## Anatomy of YARN application runs

The anatomy of a YARN (Yet Another Resource Negotiator) application run involves several key components and steps. Here's an overview of the process:

**Key Components**

1. **ResourceManager (RM)**: Manages the resources across the cluster.
2. **NodeManager (NM)**: Manages the lifecycle of containers on individual nodes.
3. **ApplicationMaster (AM)**: Manages the lifecycle of applications.
4. **Containers**: Execution units that run the application tasks.

**Steps in a YARN Application Run**

1. **Client Submission**:
    - The client submits an application to the ResourceManager.
    - The submission includes application code, necessary files, and resource requirements.
2. **ResourceManager**:
    - The ResourceManager allocates a container for the ApplicationMaster.
    - It notifies the NodeManager to launch the container.
3. **ApplicationMaster Initialization**:
    - The ApplicationMaster is launched in the allocated container.
    - It registers with the ResourceManager and negotiates resources for the application's tasks.
4. **Resource Allocation**:
    - The Application Master requests resources for application tasks.
    - The Resource Manager allocates containers on appropriate NodeManagers based on resource availability and requirements.
5. **Container Launch**:
    - The Node Manager launches containers as per the instructions from the Resource Manager.
    - Containers execute the application tasks.
6. **Task Execution**:
    - The tasks run inside the containers and perform the necessary computations.
    - Containers can communicate with each other as needed.
7. **Monitoring and Progress Reporting**:
    - The ApplicationMaster monitors the progress of the tasks and reports to the ResourceManager.
    - The ResourceManager monitors the overall application status and resource utilization.
8. **Completion**:

   - Once all tasks are completed, the ApplicationMaster notifies the ResourceManager.
   - The ResourceManager deallocates the resources.
   - The ApplicationMaster exits, and the client is notified of the application completion.

9. **Cleanup**:
   - The NodeManager cleans up the resources used by the containers.
   - Logs and other artifacts may be collected for further analysis.

**Diagrammatic Representation**

Here’s a simplified flow diagram of a YARN application run:

arduino

Copy code

```
Client
  |
  V
ResourceManager <----> NodeManager (Launches ApplicationMaster)
  |                      |
  |                      V
  |                 ApplicationMaster
  |                      |
  |                      V
  |              Request Resources for Tasks
  |                      |
  |                      V
  |               Resource Allocation
  |                      |
  |                      V
  |                 NodeManager (Launches Task Containers)
  |                      |
  |                      V
  |                  Task Execution
  |                      |
  |                      V
  |                 Monitoring & Reporting
  |                      |
  |                      V
  |                  Completion Notification
  |                      |
```

```
 |                   V
ResourceManager <----> ApplicationMaster (Exit)
 |
 V
Cleanup
```

This is a high-level overview. Each step involves detailed interactions and protocols ensuring efficient resource management and task execution in a distributed computing environment.

## YARN compared to Map Reduce 1

YARN (Yet Another Resource Negotiator) and MapReduce 1 (MR1) are both frameworks for processing large data sets in a distributed environment, but they have significant differences in architecture and functionality. Here's a detailed comparison:

**Architecture**

**MapReduce 1 (MR1)**:

- **Single Master Node**: MR1 has a single master node called the **JobTracker**.
  - **JobTracker**: Manages both job scheduling and resource management.
  - **TaskTrackers**: Slave nodes that run tasks and report progress back to the JobTracker.
- **Limitations**:
  - Scalability issues due to the single JobTracker.
  - Potential bottleneck and single point of failure.

**YARN**:

- **Separation of Responsibilities**: YARN separates resource management from job scheduling and monitoring.
  - **ResourceManager (RM)**: Manages resources across the cluster.
  - **NodeManagers (NM)**: Manage resources on individual nodes and monitor containers.
  - **ApplicationMaster (AM)**: Manages the lifecycle of a specific application, handling job scheduling and monitoring.

**Resource Management**

**MapReduce 1 (MR1)**:

- **Fixed Slots**: TaskTrackers have fixed slots for Map and Reduce tasks.
  - Underutilization: If there are no Map tasks, Reduce slots remain idle, and vice versa.
- **Centralized Control**: JobTracker handles all resource allocation and scheduling.

**YARN**:

- **Dynamic Resource Allocation**: Resources are allocated dynamically based on the needs of the application.

  - Containers: Units of resource allocation that can run any type of task (Map, Reduce, or other).
- **Scalability**: More scalable as resource management is distributed, reducing the burden on a single node.

**Flexibility**

**MapReduce 1 (MR1)**:

- **Rigid Framework**: Designed specifically for MapReduce applications.
  - Limited to Map and Reduce programming model.

**YARN**:

- **Flexible Framework**: Can run various types of distributed applications beyond MapReduce.
  - Supports multiple programming models like Graph processing (Apache Giraph), Iterative processing (Apache Spark), and more.

**Fault Tolerance and Recovery**

**MapReduce 1 (MR1)**:

- **Single Point of Failure**: JobTracker is a single point of failure.
  - If JobTracker fails, running jobs may fail or need to be restarted.

**YARN**:

- **Better Fault Tolerance**: ApplicationMaster failures can be handled by restarting them on different nodes.
  - ResourceManager High Availability: ResourceManager can be configured for high availability.

**Resource Utilization**

**MapReduce 1 (MR1)**:

- **Static Slot Configuration**: Fixed Map and Reduce slots can lead to inefficient resource utilization.

**YARN**:

- **Resource Containers**: Flexible and efficient resource utilization through dynamic containers that can be allocated based on application requirements.

**Ecosystem Integration**

**MapReduce 1 (MR1)**:

- **MapReduce Centric**: Primarily designed to support MapReduce jobs.

**YARN**:

- **Broad Ecosystem**: Integrates with a wide range of data processing frameworks and applications.
  - Ecosystem support: HDFS, HBase, Hive, Spark, and others.

**Evolution and Community Support**

**MapReduce 1 (MR1)**:

- **Legacy System**: The original Hadoop processing framework, but largely superseded by YARN.

**YARN**:

- **Modern Standard**: Adopted as the standard resource management layer in Hadoop 2 and beyond.
  - Continual improvements and wide community support.

