv('datafile.csv.gz', comment='#')
```

Handles header fields

Defined via argument

Ignored if a schema is defined

```
df1 = spark.read.csv('datafile.csv.gz', header='True')
```

Count the number of rows beginning with '#'

```
comment_count =
annotations_df.where(col('_c0').startswith('#')).count()
```

```
# Split _c0 on the tab character and store the list in a variable
tmp_fields = F.split(annotations_df['_c0'], '\t')

# Create the colcount column on the DataFrame
annotations_df = annotations_df.withColumn('colcount', F.size(tmp_fields))

# Remove any rows containing fewer than 5 fields
annotations_df_filtered = annotations_df.filter(~ (annotations_df["colcount"] <

# Count the number of rows
final_count = annotations_df_filtered.count()
print("Initial count: %d\nFinal count: %d" % (initial_count, final_count))
```

Automatic column creation

Spark will:

Automatically create columns in a DataFrame based on sep argument `df1 = spark.read.csv('datafile.csv.gz', sep=',')`

Defaults to using `,`

Can still successfully parse if sep is not in string

```
df1 = spark.read.csv('datafile.csv.gz', sep='*')
```

Stores data in column defaulting to `_c0`

Allows you to properly handle nested separators

```
# Split the content of _c0 on the tab character (aka, '\t')
split_cols = F.split(annotations_df['_c0'], '\t')

# Add the columns folder, filename, width, and height
split_df = annotations_df.withColumn('folder', split_cols.getItem(0))
split_df = split_df.withColumn('filename', split_cols.getItem(1))
split_df = split_df.withColumn('width', split_cols.getItem(2))
split_df = split_df.withColumn('height', split_cols.getItem(3))


def retriever(cols, colcount):
  # Return a list of dog data
  return cols[4:colcount]

# Define the method as a UDF
udfRetriever = F.udf(retriever, ArrayType(StringType()))

# Create a new column using your UDF
split_df = split_df.withColumn('dog_list', udfRetriever(split_df.split_cols, sp

# Remove the original column, split_cols, and the colcount
split_df = split_df.drop('_c0').drop('colcount').drop('split_cols')
```

Data Validation

Validation is:

Verifying that a dataset complies with the expected format

Number of rows / columns

Data types

Complex validation rules

Validating via joins

Compares data against known values

Easy to find data in a given set

Comparatively fast

```
parsed_df = spark.read.parquet('parsed_data.parquet')
company_df = spark.read.parquet('companies.parquet')
verified_df = parsed_df.join(company_df, parsed_df.company == company_df.company
```

```
# Rename the column in valid_folders_df
valid_folders_df = valid_folders_df.withColumnRenamed('_c0', 'folder')

# Count the number of rows in split_df
split_count = split_df.count()

# Join the DataFrames
joined_df = split_df.join(F.broadcast(valid_folders_df), "folder")

# Compare the number of rows remaining
joined_count = joined_df.count()
print("Before: %d\nAfter: %d" % (split_count, joined_count))
```

This *automatically* removes any rows with a company not in the valid_df !

Complex rule validation

Using Spark components to validate logic:

Calculations

Verifying against external source

Likely uses a UDF to modify / verify the DataFrame

```
# Determine the row counts for each DataFrame
split_count = split_df.count()
joined_count = joined_df.count()

# Create a DataFrame containing the invalid rows
invalid_df = split_df.join(F.broadcast(joined_df), 'folder', 'left_anti')

# Validate the count of the new DataFrame is as expected
invalid_count = invalid_df.count()
print(" split_df:\t%d\n joined_df:\t%d\n invalid_df: \t%d" % (split_count, joine

# Determine the number of distinct folder columns removed
invalid_folder_count = invalid_df.select('folder').distinct().count()
print("%d distinct invalid folders found" % invalid_folder_count)
```

```
# Select the dog details and show 10 untruncated rows
print(joined_df.select('dog_list').show(truncate=False))

# Define a schema type for the details in the dog list
DogType = StructType([
        StructField("breed", StringType(), False),
    StructField("start_x", IntegerType(), False),
    StructField("start_y", IntegerType(), False),
    StructField("end_x", IntegerType(), False),
    StructField("end_y", IntegerType(), False)
])


