Cache Eviction

In Apache Spark, cache eviction refers to the process of removing cached data from memory when it becomes full or needs space for new data. Spark uses an in-memory storage layer called the RDD cache, which allows for storing RDDs (Resilient Distributed Datasets) in memory for faster access in iterative processes.

Here's how cache eviction works in Spark:

1. Caching RDDs: When you cache an RDD, Spark stores it in memory (and optionally on disk) across the cluster nodes. This allows Spark to reuse the data without recomputing it in future actions or transformations, which improves performance.

2. Memory Limitations: The memory allocated to cache RDDs is limited, so if you try to cache more data than available memory, Spark may not have enough space.

3. Eviction Policy: When memory reaches its limit, Spark evicts older RDDs based on an LRU (Least Recently Used) policy. This means that the least recently accessed RDD partitions will be removed to make room for new ones.

4. Disk Spill: If you configure caching with `MEMORY_AND_DISK`, Spark will write evicted RDD partitions to disk when memory is insufficient, ensuring data availability even when it can't fit entirely in memory.

Cache eviction is crucial for managing memory resources effectively in Spark, especially in iterative applications or those with complex data workflows.

1. JVM Heap and Python Memory

- JVM Heap: This is the memory allocated to each Spark executor running on the JVM. The `spark.executor.memory` parameter configures this memory limit. The JVM heap holds various memory components required for Spark's operations, such as storage and shuffle operations.

- Python Memory: For applications using PySpark (Spark with Python), Python memory is separate from JVM memory, set by `spark.python.worker.memory`. This memory is used by Python processes that interact with Spark's JVM processes.

2. Safe Memory (90% of Heap)

- Within the JVM Heap, 90% is labeled as "Safe," a fraction that Spark considers safe for storage and shuffle memory. This boundary is controlled by the `spark.storage.safetyFraction` parameter.
- 10% of the heap is reserved for other internal purposes, such as metadata and other system operations, which helps avoid OutOfMemory (OOM) errors.

3. Storage Memory (60% of Safe)

- Storage memory is allocated from the "Safe" memory for caching RDDs, DataFrames, and other intermediate data. By default, 60% of the "Safe" memory is reserved for storage, controlled by the `spark.storage.memoryFraction` parameter.
- If Spark operations, such as iterative processing or repeated transformations, require caching, the data is stored here.

4. Unroll Memory (20% of Storage)

- Unroll memory is a subset of storage memory specifically used for unrolling large datasets into storage memory. Unrolling is the process of gradually caching an RDD partition before it is fully stored. This buffer helps avoid memory pressure and ensures the dataset fits into the cache.
- By default, 20% of storage memory is reserved for unroll, configurable with `spark.storage.unrollFraction`.

5. Shuffle Memory (20% of Safe)

- Shuffle memory is a part of the "Safe" memory allocated for shuffle operations, which involve sorting, grouping, and aggregating data between stages of a Spark job.
- The shuffle memory fraction is controlled by `spark.shuffle.memoryFraction`, and its default is 20% of the "Safe" memory.

In Spark, managing memory allocation for these components is essential to ensure efficient operation and prevent memory-related failures, such as OOM errors. Tuning these fractions is necessary for optimizing performance, especially in memory-intensive jobs.

In the right side of the diagram, labeled as "Python Memory," the concept of *Safe* memory means the amount of memory available for Python worker processes in a PySpark application.

Here's what it means in context:

- **Python Memory (512MB)**: This section represents the memory allocated specifically for Python workers when running PySpark (Spark with Python) applications. Spark creates separate Python processes to execute Python code, which interact with Spark's JVM processes for data processing.
- **Safe (90% of Total)**: In the Python Memory area, 90% of the allocated memory (e.g., if 512MB is allocated, then ~460MB is "safe") is designated as "Safe" memory. This term is a bit different here than in the JVM context.
    - In this case, "Safe" refers to the memory that Python can safely use without causing memory pressure or risking out-of-memory errors.
    - The remaining 10% of Python memory is left unreserved as a buffer for system operations or unexpected spikes in memory demand, which helps prevent the Python process from crashing due to memory overuse.
    -

This setup ensures that Python memory is managed separately from JVM memory and has its own safeguard to keep PySpark processes stable.

