APACHE SPARK DAY 1 LECTURE

Apache Spark Notes

1. What is Apache Spark?

Apache Spark is a **distributed computing framework** designed to efficiently process very large amounts of data. It allows for scalable, high-performance computation and is widely used for big data analytics and machine learning.

2. Why is Spark so Good?

1. **Efficient Use of RAM**
    - Spark leverages **in-memory computing** for faster data processing.
    - Traditional frameworks like Hive or MapReduce write data to disk at every step, making them resilient but incredibly slow.

- Spark minimizes disk I/O by writing to disk only when there isn't enough memory available (a process called **spilling to disk**).
- **Key Insight**: Avoid spilling to disk for optimal performance. When Spark uses RAM effectively, computations are fast and efficient.

2. **Storage Agnostic**
   - Spark allows **decoupling of storage and compute**, meaning it can read data from various sources, such as:
     - Relational databases
     - Data lakes
     - File systems
     - NoSQL databases (e.g., MongoDB)
3. **Large Developer Community**
   - Spark has a **thriving community**, which ensures consistent updates, support, and a vast pool of resources for developers.

---

3. When is Spark Not So Good?

1. **Knowledge Gap in the Organization**
   - If no one else in the company knows Spark, its adoption may face resistance or inefficiencies. Spark is not immune to the **"bus factor"** (risk of depending on a single expert).
2. **Pre-existing Infrastructure**
   - If your company already heavily relies on another big data framework or tool, it might not be practical to switch to Spark.

---

4. How Does Spark Work?

Apache Spark operates using three main components:

4.1. The Plan

- Represents the **logical execution plan** Spark creates to process data.

- It defines the sequence of operations (e.g., transformations and actions) required to achieve the desired outcome.

4.2. The Driver

- Acts as the **coach** of the operation.
- It is responsible for:
  - Translating the plan into tasks.
  - Distributing tasks to executors.
  - Monitoring the progress of the job.

4.3. The Executors

- These are the **players** on the team who perform the actual computations.
- Executors:
  - Receive tasks from the driver.
  - Perform data processing in parallel across the cluster.
  - Send the results back to the driver.

Spark as a Basketball Team Analogy

Think of Spark as a basketball team:

- **The Plan** is like the **playbook** or strategy the team follows.
- **The Driver** is the **coach**, creating the plan and instructing players on their roles.
- **The Executors** are the **players**, executing the plan and scoring points (processing data)

---

5. The Plan

The **Plan** in Spark refers to the sequence of transformations you define using **Python**, **Scala**, or **SQL**. It outlines the logical steps for processing data but does not execute them immediately.

Key Characteristics of the Plan:

1. **Lazy Evaluation**
    - The plan is evaluated **lazily**, meaning it doesn't execute until an action triggers it.
    - **What does "lazy" mean?**
        - The transformations you describe (e.g., filtering, mapping) are **recorded** but not run immediately.
        - Execution happens only when you perform an action like writing output or collecting data.

Triggering Execution (When Does Spark Execute the Plan?):

1. **When writing output**
    - If the plan involves writing data to a storage location (e.g., saving results to a file or database), execution is triggered.
2. **When part of the plan depends on the data itself**
    - For example, calling `DataFrame.collect()` to retrieve data or determine the next set of transformations.
    - Collecting data brings it back to the **driver** (the coach), which might adjust the plan or decide the next steps.

Analogy: The Basketball Team

- The **Plan** is like a basketball team's **playbook**, which outlines the strategy.
- A **shot** is taken when the team either:
    - Writes output (scoring points).
    - Brings data back to the coach (driver) for further instructions, similar to passing the ball back to the coach for a new play.

---

6. The Driver

The **Driver** is the central component in Spark that reads the plan and acts as the **coach**, deciding which play (plan) to execute and how to run it. It is critical for managing the overall execution of the job.

Key Responsibilities of the Driver:

1. Reads the **Plan** and determines:
    - When to start executing the job (breaking out of lazy evaluation).
    - How to perform joins between datasets.
    - The level of parallelism required for each step of the computation.
2. Oversees task distribution and monitors execution across the cluster.

Important Spark Driver Settings:

Setting	Description
`spark.driver.memory`	Specifies how much memory the driver can use. Increase this for complex jobs or when using `DataFrame.collect()` to avoid running out of memory (OOM).
`spark.driver.memoryOverheadFactor`	Defines the fraction of non-heap memory required by the driver. Defaults to 10%, but may need to be increased for complex jobs.

7. The Executors (Who Do the Actual Work)

The **Executors** are the workers in Spark that handle the actual data processing. They receive tasks from the driver, perform computations (e.g., transformations, filtering, aggregation), and return results.

Key Responsibilities of Executors:

- Execute tasks assigned by the driver.
- Perform transformations, filtering, and aggregations.
- Manage intermediate and final results, reporting back to the driver.

