03 Spark Internal Workings

Spark Internal Workings

1. Introduction to Apache Spark

Apache Spark is a powerful, open-source distributed processing system designed for speed and ease of use. It excels at handling large-scale data processing tasks.

Key Characteristics:

- **In-Memory Processing:** Spark's primary advantage is its ability to perform computations in memory, significantly reducing the disk I/O bottlenecks common in other systems like Hadoop MapReduce. This leads to much faster processing speeds.
- **Parallel Processing & Scalability:** Spark is designed for parallelism. It distributes data and computations across multiple nodes in a cluster. As data volume grows, you can scale Spark by adding more worker nodes.
- **Fault Tolerance:** Spark RDDs (Resilient Distributed Datasets) and DataFrames/Datasets can reconstruct lost data partitions, providing fault tolerance.
- **Unified Analytics Engine:** Spark supports various workloads, including batch processing, interactive queries (SQL), real-time stream processing, machine learning, and graph processing.

2. Spark Architecture: Core Components

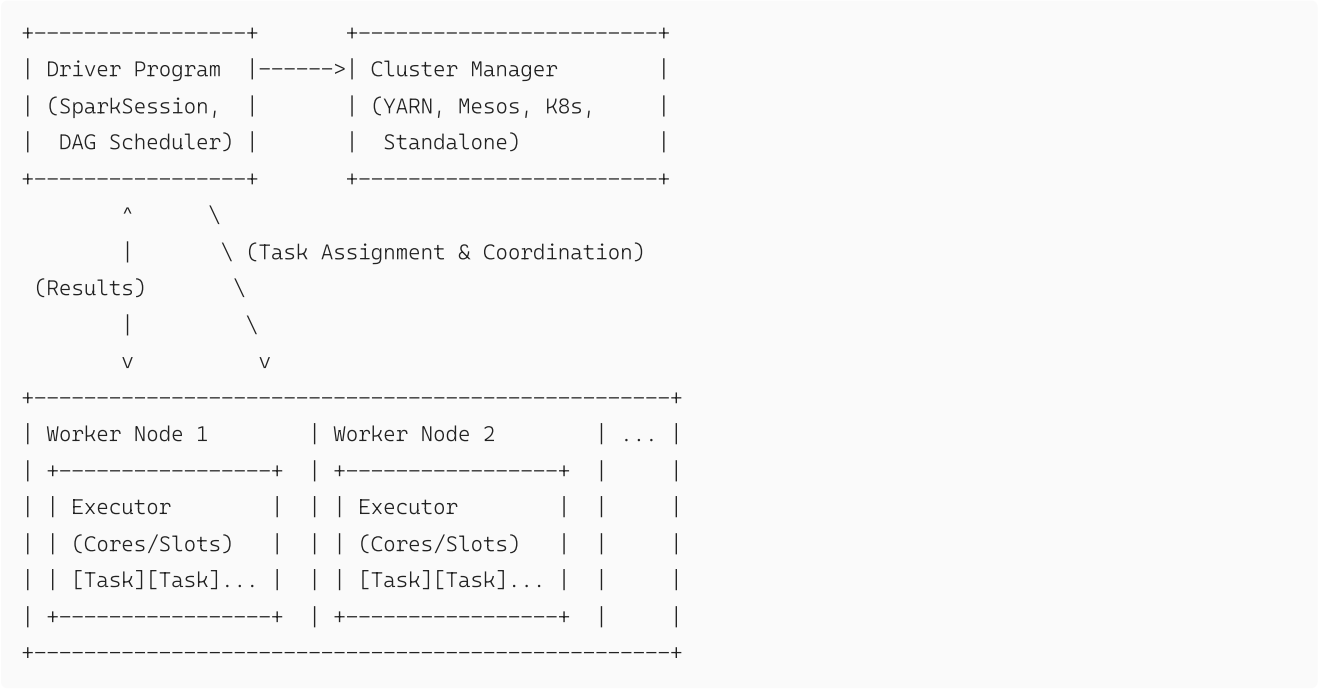
A Spark application runs as a set of independent processes on a cluster, coordinated by the `SparkContext` in your main program (the **driver program**).