The architecture of a Spark application running on a YARN cluster, detailing the roles of the **Client Node** (where the Spark Driver resides) and **Worker Nodes** (where Spark Executors reside).

1. Client Node

- **Driver JVM**: The Driver JVM is where the Spark driver program runs. It holds the Spark Context, which coordinates all distributed processing in the Spark application.
- **Spark Context**: This manages and schedules tasks across the cluster, tracks their status, and maintains metadata. It's controlled by the spark.driver.memory parameter, which sets the maximum memory available for the driver.
- The Driver JVM is responsible for:
  - Splitting the work into tasks and distributing them to the executors.
  - Aggregating results from executors.
  - Managing dependencies and task scheduling.

2. Worker Nodes

- Each Worker Node in the YARN cluster has:
  - **YARN Node Manager**: Manages resources and monitors application containers (executors) running on that node.
  - **HDFS Datanode**: Provides data storage as part of the Hadoop Distributed File System (HDFS) cluster. Spark reads and writes data from/to HDFS for distributed storage.
- **Node Memory Pool**: This is the total memory allocated to each node, set by the yarn.nodemanager.resource.memory-mb parameter. It includes memory for all executors running on that node.
- Each worker node can have multiple **Executor JVMs** running on it, based on the resources available and configuration.

3. Executor JVMs

- Executors are JVM processes that run on the worker nodes, controlled by spark.executor.memory (for memory) and spark.executor.cores (for CPU cores) parameters.
- Each Executor JVM is responsible for:
  - Executing multiple **Tasks** in parallel, where each task is a unit of work on a partition of the data.

    - Storing intermediate data (cache) and providing it to the driver as needed.
    - Communicating with the driver for task completion status.
- Executors can have multiple tasks running simultaneously, each requiring CPU resources defined by spark.task.cpus. The number of cores available to an executor limits the number of tasks it can run concurrently.

4. Tasks within Executors

- Each Task is a single-threaded unit of work executed within an executor. Tasks operate on data partitions and perform operations like transformations and actions on RDDs or DataFrames.
- Tasks are assigned a specific number of CPU cores via the spark.task.cpus parameter.

Summary

- **Client Node** (Driver): Manages and coordinates the execution, sends tasks to executors, and gathers results.
- **Worker Nodes**: Host the executors, which carry out the actual data processing. Each worker node has a pool of memory from which executors get their share.
- **Executors**: Run on worker nodes to execute tasks and manage data (e.g., caching), limited by allocated memory and CPU cores.

This setup ensures distributed processing across multiple nodes, with the driver handling orchestration and executors handling the computation.

- In this architecture, the Client Node (driver) coordinates the Spark job, while each Worker Node contains multiple executors that perform the job's tasks in parallel.

- The Python processes and workers on each node handle Python-specific tasks, with YARN managing resources across the entire setup to optimize processing and maintain task isolation.

- This setup allows for efficient distributed data processing across a large cluster.

1. Client Node

The Client Node is where the Spark application is initially submitted. It acts as the "control center" for the Spark job and includes the following components:

Driver JVM (Java Virtual Machine): This is where the Spark driver program runs. The driver is the main coordinator that:

- Defines the job's tasks and transformations.
- Manages the distribution of data and tasks across the cluster.
- Tracks the progress of the tasks and receives results.

Worker Monitor for Python: This module is responsible for monitoring the memory allocated to Python workers that are part of this Spark job. This is especially important for PySpark jobs, where Spark uses Python processes to execute tasks.

Python Driver: This component is specific to PySpark applications and acts as a bridge for Python-related tasks. It manages communication between the driver JVM and Python worker processes, enabling Python code to be executed within the Spark job.
Spark Context: This is the entry point for a Spark application. The Spark Context is responsible for:

- Creating RDDs (Resilient Distributed Datasets) and managing their transformations and actions.
- Establishing connections to the cluster (YARN in this case) and allocating resources for executors.

Controls:

- `spark.driver.memory`: Controls the amount of memory allocated to the driver process.
- `spark.python.worker.memory`: Controls the memory for Python workers.
- `spark.executor.cores`: Specifies the number of CPU cores allocated to each executor.