Important Spark Executor Settings:

Setting	Description
`spark.executor.memory`	Determines how much memory each executor can use. A low value may cause Spark to "spill to disk," significantly slowing down computations.
`spark.executor.cores`	Specifies how many tasks can run simultaneously on an executor. Default is 4; setting higher than 6 is generally not recommended.
`spark.executor.memoryOverheadFactor`	Indicates the percentage of memory reserved for non-heap tasks (e.g., UDFs, network buffers). Defaults to 10%, but may need to be increased for complex jobs.

Key Insight:

- **Engineering time is more expensive than cloud costs!** Allocating sufficient resources to the driver and executors ensures efficient computation and avoids costly delays.

---

8. Types of JOINs in Spark

In Spark, different types of JOINs are available for various data scenarios. Choosing the right type of JOIN significantly impacts performance.

8.1. Shuffle Sort-Merge JOIN

- **Description**:
  - The **least performant** join type but also the **most versatile**.
  - It works regardless of the data size and is the **default join strategy** since Spark 2.3.
- **When to Use**:
  - Best for cases where **both sides of the join are large**.
- **Key Points**:
  - Requires shuffling and sorting, making it slower and more resource-intensive.

8.2. Broadcast Hash JOIN

- **Description**:
  - A **highly efficient** join type for scenarios where one side of the join is small.
  - Instead of shuffling data, the smaller dataset is **broadcasted** to all executors, avoiding shuffle altogether.
- **When to Use**:
  - Ideal when the left side of the join is **small** (e.g., less than the broadcast threshold).
- **Configuration**:
  - Controlled by `spark.sql.autoBroadcastJoinThreshold`:
    - Default: 10 MB
    - Can go as high as **8 GB**, but values larger than 1 GB may cause **memory issues**.
- **Key Advantage**:
  - This is a **join without shuffle**, making it extremely fast for small datasets.

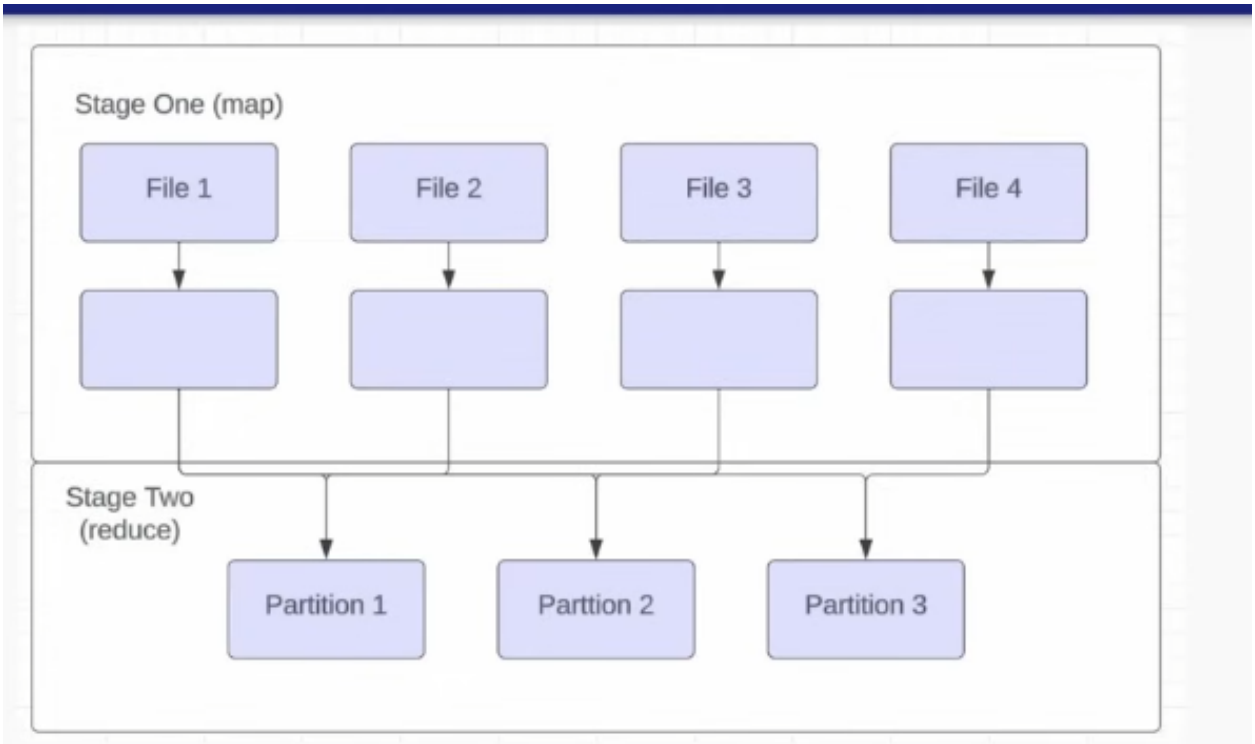
8.3. Bucket JOIN

- **Description**:
  - Another **join without shuffle**, but it requires pre-bucketing the data.
- **When to Use**:
  - Efficient when both datasets are **bucketed** on the same column, enabling direct joins without shuffling.