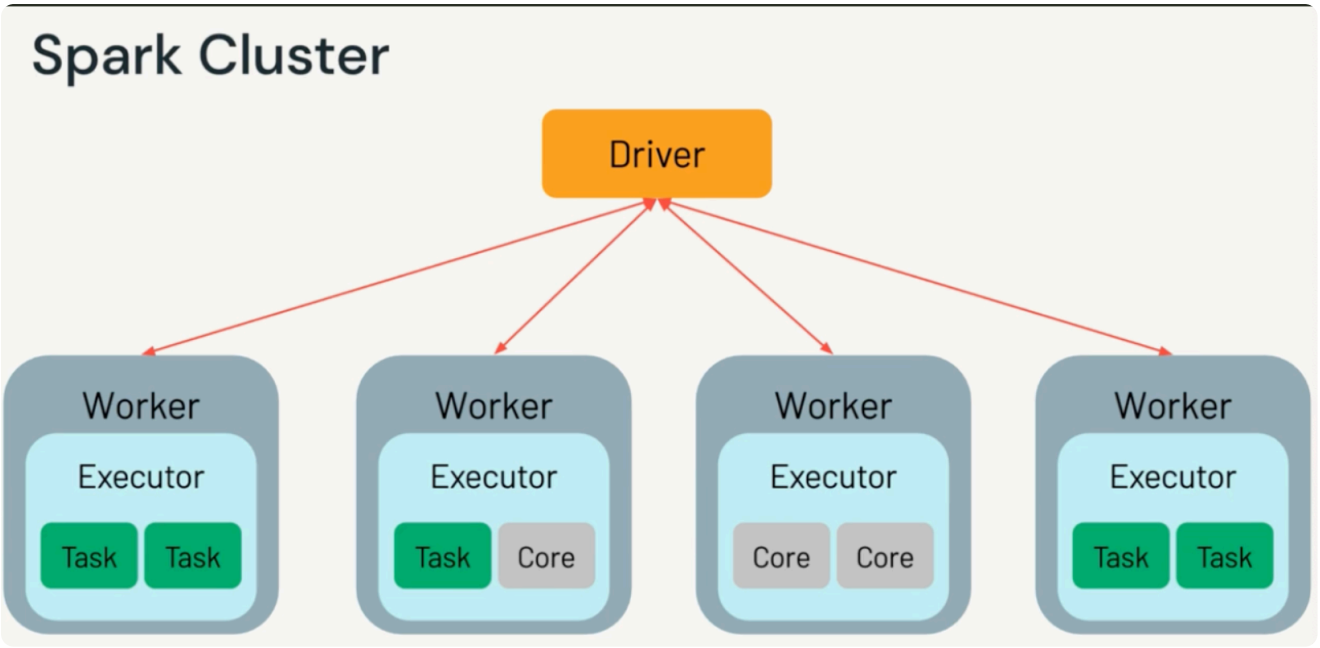
The main components are:

- **Driver Program:**
  - The process running the `main()` function of your application and creating the `SparkSession` (or `SparkContext`).
  - **Responsibilities:**
    - Instantiates the `SparkSession`.
    - Analyzes, optimizes, and transforms user code (DataFrames/SQL operations) into a Directed Acyclic Graph (DAG) of tasks.
    - Schedules these tasks and distributes them to worker nodes.
    - Tracks the execution of tasks.
    - Returns results to the user or writes output.
- **Cluster Manager:**
  - An external service responsible for acquiring and managing resources on the cluster for Spark applications.
  - **Common Cluster Managers:**
    - **Standalone:** A simple cluster manager included with Spark.

    - **Apache Mesos:** A general-purpose cluster manager.
    - **Hadoop YARN (Yet Another Resource Negotiator):** The resource manager in Hadoop.
    - **Kubernetes:** An open-source container orchestration system.
- **Worker Nodes:**
    - Nodes in the cluster that run the application code. Each worker node hosts one or more Executors.
- **Executors:**
    - Processes launched on worker nodes at the beginning of a Spark application.
    - **Responsibilities:**
        - Execute tasks assigned by the driver.
        - Store data partitions in memory or on disk (caching).
        - Return computation results to the driver.
    - Each executor has a certain number of **cores** (or **slots**), determining how many tasks it can run concurrently.
- **Tasks:**
    - The smallest unit of work in Spark.
    - The driver sends tasks to executors for execution.
    - Typically, one task processes one partition of data on one core. (1 task : 1 partition : 1 core).

Simplified Cluster Diagram:





3. SparkSession: The Entry Point

Since Spark 2.0, `SparkSession` is the unified entry point for interacting with Spark functionalities. It subsumes previous contexts like `SparkContext`, `SQLContext`, `HiveContext`, and `StreamingContext`.