**Summary**

| Feature | MapReduce 1 (MR1) | YARN |
|---|---|---|
| Architecture | Single JobTracker, multiple TaskTrackers | ResourceManager, NodeManagers, ApplicationMasters |
| Resource Management | Fixed slots for Map and Reduce tasks | Dynamic resource allocation with containers |
| Flexibility | Designed for MapReduce | Supports various distributed applications |
| Fault Tolerance and Recovery | Single point of failure (JobTracker) | Better fault tolerance and HA for ResourceManager |
| Resource Utilization | Static slot configuration | Efficient, dynamic container allocation |
| Ecosystem Integration | MapReduce focused | Broad integration with various frameworks |
| Evolution and Community Support | Legacy system | Modern standard with wide community support |

YARN offers significant improvements in scalability, flexibility, and resource utilization over MapReduce 1, making it a more versatile and robust choice for modern distributed computing needs.

## Scheduling in YARN

Scheduling in YARN (Yet Another Resource Negotiator) is a critical aspect of managing resources efficiently in a distributed environment. YARN employs several schedulers, each with its own approach to resource allocation and job prioritization. Here’s a detailed overview of scheduling in YARN:

**Key Schedulers in YARN**

1. **FIFO Scheduler**:
   - **First In, First Out**: Jobs are scheduled in the order they are submitted.
   - **Simplicity**: Easy to implement and understand.
   - **Limitations**: Can lead to resource starvation for smaller or later-submitted jobs.
2. **Capacity Scheduler**:

   - **Resource Guarantees**: Divides cluster resources into queues, each with a guaranteed capacity.
   - **Hierarchical Queues**: Supports nested queues, enabling resource sharing within organizational hierarchies.
   - **Resource Allocation**: Dynamically adjusts resource allocation based on demand and queue configurations.
   - **Use Case**: Suitable for multi-tenant environments where resource fairness and guarantees are important.
3. **Fair Scheduler**:
   - **Fairness**: Attempts to allocate resources such that all jobs get, on average, an equal share of resources over time.
   - **Preemption**: Supports preemption to ensure fairness, where long-running jobs may be preempted to make way for others.
   - **Pools**: Jobs can be grouped into pools, each with its own policies and resource shares.
   - **Use Case**: Ideal for environments requiring fair resource distribution among users or applications.

**Scheduling Mechanisms**

1. **Resource Requests**:
   - **Containers**: Applications request containers specifying resource requirements (memory, CPU).
   - **Priority**: Applications can specify priority for different resource requests.
2. **Resource Allocation**:
   - **NodeManagers**: Track available resources on each node and report to the ResourceManager.
   - **ResourceManager**: Uses the scheduler to match resource requests with available resources.
3. **Node Selection**:
   - **Data Locality**: Scheduler tries to place tasks on nodes where the data resides, minimizing data transfer.
   - **Rack Awareness**: If data-local nodes are not available, tasks are placed on nodes within the same rack.
4. **Preemption**:
   - **Fairness and Capacity**: In scenarios where certain jobs or users are consuming more than their fair share or guaranteed capacity, preemption can occur.
   - **Controlled Preemption**: Ensures minimal disruption and fairness by preempting tasks in a controlled manner.
5. **Resource Utilization**:

- **Dynamic Allocation**: Resources are allocated and de-allocated dynamically based on the application's lifecycle.
- **Efficiency**: Scheduler aims to maximize cluster utilization while maintaining fairness and capacity guarantees.

**Example Flow of Scheduling**

1. **Job Submission**: A client submits an application to the ResourceManager.
2. **ApplicationMaster Launch**: The ResourceManager allocates a container for the ApplicationMaster (AM) on a NodeManager.
3. **Resource Requests**: The AM requests containers for tasks from the ResourceManager.
4. **Scheduling Decisions**: The scheduler evaluates available resources and job priorities to make allocation decisions.
5. **Container Launch**: NodeManagers launch containers as per the scheduler's instructions.
6. **Task Execution**: Tasks run within containers, processing data.
7. **Monitoring and Adjustment**: The AM monitors task progress and may request additional resources or release unused ones.
8. **Completion**: Once all tasks are complete, the AM releases all resources and reports job completion to the ResourceManager.

**Advantages and Challenges**

**Advantages**:

- **Flexibility**: Multiple scheduling policies support diverse workload requirements.
- **Scalability**: Efficient resource management in large clusters.
- **Fairness**: Ensures fair resource distribution among users and applications.

**Challenges**:

- **Complexity**: Managing and configuring multiple schedulers and policies can be complex.
- **Resource Contention**: Ensuring optimal resource allocation without contention requires careful tuning.
- **Preemption Overhead**: Preempting tasks can lead to overhead and performance degradation if not managed properly.

**Conclusion**

YARN's scheduling framework is designed to balance resource utilization, fairness, and efficiency in a distributed environment. By offering multiple schedulers like FIFO, Capacity, and Fair schedulers, YARN provides flexibility to cater to different workload demands and organizational policies. Effective scheduling in YARN ensures that resources are used optimally, jobs are executed efficiently, and the overall system performance is maximized.

## Anatomy of Map Reduce job run

The anatomy of a MapReduce job run involves multiple stages, each crucial for the successful execution of the job. Here's a detailed overview of the key stages and components involved in a MapReduce job:

**Key Components**

1. **JobClient**: Submits the job and tracks its status.
2. **JobTracker**: Manages the job scheduling and resource allocation (In MR1).
3. **TaskTracker**: Executes the individual tasks of the job (In MR1).
4. **MapTask**: Processes the input data and produces intermediate key-value pairs.
5. **ReduceTask**: Processes the intermediate data and produces the final output.
6. **HDFS**: Hadoop Distributed File System, where input and output data are stored.

**Stages of a MapReduce Job Run**

1. **Job Submission**:
   - **Client**: The user or client submits a job to the JobTracker (MR1) or ResourceManager (YARN).
   - **Job Configuration**: The client specifies job configuration parameters, including input/output paths, mapper/reducer classes, and other settings.
2. **Job Initialization**:
   - **JobTracker (MR1)**: Assigns a unique job ID and creates a job configuration.
   - **ResourceManager (YARN)**: Allocates a container for the ApplicationMaster (AM), which then initializes the job.
3. **Splitting the Input Data**:
   - **InputFormat**: Determines how the input data is split into smaller chunks (input splits).
   - **Input Splits**: Logical divisions of the input data, each assigned to a mapper.
4. **Map Phase**:
   - **Mapper Tasks**: Each input split is processed by a mapper task, which reads the data, processes it, and produces intermediate key-value pairs.
   - **RecordReader**: Converts input splits into key-value pairs for the mapper.
5. **Shuffling and Sorting**:
   - **Intermediate Data**: The key-value pairs produced by mappers are partitioned by key and shuffled to the reducers.
   - **Sorting**: The data is sorted by key within each partition to facilitate the reduce phase.
6. **Reduce Phase**:
   - **Reducer Tasks**: Each reducer processes a partition of the intermediate data, aggregating the values for each key and producing the final output.
   - **OutputFormat**: Determines how the final output is written to the file system.
7. **Job Completion**:
   - **Task Completion**: TaskTrackers (MR1) or NodeManagers (YARN) report task completion to the JobTracker or ApplicationMaster.
   - **JobTracker (MR1)**: Monitors the progress of the job and updates its status.