Key Insight:

- **Broadcast Hash JOIN** and **Bucket JOIN** are typically the **best options** for performance.
- Use **Shuffle Sort-Merge JOIN** only when necessary (e.g., when dealing with large datasets that cannot be optimized for broadcast or bucketing).

9. How Does Shuffle Work in Spark?



Shuffle is a **process of redistributing data** across partitions in Spark. It is often the most resource-intensive and least scalable part of Spark's operations. As the amount of data increases, shuffle becomes slower and may eventually become impractical for processing extremely large datasets (e.g., 20–30 TB per day).

Stages of Shuffle in Spark

1. **Stage One (Map):**
    - Spark processes the input data in parallel by reading files (e.g., File 1, File 2, File 3, File 4).
    - **Map operations** are performed, such as adding a new column, filtering data, or other transformations (e.g., `.select()`, `.withColumn()` in the DataFrame API).

- These map operations are **infinitely scalable** as they depend only on the number of input files.

2. **Stage Two (Reduce):**
   - Data is **redistributed** based on keys (e.g., `user_id` for a `groupBy` operation).
   - Spark creates partitions to organize the data:
     - Data is assigned to partitions based on a hash function (e.g., `user_id % numPartitions`).
     - For example:
       - If the remainder is `0`, data goes to **Partition 1**.
       - If the remainder is `1`, data goes to **Partition 2**, and so on.
   - Shuffle ensures all related data is placed in the same partition for further operations like grouping or joining.

Example Scenarios

Scenario 1: GroupBy Operation

- Input: A table with 4 files, where the data needs to be grouped by `user_id`.
- Process:
  1. Data from all files (File 1, File 2, etc.) is distributed to partitions based on the hash of `user_id`.
  2. Related data is collected in the same partition for processing.

Scenario 2: Join Operation

- Input: Two tables (Files 1 & 2 for Table A, Files 3 & 4 for Table B).
- Process:
  1. Shuffle redistributes data across partitions so that matching `user_id`s from both tables are in the same partition.
  2. Once the data is in the right partitions, Spark performs a **shuffle sort-merge join** to combine the tables.

Broadcast Join vs. Shuffle

1. **Shuffle (Sort-Merge Join):**
   - Data is shuffled across the cluster, requiring significant resources and time.
   - Best for large datasets on both sides of the join.
2. **Broadcast Join:**
   - Instead of shuffling data, Spark broadcasts the smaller dataset (e.g., Files 3 & 4) to all executors.
   - Executors can perform comparisons locally without shuffling.
   - Ideal for scenarios where one dataset is **small enough** (default size: ≤10 MB, configurable).

Visualization

The uploaded diagram demonstrates how Spark processes files in two stages:

1. **Stage One (Map)**: Reads files and performs transformations.
2. **Stage Two (Reduce)**: Redistributes data across partitions for operations like `groupBy` or `join`.

9.1 Shuffle

Shuffle Partition and Parallelism

- **Shuffle partition** and **parallelism** are closely related and often the same concept in Spark.
- `spark.sql.shuffle.partition` and `spark.default.parallelism` are usually identical, but with some exceptions:
  - When using the **RDD API directly**, `spark.default.parallelism` is the setting to use.
  - When using higher-level APIs like **DataFrame**, **Spark SQL**, or **Dataset**, `spark.sql.shuffle.partition` is the relevant configuration.

Key Recommendation

- Avoid using the RDD API in 99% of cases.
  - This is guidance from **Databricks** and **Spark** teams.
- High-level APIs are more efficient and user-friendly.

Role of Shuffle

- Shuffle determines the number of partitions after operations like **join** or **group by**.

Is Shuffle Good or Bad?

When Shuffle is Good

- **Low to Medium Volume**:
  - Shuffle works well and is efficient.
  - Example: 1 TB of shuffle data is manageable and provides good performance.
- **Benefits**:
  - Simplifies processing.
  - Suitable for most use cases with reasonable data volume.

When Shuffle Becomes Challenging

- **High Volume (>10 TB)**:
  - Shuffle can be extremely painful.
  - Example: At Netflix, shuffle negatively impacted the **IP enrichment pipeline**.
- **Real-Life Example**:
  - At Netflix, a pipeline processing 100 TB per hour required joining datasets:
    - Dataset 1: Network requests to Netflix.
    - Dataset 2: IP mappings to microservices.