# Create a function to return the number and type of dogs as a tuple
def dogParse(doglist):
  dogs = []
  for dog in doglist:
    (breed, start_x, start_y, end_x, end_y) = dog.split(',')
    dogs.append((breed, int(start_x), int(start_y), int(end_x), int(end_y)))
  return dogs

# Create a UDF
udfDogParse = F.udf(dogParse, ArrayType(DogType))

# Use the UDF to list of dogs and drop the old column
joined_df = joined_df.withColumn('dogs', udfDogParse('dog_list')).drop('dog_lis

# Show the number of dogs in the first 10 rows
joined_df.select(F.size('dogs')).show(10)
```

```
# Define a UDF to determine the number of pixels per image
def dogPixelCount(doglist):
  totalpixels = 0
  for dog in doglist:
    totalpixels += (dog[3] - dog[1]) * (dog[4] - dog[2])
  return totalpixels

# Define a UDF for the pixel count
udfDogPixelCount = F.udf(dogPixelCount, IntegerType())
joined_df = joined_df.withColumn('dog_pixels', udfDogPixelCount('dogs'))

# Create a column representing the percentage of pixels
joined_df = joined_df.withColumn('dog_percent', (joined_df.dog_pixels / (joined_

# Show the first 10 annotations with more than 60% dog
joined_df.where('dog_percent > 60').show(10)
```

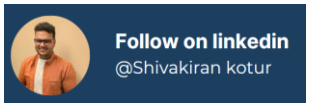
Key Differences in Apache Spark Components and Concepts

Hadoop vs. Spark Architecture

Aspect	Hadoop	Spark
Storage	Uses HDFS for storage	Uses in-memory processing for speed
Processing	MapReduce is disk-based	In-memory processing improves performance
Integration	Runs independently or with Hadoop ecosystem	Can run on top of Hadoop; more flexible
Complexity	More complex setup and deployment	Simpler to deploy and configure
Performance	Slower for iterative tasks due to disk I/O	Better performance for iterative tasks

RDD vs. DataFrame vs. Dataset

Aspect	RDD	DataFrame	Dataset
API Level	Low-level, more control	High-level, optimized with Catalyst	High-level, type-safe
Schema	No schema, unstructured	Uses schema for structured data	Strongly typed, compile-time type safety
Optimization	No built-in optimization	Optimized using Catalyst	Optimized using Catalyst, with type safety
Type Safety	No type safety	No compile-time type safety	Provides compile-time type safety
Performance	Less optimized for performance	Better performance due to optimizations	Combines type safety with optimization



Action vs. Transformation

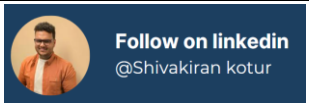
Aspect	Action	Transformation
Execution	Triggers execution of the Spark job	Builds up a logical plan of data operations
Return Type	Returns results or output	Returns a new RDD/DataFrame
Evaluation	Eager evaluation; executes immediately	Lazy evaluation; executed when an action is triggered
Computation	Involves actual computation (e.g., collect())	Defines data transformations (e.g., map())
Performance	Can cause data processing; affects performance	Does not affect performance until an action is called

Map vs. FlatMap

Aspect	Map	FlatMap
Output	Returns one output element per input element	Can return zero or more output elements per input
Flattening	Does not flatten output	Flattens the output into a single level
Use Case	Suitable for one-to-one transformations	Suitable for one-to-many transformations
Complexity	Simpler, straightforward	More complex due to variable number of outputs
Examples	map(x => x * 2)	flatMap(x => x.split(" "))

GroupBykey vs ReduceBykey

Aspect	GroupByKey	ReduceByKey
Operation	Groups all values by key	Aggregates values with the same key
Efficiency	Can lead to high shuffling	More efficient due to partial aggregation
Data Movement	Requires shuffling of all values	Minimizes data movement through local aggregation
Use Case	Useful for simple grouping	Preferred for aggregations and reductions
Performance	Less efficient with large datasets	Better performance for large datasets

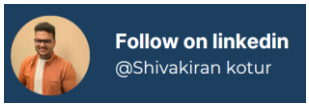


Repartition Vs Coalesce

Aspect	Repartition	Coalesce
Partitioning	Can increase or decrease the number of partitions	Only decreases the number of partitions
Shuffling	Involves full shuffle	Avoids full shuffle, more efficient
Efficiency	More expensive due to shuffling	More efficient for reducing partitions
Use Case	Used for increasing partitions or balancing load	Used for reducing partitions, typically after filtering