  - **ResourceManager (YARN)**: The ApplicationMaster coordinates with the ResourceManager to monitor job progress.

8. **Cleanup**:
   - **Temporary Data**: Intermediate data and temporary files are cleaned up.
   - **Resource Release**: Resources allocated for the job (containers, memory, etc.) are released.

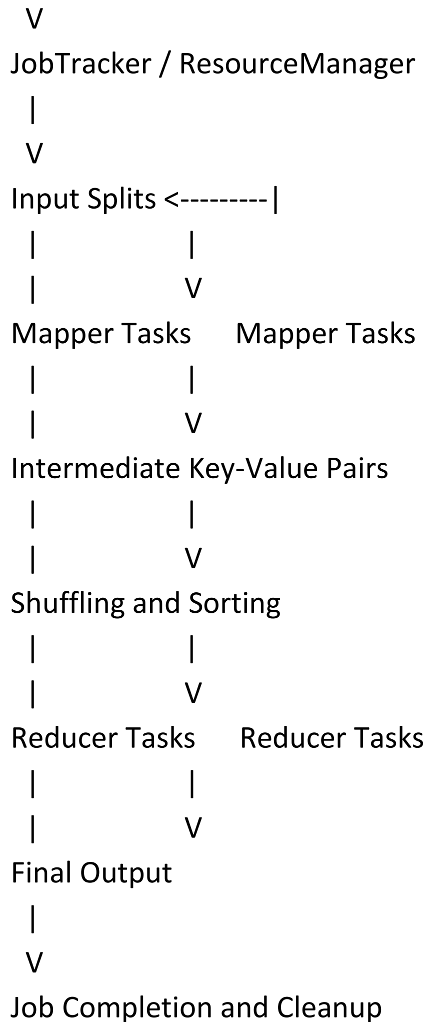
**Detailed Flow of a MapReduce Job Run**

1. **Job Submission**:
   - The client submits the job to the JobTracker (MR1) or ResourceManager (YARN).
   - The job configuration is sent along with the job.
2. **Job Initialization**:
   - The JobTracker (MR1) or ResourceManager (YARN) initializes the job.
   - The input data is split into input splits.
3. **Map Phase**:
   - Each input split is assigned to a mapper task.
   - The mapper processes the input data and produces intermediate key-value pairs.
   - The output of the mapper is written to local disk.
4. **Shuffling and Sorting**:
   - The intermediate data is shuffled and sorted by key.
   - Each reducer receives all the values associated with a particular key.
5. **Reduce Phase**:
   - The reducer processes the intermediate data.
   - The output of the reducer is written to the output location specified in the job configuration.
6. **Job Completion**:
   - The JobTracker (MR1) or ApplicationMaster (YARN) monitors the job's progress.
   - The client is notified upon job completion.
7. **Cleanup**:
   - Intermediate and temporary data are cleaned up.
   - Resources are released back to the cluster.

**Diagrammatic Representation**

Here’s a simplified diagram of a MapReduce job run:

```
Client
  |
  V
Job Submission
  |
```




**Summary**

A MapReduce job run involves submitting a job, splitting the input data, processing it through map and reduce tasks, shuffling and sorting intermediate data, and finally writing the output. The process is managed by JobTracker and TaskTrackers in MR1, or ResourceManager and ApplicationMaster in YARN, ensuring efficient resource utilization and fault tolerance.

## Failures

Handling failures is a critical aspect of the MapReduce framework to ensure robustness and reliability in a distributed computing environment. Here's a detailed look at the types of failures that can occur in a MapReduce job run and how they are handled:

**Types of Failures**

1. **Task Failure**:
   - **Map Task Failure**: Occurs when a mapper fails to process its input split.
   - **Reduce Task Failure**: Occurs when a reducer fails to process its intermediate data.
2. **Node Failure**:

- **TaskTracker/NodeManager Failure**: Occurs when a node running TaskTracker (MR1) or NodeManager (YARN) crashes or becomes unreachable.

3. **JobTracker/ResourceManager Failure**:
    - **JobTracker Failure (MR1)**: Single point of failure in MR1; if the JobTracker fails, the entire job may need to be restarted.
    - **ResourceManager Failure (YARN)**: Can be mitigated with high availability configurations.
4. **ApplicationMaster Failure (YARN)**:
    - Occurs when the ApplicationMaster managing the job fails in a YARN environment.
5. **Network Failures**:
    - **Data Transfer Issues**: Failures in transferring data between nodes during shuffling and sorting.

**Handling Failures**

**Task Failure Handling**

1. **Map Task Failure**:
    - **Retries**: If a map task fails, it is retried a certain number of times (usually 4 by default).
    - **Re-execution**: The task is re-executed on a different node. The input split is read again and processed by a new mapper.
2. **Reduce Task Failure**:
    - **Retries**: Similar to map tasks, reduce tasks are retried a certain number of times upon failure.
    - **Re-execution**: The task is re-executed on a different node. The intermediate data is fetched from the mappers again and processed by a new reducer.

**Node Failure Handling**

1. **TaskTracker/NodeManager Failure**:
    - **Task Reassignment**: Tasks assigned to a failed TaskTracker (MR1) or NodeManager (YARN) are reassigned to other healthy nodes.
    - **Heartbeat Mechanism**: The JobTracker (MR1) or ResourceManager (YARN) detects node failures through periodic heartbeats from TaskTrackers or NodeManagers.

**JobTracker/ResourceManager Failure**

1. **JobTracker Failure (MR1)**:
    - **Single Point of Failure**: In MR1, JobTracker failure is a critical issue. The job might need to be restarted from scratch.
2. **ResourceManager Failure (YARN)**:

    - **High Availability**: YARN can be configured for high availability, with multiple ResourceManagers in an active-standby configuration.
    - **Failover**: Upon ResourceManager failure, a standby ResourceManager takes over, minimizing disruption.