- A `SparkSession` instance is typically created as `spark` in Spark shells and Databricks notebooks.
- While `SparkSession` is the entry point for DataFrame and Dataset APIs, a `SparkContext` still underlies it and can be accessed via `spark.sparkContext`. An application typically has one active `SparkContext`.

Key `SparkSession` Methods:

Method	Description
`sql(query)`	Returns a DataFrame representing the result of the given SQL query.
`table(tableName)`	Returns the specified table as a DataFrame.
`read`	Returns a `DataFrameReader` used to read data from various sources (CSV, JSON, Parquet, JDBC, etc.) into a DataFrame.
`range(...)`	Creates a DataFrame with a column containing elements in a range, useful for testing or generating sequences.
`createDataFrame(...)`	Creates a DataFrame from an existing collection (e.g., a list of tuples or rows), often used for testing.
`sparkContext`	Accesses the underlying `SparkContext`.
`stop()`	Stops the SparkSession and releases resources.

Example Usage:

```
# spark is the SparkSession variable

# Reading data from a Parquet file
df_parquet = spark.read.parquet("path/to/your/data.parquet")

# Reading data from a table
df_table = spark.table("my_database.my_table")

# Executing a SQL query
df_sql_result = spark.sql("SELECT column1, COUNT(*) FROM my_table GROUP BY column1")

# Creating a DataFrame from a list of tuples
data = [("Alice", 1), ("Bob", 2), ("Charlie", 3)]
```

```
columns = ["name", "id"]
df_from_list = spark.createDataFrame(data, columns)
```

4. Spark APIs

Spark provides a rich set of APIs for different use cases:

- **Spark SQL & DataFrames/Datasets:**
  - Allows querying structured and semi-structured data using SQL or the expressive DataFrame/Dataset API (available in Scala, Java, Python, R).
  - DataFrames are distributed collections of data organized into named columns, conceptually similar to tables in a relational database.
  - Datasets (Scala/Java) are an extension of DataFrames providing type safety and object-oriented programming benefits.
- **Spark Structured Streaming:**
  - A scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
  - Allows processing of live data streams using the same DataFrame/Dataset API.
- **MLlib (Machine Learning Library):**
  - Provides a suite of machine learning algorithms and tools for tasks like classification, regression, clustering, and dimensionality reduction.
- **GraphX (and GraphFrames):**
  - An API for graph computation and analysis. GraphX operates on RDDs, while GraphFrames are built on top of DataFrames for more optimized graph processing.

5. Computation in Spark: The Execution Model

Spark's execution model is based on lazy evaluation and the construction of a Directed Acyclic Graph (DAG).

- **Resilient Distributed Datasets (RDDs):**
  - The fundamental data structure in Spark (though often abstracted by DataFrames/Datasets).
  - Immutable, partitioned collections of records that can be operated on in parallel.
  - RDDs track their lineage (the sequence of transformations used to create them), which allows for fault tolerance.
- **Transformations:**
  - Operations on RDDs/DataFrames that create a new RDD/DataFrame (e.g., `map`, `filter`, `select`, `groupBy`).
  - **Lazy Evaluation:** Transformations are not executed immediately. Spark builds up a DAG of transformations.
- **Actions:**
  - Operations that trigger computation and return a result to the driver program or write data to an external storage system (e.g., `count`, `collect`, `save`, `show`).
  - When an action is called, Spark evaluates the DAG.
- **Lineage (DAG):**
  - Spark records the sequence of transformations applied to data. This lineage forms a DAG.
  - Each node in the DAG represents an RDD/DataFrame, and edges represent transformations.
  - The DAG allows Spark to optimize the execution plan and reconstruct lost partitions.

```
Example Lineage:
1. read (source DataFrame)
   |
2. select (transformed DataFrame, depends on #1)
   |
3. filter (transformed DataFrame, depends on #2)
   |
4. groupBy (transformed DataFrame, depends on #3)
   |
5. agg (transformed DataFrame, depends on #4)
   |
6. write (Action - triggers computation)
```

- **Jobs, Stages, and Tasks:**
  - **Job:** An action triggers a job. A job corresponds to the computation of a complete DAG (or part of it) needed for that action.
  - **Stage:** Jobs are divided into stages. Stages are sets of tasks that can be executed together without shuffling data. A new stage is typically created at a "shuffle boundary" (wide transformation like `groupByKey`, `reduceByKey`, `join` where data needs to be redistributed across partitions).
  - **Task:** The smallest unit of execution. Each task runs on a single executor core and processes a single partition of data for a specific stage.
- **Parallelism Levels:**
  1. **Among Executors:** Work is split among the executors in the cluster. Each executor processes a subset of the data partitions.
  2. **Within Executors:** Each executor has multiple cores (slots), allowing it to execute multiple tasks concurrently, each on a different data partition.