- Initially, the IP lookup table (a few GBs) enabled a **broadcast join**, which performed well.
- When migrating to IPv6:
  - The IP address space increased 10,000x compared to IPv4.
  - **Broadcast join** was no longer feasible, necessitating a **shuffle join**, which failed to scale.

Another Case: Feature Generation at Facebook

- Use Case: Joining two large tables for notification data:
  - Table 1: 10 TB.
  - Table 2: 50 TB.
- Impact:
  - The pipeline consumed **30% of compute resources** for notifications.
  - This was inefficient and unsustainable.

Solutions to Shuffle Problems

1. Bucketing for Efficient Joins

- **Bucketing** eliminates the need for shuffle in certain scenarios:
  - Example: Facebook's notification dataset pipeline.
  - **Solution**:
    - Both left and right tables were bucketed by **User ID** into 1024 buckets.
    - This ensured:
      - Data for each User ID existed in pre-determined files.
      - Joins could directly match corresponding buckets, avoiding shuffle.
    - Significant performance improvement.

2. How Bucketing Works

- Data is written to buckets based on a **hashing function**:
  - Example: Using User ID modulo the number of buckets.
  - This guarantees relevant data is in the correct bucket.

3. Aligning Buckets

- If two tables have **different bucket counts**:
  - Ensure one is a **multiple of the other** to still achieve efficient bucket joins.
- **Best Practice**:
  - Always bucket tables using powers of 2 (e.g., 2, 4, 8, 16).
  - The number of buckets depends on the data volume.

Key Lessons

- **Shuffle Performance**:
  - Works well for low-to-medium data volumes.
  - Can cause inefficiency for high-volume pipelines.
- **Bucketing**:
  - A powerful technique to avoid shuffle and boost performance.
  - Requires careful planning of bucket counts and data organization.
- **Always Optimize**:
  - Choose configurations based on data size and pipeline requirements.

---

10. How to Minimize Shuffle at High Volumes

1. **Use Bucketing**:
   - Bucket the data if multiple **joins** or **aggregations** are happening downstream.
   - Spark allows bucketing to **reduce** or **eliminate shuffle** during joins.
2. **Bucket Joins**:

- **Advantages**:
  - Highly efficient for specific use cases.
- **Drawbacks**:
  - The initial parallelism is limited by the **number of buckets**.
  - Bucket joins only work if the **bucket counts** of the two tables are multiples of each other.

3. **Best Practices**:
   - Always use **powers of 2** for the number of buckets (e.g., 2, 4, 8, 16, etc.).

---

11. Shuffle and Skew

What is Skew?

- Skew occurs when some partitions have significantly more data than others, leading to inefficiencies in processing.

Why Does Skew Happen?

1. **Insufficient Partitions**:
   - When the number of partitions is too low, data is unevenly distributed across the partitions.
2. **Nature of the Data**:
   - Some data inherently causes skew due to its distribution.
   - **Example**:
     - Beyoncé, being a highly popular figure, receives far more notifications compared to the average Facebook user. This leads to a disproportionate amount of data being assigned to the partition processing her notifications.

---

12. How to tell if your data is skewed ?

- Most common is a job getting to 99% taking forever and failing

- Another , more scientific way is to do a box and whiskers plot of the data to see if there's any extreme outliers

13. Ways to deal with Skew

- To solve this problem is but its only availavle in spark 3+ is you need to use adaptive query execution
    - Set spark.sql.adaptive.enabled = True
- Salting the GROUP BY - best option before Spark 3
    - GROUP BY A random number , aggregate + GROUP by AGAIN
    - Be carefull with things like AVG - break it into SUM and COUNT and divide !

```
df.withColumn("salt_random_column", (rand * n).cast(IntegerType))
  .groupBy(groupByFields, "salt_random_column")
  .agg(aggFields)
  .groupBy(groupByFields)
  .agg(aggFields)
```

---

1. How to Look at Spark Query Plans

- Use the `explain()` method on your DataFrames:
    - This will display the **join strategies** that Spark plans to use during execution.

---

2. How Can Spark Read Data?

From the Lake:

- Sources include:
    - Delta Lake
    - Apache Iceberg
    - Hive Metastore

From an RDBMS:

- Examples:
  - Postgres
  - Oracle
  - Other relational databases

From an API:

- Spark can make **REST API calls** and convert the responses into data.
  - **Caution**: This operation typically runs on the **Driver**, so it can be resource-intensive.

From a Flat File:

- Formats like **CSV** and **JSON** are supported.

---

3. Spark Output Datasets

- Output datasets should almost always be **partitioned by "date"**:
  - The partition date corresponds to the **execution date** of the pipeline.
  - In **Big Tech**, this practice is often referred to as **“ds partitioning”** (dataset partitioning).