**Application Master Failure (YARN)**

- **Retries**: If the Application Master fails, it is restarted on another node.
- **State Persistence**: Application Masters can persist their state, allowing them to recover and continue from the point of failure.

**Network Failures**

- **Retries and Timeouts**: Data transfer operations have built-in retries and timeouts to handle transient network issues.
- **Data Replication**: HDFS stores data in multiple replicas to ensure availability even if some nodes or network paths are unavailable.

**Fault Tolerance Mechanisms**

1. **Speculative Execution**:
    - **Slow Task Detection**: The framework detects slow-running tasks (stragglers) and launches duplicate tasks (speculative tasks) on other nodes.
    - **Improved Performance**: The first task to complete successfully is used, improving overall job performance.
2. **Data Replication in HDFS**:
    - **Multiple Replicas**: Input data is stored in multiple replicas across different nodes.
    - **Data Availability**: If a node fails, data can still be accessed from another replica.
3. **Job and Task Monitoring**:
    - **Progress Tracking**: The JobTracker (MR1) or ApplicationMaster (YARN) monitors job and task progress, detecting and handling failures promptly.
    - **Retries and Task Failover**: Automatic retries and task reassignment ensure continued progress despite failures.

**Summary**

Failures in a MapReduce job can occur at various levels: task, node, JobTracker/ResourceManager, ApplicationMaster, or network. The framework includes robust mechanisms to handle these failures, such as retries, re-execution, task reassignment, speculative execution, and high availability configurations. These mechanisms ensure that MapReduce jobs can complete successfully even in the presence of failures, making the system reliable and resilient.

## Shuffle and sort

The Shuffle and Sort phase is a critical step in the MapReduce framework, responsible for transferring and organizing intermediate data between the map and reduce phases. Here's a detailed overview of the Shuffle and Sort process:

**Overview of Shuffle and Sort**

The Shuffle and Sort phase involves the following steps:

1. **Shuffling**: The process of moving intermediate data produced by the mappers to the reducers.
2. **Sorting**: Organizing the intermediate data by key before it is processed by the reducers.

**Detailed Steps in Shuffle and Sort**

**During the Map Phase**

1. **Intermediate Data Generation**:
    - Each mapper produces intermediate key-value pairs from the input splits.
    - The output is written to local disk, partitioned by the key's hash code.
2. **Partitioning**:
    - The intermediate data is divided into partitions, each corresponding to a reducer.
    - The number of partitions equals the number of reducers.
3. **Combiner (Optional)**:
    - A local aggregation step that runs after the mapper to reduce the volume of data transferred across the network.
    - Combines intermediate data with the same key, reducing the number of key-value pairs.

**During the Shuffle Phase**

1. **Data Transfer**:
    - The intermediate data is transferred from the mappers to the reducers.
    - Each reducer pulls its corresponding partition from all the mappers.
2. **Data Spilling**:
    - If the intermediate data exceeds the memory buffer, it is spilled to disk.
    - The data is periodically merged and sorted to reduce the number of spills.

**During the Sort Phase**

1. **Sorting**:
    - The intermediate data pulled by each reducer is sorted by key.
    - Sorting ensures that all values for a given key are grouped together.
2. **Merge Sort**:
    - Multiple sorted files (if data was spilled to disk) are merged to produce a single sorted output.

- This merge sort is efficient and necessary for handling large data volumes that cannot fit into memory.

**Data Flow Example**

1. **Map Phase**:
   - Input data: [("apple", 1), ("banana", 1), ("apple", 1), ("cherry", 1)]
   - Mapper output: [("apple", 1), ("apple", 1), ("banana", 1), ("cherry", 1)]
2. **Partitioning**:
   - Partition 0: [("apple", 1), ("apple", 1)]
   - Partition 1: [("banana", 1), ("cherry", 1)]
3. **Shuffling**:
   - Reducer 0 pulls data from Partition 0 of all mappers.
   - Reducer 1 pulls data from Partition 1 of all mappers.
4. **Sorting (Reducer 0)**:
   - Before: [("apple", 1), ("apple", 1)]
   - After: [("apple", [1, 1])]
5. **Sorting (Reducer 1)**:
   - Before: [("banana", 1), ("cherry", 1)]
   - After: [("banana", [1]), ("cherry", [1])]

**Diagrammatic Representation**

Here's a simplified diagram of the Shuffle and Sort process:

lua

Copy code

```
Map Phase:
---------------
Mapper 1:        Mapper 2:
("apple", 1)     ("banana", 1)
("apple", 1)     ("cherry", 1)
("banana", 1)    ("apple", 1)

Shuffle and Sort Phase:
---------------
Reducer 1:       Reducer 2:
("apple", 1)     ("banana", 1)
("apple", 1)     ("cherry", 1)
("apple", 1)

Sort Phase:
---------------
```

Reducer 1: Reducer 2:
("apple", [1, 1, 1]) ("banana", [1])
("cherry", [1])

**Key Considerations**

1. **Network I/O**:
   - Shuffling can be network-intensive as it involves transferring data across nodes.
   - Efficient network communication is crucial for performance.
2. **Memory and Disk Usage**:
   - Proper memory management and efficient use of disk for spills are essential.
   - Sorting large datasets that exceed memory limits requires efficient disk I/O operations.
3. **Combiner Effectiveness**:
   - The combiner can significantly reduce the amount of data transferred, improving performance.
   - It should be used when possible to aggregate intermediate data.
4. **Data Locality**:
   - Ensuring data locality (processing data close to where it is stored) can minimize network I/O.
   - Hadoop attempts to place tasks on nodes where data is already present.

**Summary**

The Shuffle and Sort phase is a vital part of the MapReduce framework, enabling the transition from the map phase to the reduce phase. It involves partitioning intermediate data, transferring it across the network, and sorting it by key. Effective management of this phase is crucial for the performance and scalability of MapReduce jobs, as it directly impacts network I/O, memory, and disk usage.