2. Worker Nodes

The Worker Nodes are the physical or virtual machines in the cluster that run Spark tasks. Each worker node has its own components and memory pools. Let's break down these elements further:

Core Components

YARN Node Manager: This is a daemon that manages resources on each worker node. It allocates memory and CPU resources to containers on the node, where Spark executors are

launched. The Node Manager also monitors the health and usage of resources on the node and can kill tasks if they exceed their allocated resources.

HDFS Datanode: This component is part of Hadoop's distributed storage system. It manages the storage of blocks of data within HDFS on this worker node. The Datanode allows Spark executors to read and write data directly to and from HDFS without needing to transfer it over the network, making data access faster and more efficient.

Node Memory Pool: This pool, defined by the parameter `yarn.nodemanager.resource.memory-mb`, represents the total memory available for all executors on this node. This memory pool is managed by YARN, which allocates specific portions of it to each executor JVM based on the Spark job's requirements.

3. Executor JVMs

Each worker node can host multiple Executor JVMs (Java Virtual Machines), which are dedicated processes responsible for running Spark tasks. Here's a closer look at each component within an executor:

Executor JVM #1, #2, … up to Executor JVM #M:

- Each JVM represents a Spark executor, which is a process responsible for executing tasks in a distributed manner. Executors receive tasks from the driver and perform computations, storing intermediate results either in memory or on disk.
- Executors handle tasks across partitions of data, enabling parallel processing.

Worker Monitor for Python:

- Each executor JVM has one or more **Worker Monitors for Python** to manage the Python worker processes used by PySpark. These monitors track memory allocation and usage for Python workers to ensure they stay within the allocated limits.

Controls:

- `spark.executor.cores`: This setting defines the number of CPU cores allocated to each executor JVM, allowing it to process multiple tasks in parallel.
- `spark.executor.memory`: Specifies the amount of memory available to each executor JVM. This is crucial for ensuring executors have sufficient resources for data processing, caching, and task execution.

4. Python Processes

Each executor JVM spawns one or more "Python Processes" if the job involves PySpark. These Python processes are responsible for executing tasks written in Python. Within each Python Process, there are Python Workers:

Python Process #1, #2, … up to Python Process #M:

- These processes are launched to run Python code as part of Spark tasks. For example, if a Spark job includes Python UDFs (User-Defined Functions), these will run within Python Processes.

Python Workers (Python Worker #1, #2, etc.):

- Within each Python Process, there can be multiple Python Worker threads. These workers perform the actual task computations written in Python, such as transformations and actions on RDDs.
  - Python workers handle the computations by processing data partitions in parallel across the distributed setup.

Controls:

- `spark.python.worker.memory`: This configuration defines the memory allocation for each Python worker. Since Python workers can consume significant memory, especially with large datasets, managing this setting is essential.
- `spark.executor.cores`: Controls the number of CPU cores available to the Python workers within each executor, allowing for parallel processing within each Python process.

Summary of Key Configurations:

Driver-Level:

- `spark.driver.memory`: Memory allocated to the driver JVM on the client node.
- `spark.python.worker.memory`: Memory allocated to each Python worker.
- `spark.executor.cores`: CPU cores assigned to each executor.

Node-Level:

`yarn.nodemanager.resource.memory-mb`: Defines the memory pool on each worker node.

Executor-Level:

- `spark.executor.memory`: Memory allocated to each executor JVM.
- `spark.executor.cores`: CPU cores allocated to each executor JVM.

Python Worker-Level:

- `spark.python.worker.memory`: Controls memory usage for each Python worker.
- `spark.executor.cores`: Specifies CPU allocation for each Python worker.

Shuffling

- In distributed computing, shuffling refers to the process of redistributing data across the cluster to ensure that related data (based on some key) is grouped together for specific computations.

- This is essential for operations where data needs to be aggregated, joined, or processed collectively by keys.

Example 1: Counting Calls by Day

Imagine you have a large dataset of phone call records spread across multiple machines in a cluster. Each record contains information about a single call, including the day on which the call happened. Now, let's say you want to calculate the total number of calls made each day.

1.Setting the Key: In this case, the "day" field will act as the key. Each record (representing one call) will emit a value of "1" for that key (the day), as each record represents one call.