6. Query Optimization: Catalyst Optimizer & Tungsten

Spark SQL uses the **Catalyst Optimizer** to translate DataFrame/SQL queries into efficient physical execution plans.

Catalyst Optimizer Phases:

1. **Analysis:**
   - The initial query (Unresolved Logical Plan) is parsed.
   - Table and column names are resolved against a catalog (metadata).
   - An Abstract Syntax Tree (AST) is created, resulting in a Logical Plan.
2. **Logical Optimization:**
   - A set of rule-based optimizations are applied to the Logical Plan (e.g., predicate pushdown, constant folding, projection pruning).
   - Cost-Based Optimization (CBO) can also be used if statistics are available, to choose the most efficient join orders or types. This results in an Optimized Logical Plan.
3. **Physical Planning:**
   - The Optimized Logical Plan is translated into one or more Physical Plans.
   - Spark considers different strategies for operations (e.g., different join algorithms like Broadcast Hash Join, Sort Merge Join).
   - A cost model is used to select the best Physical Plan.
4. **Code Generation (Project Tungsten):**
   - The selected Physical Plan is compiled into highly efficient Java bytecode that runs on each executor.

- **Whole-Stage Code Generation:** A key feature of Tungsten. It collapses entire stages of a query into a single, optimized Java function, minimizing virtual function calls and leveraging CPU registers for intermediate data. This often operates directly on binary data, improving memory and CPU efficiency. The output is essentially RDD code.

Viewing Execution Plans: You can inspect the logical and physical plans using `DataFrame.explain(extended=True)`:

```
df.filter("age > 30").groupBy("department").count().explain(extended=True)
```

This will show:

- `== Parsed Logical Plan ==`
- `== Analyzed Logical Plan ==`
- `== Optimized Logical Plan ==`
- `== Physical Plan ==`

7. Adaptive Query Execution (AQE)

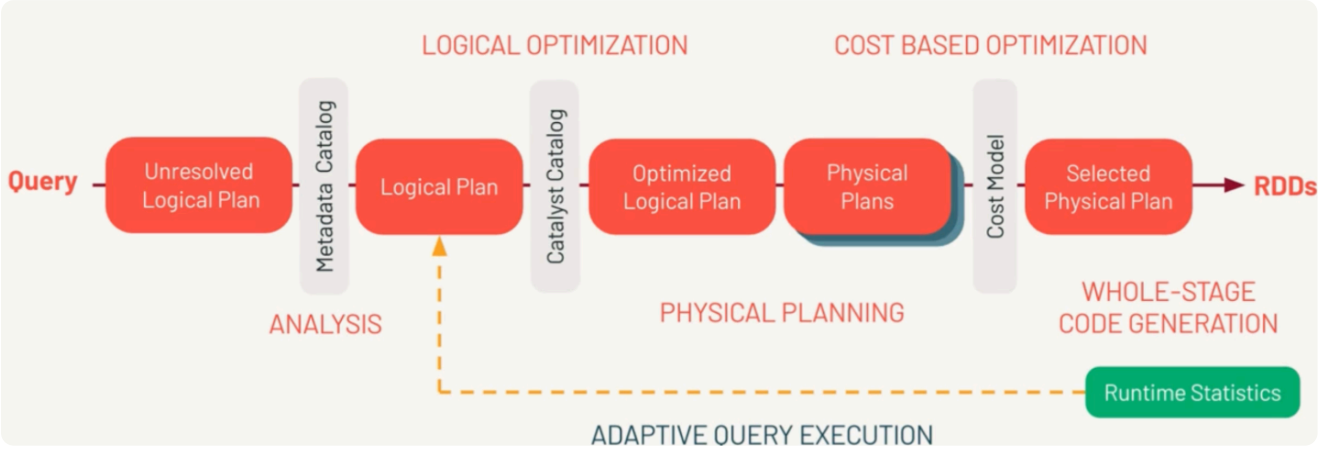
AQE is a query re-optimization technique that occurs during query execution, leveraging runtime statistics. It is enabled by default since Spark 3.0 (the document says 3.2.0, but core features started appearing and being enabled by default around 3.0; 3.2.0 further solidified it).