## Task execution

Task execution in the MapReduce framework involves the detailed steps that a task (either a map task or a reduce task) undergoes from initiation to completion. Here's a comprehensive look at the task execution process:

**Task Execution Workflow**

**Map Task Execution**

1. **Task Initialization**:
   - The TaskTracker (MR1) or NodeManager (YARN) initializes the task.
   - The task is assigned resources (CPU, memory) and input splits.
2. **Reading Input Data**:
   - **InputFormat**: Defines how input data is read and split into records.
   - **RecordReader**: Converts input splits into key-value pairs for the mapper.
3. **Map Function**:

   - **Mapper**: Processes each key-value pair to generate intermediate key-value pairs.
   - **OutputCollector**: Collects the output of the mapper.
4. **Combiner (Optional)**:
   - **Local Aggregation**: The combiner function performs local aggregation on the mapper output to reduce data size.
   - **Intermediate Output**: The output is still in the form of key-value pairs, but potentially reduced.
5. **Partitioning**:
   - **Partitioner**: Assigns each intermediate key-value pair to a reducer based on a partitioning function.
   - **Partition Files**: Intermediate data is written to local disk, partitioned by reducer.
6. **Sorting and Spilling**:
   - **Sorting**: The intermediate data is sorted by key.
   - **Spilling**: If the buffer holding intermediate data is full, it is spilled to disk.
   - **Merging**: Multiple spills are merged into a single sorted file.
7. **Completion**:
   - The TaskTracker (MR1) or NodeManager (YARN) marks the task as complete.
   - The output files are made available for the shuffle phase.

**Reduce Task Execution**

1. **Shuffle Phase**:
   - **Data Transfer**: The reducer pulls its corresponding partitioned data from all mappers.
   - **Merging**: The pulled data is merged and sorted by key.
2. **Sort Phase**:
   - **Sorting**: The intermediate data is sorted to group values by key.
   - **Merge Sort**: Efficiently merges sorted data from multiple mappers.
3. **Reduce Function**:
   - **Reducer**: Processes the sorted key-value pairs to generate the final output.
   - **OutputCollector**: Collects the output of the reducer.
4. **Writing Output**:
   - **OutputFormat**: Defines how the final output is written to HDFS.
   - **RecordWriter**: Writes the final key-value pairs to the specified output location.
5. **Completion**:
   - The TaskTracker (MR1) or NodeManager (YARN) marks the task as complete.
   - The final output is available in HDFS or the specified output location.

**Detailed Flow of Task Execution**

1. **Task Initialization**:
   - The TaskTracker (MR1) or NodeManager (YARN) launches a JVM for the task.

- Configuration parameters and input splits are passed to the task.

2. **Reading Input Data**:
   - The InputFormat determines the boundaries of input splits.
   - The RecordReader reads data from the input split and converts it into key-value pairs.
3. **Map Function**:
   - The map method is called for each key-value pair.
   - The mapper generates intermediate key-value pairs, which are collected by the OutputCollector.
4. **Combiner (Optional)**:
   - The combiner aggregates intermediate key-value pairs with the same key.
   - The output is a reduced set of key-value pairs.
5. **Partitioning**:
   - The Partitioner assigns each intermediate key-value pair to a partition.
   - The partitions correspond to reducers, ensuring data for the same key goes to the same reducer.
6. **Sorting and Spilling**:
   - The intermediate data is sorted by key within each partition.
   - If the in-memory buffer is full, the data is spilled to disk.
   - Multiple spills are merged into a single sorted file.
7. **Completion of Map Task**:
   - The TaskTracker (MR1) or NodeManager (YARN) marks the map task as complete.
   - The intermediate data is ready for the shuffle phase.
8. **Shuffle Phase (Reduce Task)**:
   - The reducer pulls its corresponding partitioned data from all mappers.
   - The data is merged and sorted by key.
9. **Sort Phase**:
   - The intermediate data is sorted to group values by key.
   - Merge sort handles large data efficiently by merging sorted chunks.
10. **Reduce Function**:
    - The reduce method is called for each key and its list of values.
    - The reducer generates the final key-value pairs.
11. **Writing Output**:
    - The OutputFormat defines the output location and format.
    - The RecordWriter writes the final key-value pairs to HDFS.
12. **Completion of Reduce Task**:

  - The TaskTracker (MR1) or NodeManager (YARN) marks the reduce task as complete.
  - The final output is stored in the specified location in HDFS.

**Example**

Consider a word count MapReduce job:

1. **Map Task**:
   - Input: "apple banana apple cherry"
   - Intermediate output: [("apple", 1), ("banana", 1), ("apple", 1), ("cherry", 1)]
2. **Shuffle and Sort**:
   - Partitioned by key: Reducer 0 gets [("apple", 1), ("apple", 1)], Reducer 1 gets [("banana", 1), ("cherry", 1)]
   - Sorted: [("apple", [1, 1]), ("banana", [1]), ("cherry", [1])]
3. **Reduce Task**:
   - Reducer 0: [("apple", [1, 1])] -> [("apple", 2)]
   - Reducer 1: [("banana", [1])] -> [("banana", 1)], [("cherry", [1])] -> [("cherry", 1)]
4. **Final Output**:
   - [("apple", 2), ("banana", 1), ("cherry", 1)]

**Fault Tolerance and Retries**

- **Retries**: If a task fails, it is retried a certain number of times (default is 4).
- **Speculative Execution**: Slow tasks may be duplicated to avoid delays.
- **TaskTracker/NodeManager Monitoring**: These components monitor task progress and handle retries or failover.

**Summary**

Task execution in MapReduce involves initializing tasks, reading and processing input data, partitioning, sorting, shuffling, and finally writing output data. Map tasks generate intermediate key-value pairs, which are shuffled and sorted before being processed by reduce tasks to produce the final output. Effective task execution is crucial for the performance and reliability of MapReduce jobs, with built-in mechanisms for fault tolerance and resource management.

## Map Reduce Features

MapReduce is a powerful programming model and processing technique used for processing large data sets in a distributed computing environment. Here are some of the key features of MapReduce:

**1. Scalability**

- **Horizontal Scalability**: MapReduce can scale horizontally by adding more nodes to the cluster. This allows it to handle increasing volumes of data efficiently.
- **Data Parallelism**: The model processes data in parallel across multiple nodes, making it suitable for handling large-scale data.

**2. Simplicity**

- **Simplified Programming Model**: The MapReduce model abstracts the complexity of parallel and distributed computing, allowing developers to focus on writing the map and reduce functions.
- **Automatic Parallelization**: The framework automatically handles the distribution of data and computation across the cluster.

**3. Fault Tolerance**

- **Task Retry**: If a map or reduce task fails, the framework retries the task on another node.
- **Speculative Execution**: Slow-running tasks (stragglers) are detected and duplicated on other nodes to ensure timely completion.
- **Data Replication**: Input data is stored in multiple replicas across different nodes in HDFS, ensuring data availability even in case of node failures.