2. Summing the Values by Key: To get the total number of calls per day, you need to "sum up all the values (1s) for each day". However, the records for each day are distributed across different machines in the cluster. If we want to compute the sum for a specific day, say January 1, all records with the key "January 1" need to be gathered together on a single machine. Otherwise, we can't calculate the sum accurately.

3. Why Shuffle?: Shuffling makes this grouping possible by moving all records with the same key to the same machine. This way, each machine has all the records it needs to compute the total number of calls for each day, allowing the aggregation to be completed.

Example 2: Joining Two Tables by ID

Shuffling is also essential for operations like "joining tables". Imagine you have two tables, each containing customer information, and you want to join them by an "ID" field.

1. Same Key in Different Tables: For each ID, say ID = 42, you need to ensure that all data related to ID = 42 from both tables is available on the same machine. If the tables are spread across multiple machines, records with the same ID might be on different machines.

2. Efficient Partitioning for Joins: Let's say IDs in both tables range from 1 to 1,000,000. To join them efficiently, we want both tables to be partitioned in the same way. For example, IDs 1–100 should be stored in Partition 1 for both tables, IDs 101–200 in Partition 2, and so on. This way, when joining, each machine only needs to look at its corresponding partition instead of searching the entire second table for each partition in the first table.

3.Reducing Computations: With data shuffled and organized by key, each partition only needs to process the matching records, making the join operation much faster and more efficient.

Why Shuffling is Important???

Shuffling is crucial for these reasons:

Data Grouping: It ensures that all records with the same key are grouped together, which is essential for aggregations (like sums, counts) and join operations.
Parallel Processing: By distributing data based on keys, the cluster can process tasks in parallel more efficiently, as each node handles only the data it needs to.
Optimized Computation: Without shuffling, operations like joins and aggregations would require excessive data transfer and redundant calculations, significantly slowing down processing times.

How Shuffling Works in Spark

In systems like Spark, shuffling happens in stages:

Map Phase: Each task (usually a "map" task) processes a partition and groups data by key.
Shuffle Phase: Data is redistributed across the cluster so that all records with the same key are located on the same machine.
Reduce Phase: The data is then processed in parallel by "reduce" tasks that perform the required operation (like sum, join, etc.) on the grouped data.

- Shuffling is the backbone of distributed data processing for operations where data needs to be grouped or matched by specific keys.

- It involves moving data across nodes in the cluster so that each key's data is located together, enabling efficient computation of aggregations, joins, and other key-based operations.

Hash shuffle

MapReduce naming convention is followed.
In the shuffle operation, the task that emits the data in the source executor is “mapper”, the task that consumes the data into the target executor is “reducer”, and what happens between them is “shuffle”.

Hash Shuffle process in Apache Spark, specifically focusing on how map tasks and output files are managed within an Executor JVM.
Hash shuffle is a technique used in Spark to redistribute data across the cluster for operations like joins, groupBy, and aggregations.

Key Components

1. Executor JVM:
- This is the environment where Spark tasks execute.
- Each executor operates on multiple partitions of data, each of which is an independent chunk of the dataset.

2. Map Task:
- A map task processes data within a partition. In a shuffle, map tasks prepare data for the subsequent "reduce" stage by organizing it into output files based on specific keys.
- The number of map tasks executed by each executor depends on the number of data partitions and the parallelism level.

3. Local Directory (spark.local.dir):
- Each executor has a local directory to store intermediate data.
- During a shuffle, this local directory holds output files that map tasks write for each reducer.

4. Output Files:
- Each output file represents data that will be read by a specific reducer. The number of output files created corresponds to the number of reducers.
- If there are multiple reducers, each map task generates a separate output file for each reducer, ensuring that the data is partitioned correctly for the reduce phase.

Steps in the Hash Shuffle Process

1. Data Partitioning in the Executor JVM:
- The data is divided into partitions within the Executor JVM. Each partition is handled by a separate map task.
- Each map task processes a specific partition and prepares data for the shuffle.

2. Execution of Map Tasks:
- The map tasks read their respective partitions, process the data, and partition the output by hash (based on keys).

- The output is organized by the hash values of the keys, which ensures that records with the same key are sent to the same reducer.