Key AQE Optimizations:

- **Dynamically Coalescing Shuffle Partitions:** Merges small adjacent shuffle partitions into larger, more optimal ones, reducing task overhead. This is especially useful when the number of shuffle partitions (`spark.sql.shuffle.partitions`) is too high for the actual data size post-shuffle.
- **Dynamically Switching Join Strategies:** Can change join strategies based on runtime statistics. For example, if one side of a join turns out to be small enough at runtime, AQE can convert a Sort Merge Join to a more efficient Broadcast Hash Join.
- **Dynamically Optimizing Skew Joins:** Detects and mitigates data skew in joins by splitting skewed partitions into smaller sub-partitions, allowing for better parallelism.

AQE helps improve performance without manual tuning, adapting to the actual data characteristics encountered during execution. The umbrella configuration is `spark.sql.adaptive.enabled` (default: `true`).

AQE in the Optimization Flow:



8. Shuffle, Partitioning, and Caching

Understanding these concepts is crucial for performance tuning.

8.1. Partitioning

Data in Spark is divided into partitions, which are the basic units of parallelism.

- **Spark Partition (In-Memory/In-Flight):**
  - A logical chunk of data within an RDD or DataFrame residing in an executor's memory (or spilled to disk if memory is insufficient). Tasks operate on these partitions.
  - **Initial Partitioning (Reading from Files):**
    - When reading files, Spark aims to create partitions of a size close to `spark.sql.files.maxPartitionBytes` (default 128MB).
    - The number of partitions is roughly `total_data_size / spark.sql.files.maxPartitionBytes`.

    - `spark.sql.files.openCostInBytes` (default 4MB) also influences this; files smaller than this might be coalesced.
    - The default parallelism (`spark.default.parallelism`) can also play a role, especially for RDD-based operations.
  - **Checking Number of Partitions:** `df.rdd.getNumPartitions()` or `df.rdd.partitions.size`.
  - **Partition Sizing:**
    - **Too Few, Too Large Partitions:** Can lead to OutOfMemoryError (OOM), long garbage collection (GC) pauses, and underutilization of cluster resources (some cores idle).
    - **Too Many, Too Small Partitions:** Can lead to excessive task scheduling overhead, and inefficient processing for some operations.
    - **Ideal Size:** Often recommended to be between 100MB to 200MB, but this is a heuristic. The goal is to keep tasks reasonably short (seconds to a few minutes) and ensure data fits in executor memory. Aim for 2-4 partitions per CPU core in your cluster for optimal parallelism.
- **Disk Partition (At Rest / File System Partitioning):**
  - Refers to how data is organized into directories and files on a distributed file system (like HDFS or S3), often using a Hive-style partitioning scheme (e.g., `/data/year=2023/month=05/day=30/file.parquet`).
  - This allows for "predicate pushdown" or "partition pruning," where Spark reads only the necessary directories based on filter conditions, significantly reducing I/O.
  - This is distinct from Spark's in-memory partitioning.

Changing Number of Partitions:

- `repartition(N)`:
  - Can increase or decrease the number of partitions to exactly `N`.
  - Always involves a **full shuffle** of the data, which is expensive.
  - Useful for increasing parallelism or when data needs to be evenly distributed across a specific number of partitions (e.g., before writing to a fixed number of files).
  - Can take a column expression for hash partitioning: `df.repartition(col("user_id"))`.
- `coalesce(N)`:
  - Only reduces the number of partitions to `N`.
  - Avoids a full shuffle by merging existing partitions on the same executor. It's a narrow transformation if `N` is less than the current number of partitions.
  - More efficient than `repartition()` when decreasing partitions.
  - Does not guarantee evenly sized partitions.

8.2. Shuffle

A shuffle is the process of redistributing data across partitions, often between executors. It's one of the most expensive operations in Spark.

- **When Shuffles Occur:**
  - During **wide transformations** where data with the same key needs to be brought together on the same partition/executor.
  - Examples: `groupByKey`, `reduceByKey`, `aggregateByKey`, `join` (except broadcast joins), `sortByKey`, `repartition`.
- **Shuffle Process:**
  1. **Shuffle Write:** Tasks in the preceding stage write shuffle data (intermediate map output) to local disks on their respective executors.
  2. **Shuffle Read:** Tasks in the subsequent stage fetch the relevant shuffle data blocks from the executors where they were written.
- **Impact of Shuffles:**
  - Involves disk I/O, data serialization/deserialization, and network I/O.
  - Can be a major performance bottleneck.
- `spark.sql.shuffle.partitions`:
  - Default is 200.
  - Determines the number of partitions for data after a shuffle (for SQL/DataFrame operations).