**4. Data Locality**

- **Processing Data Close to Storage**: MapReduce attempts to schedule tasks on nodes where the data resides, reducing network I/O and improving performance.
- **Minimized Data Transfer**: By processing data locally, the framework reduces the need for data transfer across the network.

**5. Flexibility**

- **Support for Various Data Formats**: MapReduce can process structured, semi-structured, and unstructured data from different sources.
- **Customizable Input/Output Formats**: Developers can define custom InputFormat and OutputFormat classes to read and write data in various formats.

**6. Load Balancing**

- **Dynamic Task Scheduling**: The JobTracker (MR1) or ResourceManager (YARN) dynamically schedules tasks based on the availability of resources and node health.
- **Task Prioritization**: High-priority tasks can be scheduled to run earlier, ensuring efficient use of cluster resources.

**7. Compatibility**

- **Integration with HDFS**: MapReduce works seamlessly with HDFS, providing a reliable and distributed storage system.
- **Ecosystem Integration**: It integrates well with other Hadoop ecosystem components like Hive, Pig, and HBase, providing a robust data processing environment.

**8. Cost Efficiency**

- **Commodity Hardware**: MapReduce can run on clusters of commodity hardware, reducing the cost of infrastructure.
- **Efficient Resource Utilization**: The framework efficiently utilizes cluster resources, minimizing wastage and improving cost-effectiveness.

**9. High Throughput**

- **Batch Processing**: MapReduce is designed for high-throughput batch processing, making it suitable for large-scale data processing tasks.
- **Aggregated Results**: The reduce phase aggregates intermediate results, providing summarized outputs efficiently.

**10. Security**

- **Access Control**: Integration with Kerberos and other security mechanisms ensures secure access to data and resources.
- **Data Encryption**: Data can be encrypted during transfer and at rest, ensuring data security and privacy.

**Key Components and Their Roles**

1. **JobTracker (MR1)**:
    - Manages job scheduling and resource allocation.
    - Monitors task progress and handles retries and failures.
2. **TaskTracker (MR1)**:
    - Executes map and reduce tasks assigned by the JobTracker.
    - Sends periodic heartbeats to the JobTracker to report task status.
3. **ResourceManager (YARN)**:
    - Manages resource allocation across the cluster.
    - Coordinates with NodeManagers to monitor resource usage and availability.
4. **NodeManager (YARN)**:
    - Manages resources on individual nodes.
    - Executes containerized tasks and monitors their status.
5. **ApplicationMaster (YARN)**:
    - Manages the lifecycle of a specific application (job).
    - Negotiates resources with the ResourceManager and coordinates task execution.

**Summary**

MapReduce offers a robust and scalable framework for processing large data sets in a distributed manner. Its features, such as fault tolerance, data locality, flexibility, and simplicity, make it a popular choice for big data processing tasks. The integration with Hadoop's ecosystem components and the ability to run on commodity hardware further enhance its utility and cost-effectiveness.

## Sorting

Sorting is a fundamental operation in the MapReduce framework, playing a critical role in organizing intermediate data and ensuring the correct execution of the reduce phase. Here’s a detailed look at the sorting process within MapReduce:

**Sorting in MapReduce**

1. **Map Phase Sorting**:

- **Intermediate Data**: During the map phase, each mapper produces key-value pairs as intermediate data.
- **In-Memory Buffer**: The intermediate data is initially stored in an in-memory buffer.
- **Spilling to Disk**: When the buffer fills up, the data is sorted by key and then spilled to disk. This is known as a "spill file."

2. **Combiner (Optional)**:
   - **Local Aggregation**: The combiner function may be applied to reduce the volume of intermediate data. This function acts on the sorted data before spilling to disk.
3. **Multiple Spills**:
   - **Merging Spill Files**: If the mapper generates multiple spill files, these are merged and sorted to form a single sorted output file. This merge operation can happen multiple times if necessary.

**Shuffle and Sort Phase**

1. **Shuffle Phase**:
   - **Data Transfer**: During the shuffle phase, the reducer nodes fetch the sorted intermediate data from the mapper nodes.
   - **Partitioning**: Each reducer fetches data from all mappers for its assigned partition. This ensures that all key-value pairs for a given key are sent to the same reducer.
2. **Merge Sort**:
   - **Merging Intermediate Data**: The fetched data from different mappers is merged and sorted by key. This is a critical step as it ensures that all values for a given key are grouped together.
   - **Efficient Sorting**: The merging process is efficient and can handle large amounts of data by merging sorted chunks incrementally.

**Reduce Phase Sorting**

1. **Reduce Phase**:
   - **Sorted Input**: The input to the reducer is sorted by key. This allows the reducer to process each key and its associated list of values sequentially.
   - **Reduce Function**: The reduce function is applied to each key and its list of values to produce the final output.

**Detailed Sorting Process**

**During Map Phase**

1. **In-Memory Buffer Management**:
   - The mapper writes the output to an in-memory buffer. The buffer is partitioned into regions corresponding to each reducer.
2. **Sorting and Spilling**:

- When the buffer reaches a certain threshold, it is sorted by key. The data is then spilled to disk as a sorted file.

3. **Combiner Application (Optional)**:
   - The combiner function, if specified, is applied to the sorted data to aggregate values with the same key locally.
4. **Merging Spill Files**:
   - If there are multiple spill files, they are merged into a single sorted file to minimize the number of files that reducers need to fetch.

**During Shuffle Phase**

1. **Data Transfer**:
   - Each reducer pulls its assigned partition of the intermediate data from all mappers. This involves transferring sorted spill files over the network.
2. **Intermediate Data Merging**:
   - The reducer merges the sorted intermediate data from multiple mappers into a single sorted sequence. This merging is typically done in memory if the data size allows, or using disk-based merge sort for larger datasets.

**During Reduce Phase**

1. **Processing Sorted Data**:
   - The reducer processes the sorted key-value pairs. For each key, the reducer receives a list of values, which it processes to produce the final output.
2. **Writing Final Output**:
   - The final output key-value pairs are written to the specified output location, usually HDFS.