3. Writing to Output Files:

- Each map task writes its processed data to multiple output files in the "Local Directory" (`spark.local.dir`).
- For each reducer, the map task creates a separate output file in the local directory. The number of output files per map task equals the number of reducers.
- This step is controlled by the configuration properties `spark.executor.cores` (total cores available for each executor) and `spark.task.cpus` (CPUs per task), which determine the level of parallelism.

4. Organizing Data for Reducers:

- The output files are organized so that each file corresponds to a specific reducer.
- During the reduce phase, each reducer will read its respective output files from the local directories of all executors.

5. Final Output for Reducers:

- The output files created by map tasks are used as input for the reduce tasks. Each reducer will gather all its data across the executors from the local directories, allowing it to process data grouped by keys.

In summary, during a hash shuffle, each map task processes a partition, organizes data by keys, and writes multiple output files (one per reducer) to the local directory. This ensures data is correctly distributed across reducers for the next stage in the Spark job.

Hash Shuffle with Consolidation

1. Basic Hash Shuffle Mechanism:

- In Hash Shuffle, each map task creates a separate output file for each reducer. This results in a large number of small files if there are many maps and reduce tasks.
- For example, if you have 10 map tasks and 100 reducers, there would be 1,000 output files (10 files per reducer from each map task).

2. Shuffle File Consolidation:

- To reduce the number of shuffle files, Spark introduced "Shuffle File Consolidation".
- With consolidation, instead of each map task creating its own files, multiple map tasks within an executor share consolidated output files, grouped by the reducers.

3. Executor JVM and Task Parallelism:

- Each executor JVM has multiple cores and can run multiple map tasks in parallel.
- The number of concurrent map tasks within an executor depends on `spark.executor.cores` and `spark.task.cpus` (number of cores per task).

4. Shared Output Files:

- For each reducer, there is a single consolidated output file in the local directory (`spark.local.dir`) instead of separate files from each map task.
- This means that if multiple map tasks are running in parallel, they will write their data for a specific reducer to the same file, rather than creating separate files.

5. Structure of Consolidated Files:

- The output files are divided by the number of reducers, so each reducer has its corresponding output file(s) in the local directory.
- Each output file contains data from multiple map tasks but is organized to ensure data consistency for each reducer.

6. Benefits of Consolidation:

- Reduces the number of output files, lowering the overhead on the file system.
- Improves performance by reducing the file-handling workload during the shuffle stage.
- Helps in scenarios where the shuffle data is extensive, as managing fewer files can significantly improve the efficiency of data transfer between the map and reduce stages.

"Hash Shuffle with Consolidation" allows Spark executors to share output files for reducers, minimizing file creation and management overhead, which can improve the performance of shuffle-intensive operations.

Shuffle with Consolidation

Spark Shuffle process can be optimized by consolidating files to improve performance and reduce the number of output files generated during data shuffling. Let's break down the key concepts and calculations step-by-step for a better understanding.

Key Terms and Setup

- **Executors (`E`):** These are the worker nodes in a Spark cluster that execute tasks. The number of executors can be set with `--num-executors` when using YARN as the cluster manager.
- **Cores per Executor(`C`):** Each executor has a certain number of cores, set by `spark.executor.cores` or `--executor-cores` in YARN.
- **CPU per Task (`T`):** Each Spark task can require a specific number of CPUs to run, which is defined by `spark.task.cpus`.
- **Reducers (`R`):** Reducers are tasks that process the output of the map tasks. The number of reducers depends on the shuffle operation being performed and how the data is partitioned.

- There is an optimization implemented for this shuffler, controlled by the parameter "***spark.shuffle.consolidateFiles***" (default is "false").
- When it is set to "true", the "mapper" output files would be consolidated.
- If cluster has **E** executors ("***–num-executors***" for YARN) and each of them has **C** cores ("***spark.executor.cores***" or "***–executor-cores***" for YARN) and each task asks for **T** CPUs (*"**spark.task.cpus**"*), then the amount of execution slots on the cluster would be **E * C / T**, and the number of files created during shuffle would be **E * (C / T) * R**.

Consolidating Files with `spark.shuffle.consolidateFiles`

When the parameter `spark.shuffle.consolidateFiles` is set to `true`, Spark reduces the number of output files by pooling output files for each reducer.