Performance	Can be costly for large datasets	More cost-effective for reducing partitions

Cache Vs Presist

Aspect	Cache	Persist
Storage Level	Defaults to MEMORY_ONLY	Can use various storage levels (e.g., MEMORY_AND_DISK)
Flexibility	Simplified, with default storage level	Offers more options for storage levels
Use Case	Suitable for simple caching scenarios	Suitable for complex caching scenarios requiring different storage levels
Implementation	Easier to use, shorthand for MEMORY_ONLY	More flexible, allows custom storage options
Performance	Suitable when memory suffices	More efficient when dealing with larger datasets and limited memory



Narrow Vs Wide Transformation

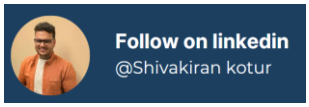
Aspect	Narrow Transformation	Wide Transformation
Partitioning	Each parent partition is used by one child partition	Requires data from multiple partitions
Shuffling	No shuffling required	Involves shuffling of data
Performance	More efficient and less costly	Less efficient due to data movement
Examples	map(), filter()	groupByKey(), join()
Complexity	Simpler and faster	More complex and slower due to data movement

Collect vs Take

Aspect	Collect	Take
Output	Retrieves all data from the RDD/DataFrame	Retrieves a specified number of elements
Memory Usage	Can be expensive and use a lot of memory	More memory-efficient
Use Case	Used when you need the entire dataset	Useful for sampling or debugging
Performance	Can cause performance issues with large data	Faster and more controlled
Action Type	Triggers full data retrieval	Triggers partial data retrieval

Broadcast Variable vs Accumulator

Aspect	Broadcast Variable	Accumulator
Purpose	Efficiently shares read-only data across tasks	Tracks metrics and aggregates values
Data Type	Data that is shared and read-only	Counters and sums, often numerical
Use Case	Useful for large lookup tables or configurations	Useful for aggregating metrics like counts
Efficiency	Reduces data transfer by broadcasting data once	Efficient for aggregating values across tasks
Mutability	Immutable, read-only	Mutable, can be updated during computation



Spark SQL vs DataFrame API

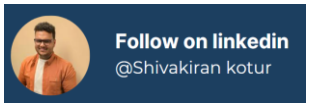
Aspect	Spark SQL	DataFrame API
Interface	Executes SQL queries	Provides a programmatic interface
Syntax	Uses SQL-like syntax	Uses function-based syntax
Optimization	Optimized with Catalyst	Optimized with Catalyst
Use Case	Preferred for complex queries and legacy SQL code	Preferred for programmatic data manipulations
Integration	Can integrate with Hive and other SQL databases	Provides a unified interface for different data sources

Spark Streaming Vs Structured Streaming

Aspect	Spark Streaming	Structured Streaming
Processing	Micro-batch processing	Micro-batch and continuous processing
API	RDD-based API	SQL-based API with DataFrame/Dataset support
Complexity	More complex and lower-level	Simplified with high-level APIs
Consistency	Can be less consistent due to micro-batches	Provides stronger consistency guarantees
Performance	Can be slower for complex queries	Better performance with optimizations

Shuffle vs MapReduce

Aspect	Shuffle	MapReduce
Operation	Data reorganization across partitions	Data processing model for distributed computing
Efficiency	Can be costly due to data movement	Designed for batch processing with high I/O
Performance	Affects performance based on the amount of data movement	Optimized for large-scale data processing but less efficient for iterative tasks
Use Case	Used in Spark for data redistribution	Used in Hadoop for data processing tasks
Implementation	Integrated into Spark operations	Core component of the Hadoop ecosystem



Union vs Join

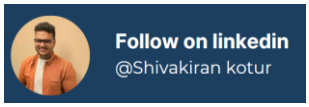
Aspect	Union	Join
Operation	Combines two DataFrames/RDDs into one	Combines rows from two DataFrames/RDDs based on a key
Data Requirements	Requires same schema for both DataFrames/RDDs	Requires a common key for joining
Performance	Generally faster as it does not require key matching	Can be slower due to key matching and shuffling
Output	Stacks data vertically	Merges data horizontally based on keys
Use Case	Appending data or combining datasets	Merging related data based on keys

Executor vs Driver

Aspect	Executor	Driver
Role	Executes tasks and processes data	Coordinates and manages the Spark application