**Example**

Consider a word count job as an example:

1. **Map Phase**:
   - Input: "apple banana apple cherry"
   - Mapper output: [("apple", 1), ("banana", 1), ("apple", 1), ("cherry", 1)]
2. **Sorting and Spilling**:
   - Sorted and spilled to disk: [("apple", 1), ("apple", 1)], [("banana", 1)], [("cherry", 1)]
3. **Shuffle and Sort**:
   - Reducer 0 pulls: [("apple", 1), ("apple", 1)]
   - Reducer 1 pulls: [("banana", 1)], [("cherry", 1)]
   - Merging and sorting at reducers:
     - Reducer 0: [("apple", [1, 1])]
     - Reducer 1: [("banana", [1])], [("cherry", [1])]
4. **Reduce Phase**:

    - Reducer 0 processes: [("apple", [1, 1])] -> [("apple", 2)]
    - Reducer 1 processes: [("banana", [1])] -> [("banana", 1)], [("cherry", [1])] -> [("cherry", 1)]
5. **Final Output**:
    - [("apple", 2), ("banana", 1), ("cherry", 1)]

**Summary**

Sorting in MapReduce ensures that intermediate data is organized by key, enabling efficient data transfer during the shuffle phase and correct execution of the reduce phase. The process involves multiple steps of in-memory sorting, spilling to disk, merging spill files, and final sorting at the reducers. This organized approach ensures that the MapReduce framework can handle large datasets efficiently and produce accurate results.

## Joins side data distribution

Joins and side data distribution are important techniques in distributed computing, especially in the context of MapReduce, to efficiently combine data from different sources. Here's a detailed look at how joins are performed and how side data is distributed in a MapReduce environment.

**Joins in MapReduce**

1. **Map-Side Join**
    - **Description**: Both datasets to be joined are pre-sorted and partitioned in the same way, allowing them to be joined during the map phase.
    - **Process**:
        - Input data is pre-sorted and partitioned.
        - Mappers read corresponding partitions from both datasets.
        - Mappers perform the join operation locally, emitting the joined records directly.
    - **Advantages**: Efficient since it avoids the shuffle and sort phase.
    - **Disadvantages**: Requires data to be pre-sorted and partitioned, which can be cumbersome.
2. **Reduce-Side Join**
    - **Description**: Joins are performed during the reduce phase after a shuffle and sort.
    - **Process**:
        - Mappers tag records from each dataset with a source identifier.
        - Mappers emit key-value pairs with the join key as the key.
        - Reducers receive all records with the same key, sort them, and perform the join operation.
    - **Advantages**: Simple to implement, works even if data is not pre-sorted.
    - **Disadvantages**: Requires the shuffle and sort phase, which can be resource-intensive.

3. **Broadcast Join (Replicated Join)**
    - **Description**: A small dataset is replicated across all nodes, and the join is performed in the map phase.
    - **Process**:
        - The small dataset (side data) is distributed to all mapper nodes.
        - Mappers read the large dataset and perform the join using the in-memory side data.
    - **Advantages**: Efficient for small side data, avoids the shuffle and sort phase.
    - **Disadvantages**: Only feasible if the side data fits into memory on each mapper node.

**Side Data Distribution**

Side data refers to additional datasets that are used in conjunction with the primary datasets being processed. Efficient distribution of side data is crucial for optimizing join operations.

1. **Distributed Cache**
    - **Description**: A mechanism provided by Hadoop to distribute side data to all nodes in the cluster.
    - **Usage**:
        - Side data files are added to the distributed cache.
        - Nodes fetch the side data from the cache and store it locally.
    - **Advantages**: Simplifies distribution of side data, ensures consistency.
    - **Disadvantages**: Limited by the size of the side data that can be efficiently distributed.
2. **HDFS-Based Distribution**
    - **Description**: Side data is stored in HDFS and accessed by nodes as needed.
    - **Usage**:
        - Side data is read from HDFS by mappers or reducers.
    - **Advantages**: Can handle larger side data compared to distributed cache.
    - **Disadvantages**: May incur additional I/O overhead, especially if accessed frequently.

**Detailed Example: Reduce-Side Join**

Consider two datasets:

- **Users**: (user_id, user_name)
- **Orders**: (order_id, user_id, product_id)

The goal is to join these datasets on user_id to get user details along with their orders.

1. **Mapper**:
    - Reads both Users and Orders datasets.
    - Emits key-value pairs with user_id as the key.
    - For Users: Emits (user_id, ("user", user_name))

- For Orders: Emits (user_id, ("order", order_id, product_id))

python
Copy code
```
def map(key, value):
  if is_user_record(value):
    user_id, user_name = parse_user_record(value)
    emit(user_id, ("user", user_name))
  elif is_order_record(value):
    order_id, user_id, product_id = parse_order_record(value)
    emit(user_id, ("order", order_id, product_id))
```

2. **Reducer**:
   - Receives all records with the same user_id.
   - Sorts records by type (user or order).
   - Joins user details with orders.

python
Copy code
```
def reduce(user_id, values):
  user_info = None
  orders = []
  for value in values:
    if value[0] == "user":
      user_info = value[1]
    elif value[0] == "order":
      orders.append((value[1], value[2]))
  if user_info is not None:
    for order_id, product_id in orders:
      emit(user_id, (user_info, order_id, product_id))
```

**Summary**

Joins in MapReduce can be performed efficiently using different strategies, such as map-side join, reduce-side join, and broadcast join, depending on the size and distribution of the datasets. Side data distribution is managed using mechanisms like the distributed cache or HDFS, ensuring that small datasets are available locally on nodes for efficient processing. These techniques enable MapReduce to handle complex data processing tasks involving multiple datasets.

## Writing map reduce programs

Writing MapReduce programs involves creating a Mapper and a Reducer class, configuring the job, and managing the input and output formats. Here's a step-by-step guide to writing a simple MapReduce program using the Hadoop framework, along with an example.

**Components of a MapReduce Program**

1. **Mapper Class**: Processes input data and produces intermediate key-value pairs.
2. **Reducer Class**: Processes intermediate key-value pairs to produce final output.
3. **Driver Class**: Configures and runs the MapReduce job.

**Example: Word Count Program**

**Step 1: Set Up Your Development Environment**

1. Install Hadoop (if not already installed).
2. Set up your project structure.
3. Include the Hadoop libraries in your project's classpath.