Steps:

1. File Pooling: Instead of each task creating new files for every reducer, Spark allows a pool of files to be reused by multiple tasks. Each "map" task requests a group of "R" output files from the pool, and when it's done, it returns the files to the pool for other tasks to reuse.

2. Impact on Number of Files: The number of file groups created will be proportional to the number of tasks that can run in parallel (i.e., `C / T` per executor). Therefore, each executor will create only `C / T` groups of `R` files.

3. Reducing Total Files: With consolidation, the total number of files created during the shuffle would be:

Total Files with Consolidation} = E *(C/T)* R

This is significantly less than the original count.

Example with Consolidation
Continuing with our example:
- `E = 100` executors
- `C = 10` cores per executor
- `T = 1` CPU per task
- `R = 46000` reducers

Then:
Total Files with Consolidation} = 100 *(10/1)* 46000 = 46,000,000

This reduces the number of output files from 2 billion (without consolidation) to 46 million. This reduction in files helps improve the performance of the shuffle process by lowering disk I/O and minimizing the load on the file system.

Summary

- Without consolidation: Each task generates its own files, resulting in a very high number of files, especially with a large number of reducers.

- With consolidation (`spark.shuffle.consolidateFiles = true`): Tasks reuse a smaller pool of files, significantly reducing the total number of files generated. This is particularly beneficial in large clusters with high parallelism, as it improves disk utilization and reduces the strain on the file system, ultimately speeding up the shuffle phase.

By controlling the shuffle output files, Spark can handle large datasets more efficiently, making data processing faster and more scalable.

The various colors across each "map output" file represent "different partitions" of data created by the map tasks. Here's what the colors signify in the context of Spark's shuffle process:

1. Partitioning of Data:

- During the shuffle phase, each map task processes a chunk of data from an initial dataset and divides the output into multiple partitions.
- Each partition is intended for a different reducer, so each map task creates several output files — one for each reducer.

2. Color Representation:

- Each color in the output files corresponds to a unique partition intended for a specific reducer.
- For example, in the image, we see output files with various colors (orange, teal, purple, green, etc.), with each color representing data destined for a particular reducer.
- If there are, say, three reducers, each map task would create three partitions, coloring each one differently to represent the reducer it's assigned to.

3. Purpose of Colors/Partitions:

- These partitions ensure that data belonging to the same key (like a specific day or ID, as in your earlier example) is collected together.

- During the shuffle, each map task will write data to different partitions (represented by colors) based on the key. When the shuffle completes, all data belonging to the same key is stored together in a single partition file, ready to be processed by the corresponding reducer.

4. Flow of Data:

- Once the map task writes the data in different partitions (colored blocks), the shuffle process ensures that all blocks of the same color from different map tasks are sent to the same reducer.
- This way, all the data related to a specific key or partition (represented by a color) is combined on one machine for further aggregation or processing.

- The different colors in each map output file visually represent the partitioning of data across reducers.
- It ensures that each reducer receives all necessary data to perform its aggregation or join operations on the appropriate subset of data.
- This mechanism is crucial for distributed computations where data must be grouped by key across a cluster.

Sort Shuffle process in Spark:

1. Default Shuffle Algorithm: Starting from Spark 1.2.0, the default shuffle algorithm is Sort Shuffle (`spark.shuffle.manager = sort`), inspired by Hadoop MapReduce.

2. Output Structure:

- Unlike Hash Shuffle, which creates separate files for each reducer, Sort Shuffle produces a single file ordered by reducer ID and indexed.
- This allows the fetching of data for a specific reducer using a single `fseek` before `fread`.

3. Fallback Plan:

- For a small number of reducers, Sort Shuffle falls back to Hash Shuffle by creating separate files, determined by `spark.shuffle.sort.bypassMergeThreshold` (default: 200 reducers).
- This fallback logic is handled by `BypassMergeSortShuffleWriter`.

4. Sorting and Reduce Side Processing:

- Sort Shuffle sorts data on the map side but does not merge these sorted results on the reduce side.
- When ordering is needed on the reduce side, Spark re-sorts the data using TimSort.
- TimSort Algorithm:
  - TimSort leverages pre-sorted inputs to merge them efficiently.
  - The complexity of merging M sorted arrays of N elements is O(MNlogM) using a Min Heap.
  - Cloudera proposed a patch to optimize reduce-side sorting by utilizing pre-sorted outputs from mappers, but it hasn't been approved due to minimal performance impact.