Memory	Memory allocated per executor for data processing	Memory used for managing application execution
Lifecycle	Exists throughout the application execution	Starts and stops the Spark application
Tasks	Runs the tasks assigned by the driver	Schedules and coordinates tasks and jobs
Parallelism	Multiple executors run in parallel	Single driver coordinates multiple executors

Checkpointing vs Caching

Aspect	Checkpointing	Caching
Purpose	Provides fault tolerance and reliability	Improves performance by storing intermediate data
Storage	Writes data to stable storage (e.g., HDFS)	Stores data in memory or on disk (depends on storage level)
Use Case	Used for recovery in case of failures	Used for optimizing repeated operations
Impact	Can be more costly and slow	Generally faster but not suitable for fault tolerance
Data	Data is written to external storage	Data is kept in memory or disk storage for quick access

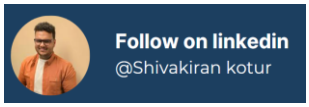


ReducebyKey vs AggregateByKey

Aspect	ReduceByKey	AggregateByKey
Operation	Combines values with the same key using a function	Performs custom aggregation and combinatory operations
Efficiency	More efficient for simple aggregations	Flexible for complex aggregation scenarios
Shuffling	Involves shuffling but can be optimized	Can be more complex due to custom aggregation
Use Case	Suitable for straightforward aggregations	Ideal for advanced and custom aggregations
Performance	Generally faster for simple operations	Performance varies with complexity

SQL Context vs Hive Context vs Spark Session

Aspect	SQL Context	Hive Context	Spark Session
Purpose	Provides SQL query capabilities	Provides integration with Hive for SQL queries	Unified entry point for Spark functionality
Integration	Basic SQL capabilities	Integrates with Hive Metastore	Combines SQL, DataFrame, and Streaming APIs
Usage	Legacy, less functionality	Supports HiveQL and Hive UDFs	Supports all Spark functionalities including Hive
Configuration	Less flexible and older	Requires Hive setup and configuration	Modern and flexible, manages configurations
Capabilities	Limited to SQL queries	Extends SQL capabilities with Hive integration	Comprehensive access to all Spark features



Broadcast Join Vs Shuffle Join

Aspect	Broadcast Join	Shuffle Join
Operation	Broadcasts a small dataset to all nodes	Shuffles data across nodes for joining
Data Size	Suitable for small datasets	Suitable for larger datasets
Efficiency	More efficient for small tables	More suited for large datasets
Performance	Faster due to reduced shuffling	Can be slower due to extensive shuffling
Use Case	Use when one dataset is small relative to others	Use when both datasets are large

Spark Context vs Spark Session

Aspect	Spark Context	Spark Session
Purpose	Entry point for Spark functionality	Unified entry point for Spark functionalities
Lifecycle	Created before Spark jobs start	Manages the Spark application lifecycle
Functionality	Provides access to RDD and basic Spark functionality	Provides access to RDD, DataFrame, SQL, and Streaming APIs
Configuration	Configuration is less flexible	More flexible and easier to configure
Usage	Older, used for legacy applications	Modern and recommended for new applications

Structured Streaming vs Spark Streaming

Aspect	Structured Streaming	Spark Streaming
Processing	Micro-batch and continuous processing	Micro-batch processing
API	SQL-based API with DataFrame/Dataset support	RDD-based API
Complexity	Simplified and high-level	More complex and low-level
Consistency	Provides stronger consistency guarantees	Can be less consistent due to micro-batches
Performance	Better performance with built-in optimizations	Can be slower for complex queries

Partitioning vs Bucketing

Aspect	Partitioning	Bucketing
Purpose	Divides data into multiple partitions based on a key	Divides data into buckets based on a hash function
Usage	Used to optimize queries by reducing data scanned	Used to improve join performance and maintain sorted data
Shuffling	Reduces shuffling by placing related data together	Reduces shuffle during joins and aggregations
Data Layout	Data is physically separated based on partition key	Data is organized into fixed-size buckets
Performance	Improves performance for queries involving partition keys	Enhances performance for join operations