- Tuning this value is critical. If too low, partitions might be too large. If too high, many small tasks can cause overhead. AQE helps mitigate issues with this setting.

Minimizing Shuffles:

- Use `reduceByKey` or `aggregateByKey` instead of `groupByKey` when possible, as they perform partial aggregation on the map side.
- Use broadcast joins for small tables.
- Design data schemas and queries to favor narrow transformations.
- Use appropriate partitioning strategies (e.g., `repartition` by join keys before multiple joins on the same keys).

8.3. Caching (Persistence)

Spark allows you to persist (or cache) RDDs/DataFrames in memory (and/or on disk) across operations.

- `df.cache()`: A shorthand for `df.persist(StorageLevel.MEMORY_AND_DISK)`.
  - `MEMORY_AND_DISK`: Stores partitions in memory. If memory is insufficient, spills excess partitions to disk.
- `df.persist(storageLevel)`: Allows specifying different storage levels:
  - `MEMORY_ONLY`: Store as deserialized Java objects in memory. If not enough memory, partitions are not cached (or recomputed on demand).
  - `MEMORY_ONLY_SER`: Store as serialized Java objects (more space-efficient).
  - `MEMORY_AND_DISK_SER`: Like `MEMORY_ONLY_SER`, but spills to disk.
  - `DISK_ONLY`: Store only on disk.
  - Levels can also be replicated (`_2`).
- **Benefits of Caching:**
  - Speeds up iterative algorithms (e.g., machine learning) that access the same dataset multiple times.
  - Useful for interactive data exploration.
  - Can save recomputation if a node fails (if replicated).
- **When to Use Cache:**
  - DataFrames that are frequently accessed or re-used in subsequent transformations and actions.
  - Intermediate results in iterative algorithms.
  - Relatively small DataFrames that fit comfortably in memory.
- **When NOT to Use Cache (or use with caution):**
  - DataFrames that are too large to fit in cluster memory (can lead to excessive spilling and GC pressure).
  - DataFrames used only once.
  - Caching can sometimes interfere with Catalyst optimizer's ability to perform certain optimizations like filter pushdown through the cached plan.
- **Lazy Evaluation:**
  - `cache()` or `persist()` are themselves transformations and are lazy.
  - The DataFrame is not actually cached until an action is performed on it (e.g., `count()`, `show()`, `write()`).
  - An action like `take(1)` might only cache the first partition if Spark determines others are not needed. To fully cache, an action that processes all data (like `count()`) is needed.
- **Unpersisting:**
  - It's crucial to release cached data when no longer needed using `df.unpersist()`. This frees up memory for other operations.

Cache and Shuffle Interaction: If a DataFrame resulting from a shuffle is cached, subsequent actions can read from the cache, avoiding re-execution of the shuffle write and preceding stages.

```
Stage 1 (e.g., read, filter)
   |
groupBy (Shuffle Write) --> Writes shuffle files
   |
Stage 2 (e.g., aggregate) --> Reads shuffle files
   |
df.cache() --> Marks for caching
   |
action1 (e.g., count()) --> Computes, shuffle files written, result cached
```

```
    |
action2 (e.g., show()) --> Reads from cache (and potentially shuffle files if not fully cached or if
some partitions were evicted)
```

If `df.cache()` is applied *before* the `groupBy`, the result of `Stage 1` would be cached. If applied *after* `groupBy` and *before* `action1`, the result of `Stage 2` (post-shuffle) is cached.

9. Key Takeaways for Performance

- **Understand Your Data:** Size, skew, and structure.
- **Partitioning is Key:** Ensure an appropriate number and size of partitions. Use file system partitioning for predicate pushdown.
- **Minimize Shuffles:** Design transformations carefully. Use broadcast joins where applicable.
- **Use Caching Wisely:** Cache frequently accessed, smaller DataFrames. Don't forget to `unpersist()`.
- **Leverage AQE:** It handles many dynamic optimizations, but understanding the underlying principles is still important.
- **Monitor Spark UI:** The Spark UI is an invaluable tool for understanding job execution, identifying bottlenecks, and debugging performance issues. Pay attention to stage durations, task times, shuffle read/write sizes, and GC time.
- **Choose the Right API:** DataFrames/SQL are generally more optimized than raw RDDs for structured data due to Catalyst and Tungsten.

These notes provide a comprehensive overview of Spark's internal workings. Continuous learning and experimentation with your specific workloads will further enhance your understanding and ability to optimize Spark applications.