5. Memory Management and Spilling:

- If there isn't enough memory to store the entire map output, intermediate data spills to disk (`spark.shuffle.spill` is enabled by default).
- The available memory for storing map outputs before spilling is `JVM Heap Size * spark.shuffle.memoryFraction * spark.shuffle.safetyFraction`.
- With default values, this equals `JVM Heap Size * 0.16`.
- For multiple threads in an executor, memory per task is divided by the number of cores, reducing available memory per task.

6. Data Storage Structure:

- Spark uses an `AppendOnlyMap` structure (a customized hash table) to store map output in memory.
- The hash table uses MurmurHash3 as a hash function and supports combining logic, allowing new values to be combined with existing ones in place.

7. Spilling Process:

- When spilling occurs, Spark applies TimSort to the data in `AppendOnlyMap` and writes sorted output to disk.

- Disk writes happen either during spilling or when mapper output is complete.

8. Real-Time Merging:

- Each spill file is written to disk separately.
- During shuffle read, reducers perform real-time merging of spill files using a Min Heap implemented by Java's `PriorityQueue` class.
- Unlike Hadoop MapReduce, Spark does not use an on-disk merger; merging is performed dynamically as reducers request data.

So, regarding this shuffle:

Pros:

1. Smaller number of files created on "map" side
2. Smaller amount of random IO operations, mostly sequential writes and reads

Cons:

1. Sorting is slower than hashing. It might worth tuning the bypassMergeThreshold parameter for your own cluster to find a sweet spot, but in general for most of the clusters it is even too high with its default
2. In case you use SSD drives for the temporary data of Spark shuffles, hash shuffle might work better for you

Tungsten Sort (Unsafe Shuffle) in Spark

Introduction:

- Tungsten Sort (Unsafe Shuffle) is an optimization feature in Spark, introduced in version 1.4.0+.
- Activated by setting `spark.shuffle.manager = tungsten-sort`.
- Part of Spark's "Tungsten" project, designed to improve performance by optimizing memory and CPU usage during shuffle operations.

Key Optimizations:

1. Operate Directly on Serialized Data:

- Processes data without deserialization, treating serialized data as a byte array.
- Uses `sun.misc.Unsafe` functions for memory copying, improving speed by copying data directly.

2. Cache-Efficient Sorting:

- Employs `ShuffleExternalSorter`, a special sorter that optimizes CPU cache usage.
- Sorts arrays of compressed pointers to records and partition IDs, using only 8 bytes per record in the sorting array for better cache efficiency.

3. Direct Spilling of Serialized Data:

- Serialized data is spilled (written to disk) directly, avoiding the need to deserialize, compare, and re-serialize during spill.

4. Spill-Merging Optimizations:

- For shuffle compression codecs that support stream concatenation (e.g., LZF serializer), spill files can be merged by simple concatenation, improving efficiency.
- Enabled through `shuffle.unsafe.fastMergeEnabled` parameter.

5. Off-Heap Storage:

- Planned as an additional optimization step, utilizing off-heap storage buffers to reduce garbage collection and increase memory efficiency.

Conditions for Using Tungsten Sort:

No Aggregation in Shuffle Dependency:

- This optimization only works when there's no aggregation, as aggregation requires deserialization to accumulate values.

Serializer Compatibility:

- The serializer must support relocating serialized values, which is currently compatible with `KryoSerializer` and Spark SQL's custom serializer.

Output Partition Limit:

- The shuffle operation should produce fewer than 16,777,216 output partitions.

Record Size Limit:

- Individual serialized records must not exceed 128 MB.

Sorting Characteristics:

- Sorting is performed only by partition ID.
- This limits some “reduce” side optimizations, such as merging pre-sorted data using TimSort.
- Sorting uses 8-byte values encoding links to the serialized data item and partition number, capping output partitions at 1.6 billion.

This configuration enhances performance in Spark jobs by minimizing memory usage and maximizing CPU efficiency, especially when working with large datasets and high partition counts.