**Step 2: Create the Mapper Class**

The Mapper class processes the input data and produces intermediate key-value pairs. In this example, the input data is a text file, and the mapper outputs each word with a count of 1.

java

Copy code

```java
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.StringTokenizer;

public class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

**Step 3: Create the Reducer Class**

The Reducer class processes the intermediate key-value pairs generated by the mapper and produces the final output. In this example, it sums up the counts for each word.

java
Copy code
```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

**Step 4: Create the Driver Class**

The Driver class configures and runs the MapReduce job.

java
Copy code
```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
```

```
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class); // Optional combiner
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

**Step 5: Compile and Run the Program**

1. **Compile the Java Code**:

bash

Copy code

```
javac -classpath $(hadoop classpath) -d wordcount_classes TokenizerMapper.java IntSumReducer.java WordCount.java
```

2. **Create a JAR File**:

bash

Copy code

```
jar -cvf wordcount.jar -C wordcount_classes/ .
```

3. **Run the Hadoop Job**:

bash

Copy code

```
hadoop jar wordcount.jar WordCount /input_path /output_path
```

**Detailed Example**

**Mapper Class (TokenizerMapper.java)**

java

Copy code

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.StringTokenizer;

public class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
```

```
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
       StringTokenizer itr = new StringTokenizer(value.toString());
       while (itr.hasMoreTokens()) {
          word.set(itr.nextToken());
          context.write(word, one);
       }
   }
}
```

**Reducer Class (IntSumReducer.java)**

java

Copy code

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
   private IntWritable result = new IntWritable();

   public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
          sum += val.get();
       }
       result.set(sum);
       context.write(key, result);
   }
}
```

**Driver Class (WordCount.java)**

java

Copy code

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
```

```java
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

**Running the Program**

1. **Ensure Hadoop is Running**: Start your Hadoop cluster if it's not already running.
2. **Prepare Input Data**: Place your input data in HDFS.

bash

Copy code

```bash
hdfs dfs -mkdir /input
hdfs dfs -put /path/to/local/input/file /input
```

3. **Execute the Job**:

bash

Copy code

```bash
hadoop jar wordcount.jar WordCount /input /output
```

4. **Check the Output**: After the job completes, check the output in HDFS.

bash

Copy code

```bash
hdfs dfs -cat /output/part-r-00000
```

**Summary**

Writing a MapReduce program involves creating Mapper, Reducer, and Driver classes. The Mapper processes input data and generates intermediate key-value pairs, while the Reducer processes these pairs to generate the final output. The Driver configures and runs the job. By following the steps outlined above, you can write, compile, and run a basic MapReduce program to perform tasks like word counting.

## Writing map reduce programs python

Writing MapReduce programs in Python can be achieved using the Hadoop Streaming API, which allows you to write the Mapper and Reducer in any language that can read from standard input and write to standard output. Here's a step-by-step guide to writing a simple MapReduce program in Python, using a word count example.

**Components of a MapReduce Program**

1. **Mapper**: Processes input data and produces intermediate key-value pairs.
2. **Reducer**: Processes intermediate key-value pairs to produce the final output.

**Example: Word Count Program in Python**

**Step 1: Write the Mapper**

The Mapper script reads input data from standard input, processes each line, and outputs key-value pairs (word and 1).

python

Copy code

```
# mapper.py
import sys

def main():
    for line in sys.stdin:
        line = line.strip()
        words = line.split()
        for word in words:
            print(f"{word}\t1")

if __name__ == "__main__":
    main()
```

**Step 2: Write the Reducer**

The Reducer script reads intermediate key-value pairs from standard input, aggregates the counts for each word, and outputs the final counts.

python

Copy code

```
# reducer.py
import sys
```

```
def main():
    current_word = None
    current_count = 0
    word = None

    for line in sys.stdin:
        line = line.strip()
        word, count = line.split('\t', 1)
        try:
            count = int(count)
        except ValueError:
            continue

        if current_word == word:
            current_count += count
        else:
            if current_word:
                print(f"{current_word}\t{current_count}")
            current_word = word
            current_count = count

    if current_word == word:
        print(f"{current_word}\t{current_count}")

if __name__ == "__main__":
    main()
```

**Step 3: Prepare Input Data**

Create an input text file with some sample data.

bash

Copy code

```
echo "hello world" > input.txt
echo "hello hadoop" >> input.txt
```

**Step 4: Run the MapReduce Job Using Hadoop Streaming**

Use the Hadoop Streaming API to run the MapReduce job with the Python scripts.

bash

Copy code

```
hadoop jar /path/to/hadoop-streaming.jar \
```

```
  -input /path/to/input.txt \
  -output /path/to/output \
  -mapper /path/to/mapper.py \
  -reducer /path/to/reducer.py \
  -file /path/to/mapper.py \
  -file /path/to/reducer.py
```

**Detailed Example**

**Mapper Script (mapper.py)**

python

Copy code

```python
import sys

def main():
  for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
      print(f"{word}\t1")

if __name__ == "__main__":
  main()
```

**Reducer Script (reducer.py)**

python

Copy code

```python
import sys

def main():
  current_word = None
  current_count = 0
  word = None

  for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
      count = int(count)
    except ValueError:
      continue
```

```
        if current_word == word:
            current_count += count
        else:
            if current_word:
                print(f"{current_word}\t{current_count}")
            current_word = word
            current_count = count

    if current_word == word:
        print(f"{current_word}\t{current_count}")

if __name__ == "__main__":
    main()
```

**Running the Program**

1. **Ensure Hadoop is Running**: Start your Hadoop cluster if it's not already running.
2. **Prepare Input Data**: Place your input data in HDFS.

bash

Copy code

```
hdfs dfs -mkdir /input
hdfs dfs -put input.txt /input
```

3. **Execute the Job**:

bash

Copy code

```
hadoop jar /path/to/hadoop-streaming.jar \
    -input /input/input.txt \
    -output /output \
    -mapper /path/to/mapper.py \
    -reducer /path/to/reducer.py \
    -file /path/to/mapper.py \
    -file /path/to/reducer.py
```

4. **Check the Output**: After the job completes, check the output in HDFS.

bash

Copy code

```
hdfs dfs -cat /output/part-00000
```

**Summary**

Writing a MapReduce program in Python involves creating Mapper and Reducer scripts that read from standard input and write to standard output. Using the Hadoop Streaming API, you

can run these scripts as part of a Hadoop job. The example above demonstrates a simple word count program, illustrating the key steps involved in setting up and executing a MapReduce job using Python.

## Deploying map-reduce python programs on Hadoop Cluster

Deploying MapReduce programs written in Python on a Hadoop cluster involves several steps. This includes setting up the Hadoop cluster, preparing your Python scripts, and running the MapReduce job using Hadoop Streaming. Here is a step-by-step guide:

**Prerequisites**

1. A working Hadoop cluster.
2. Python installed on all nodes in the Hadoop cluster.
3. Your MapReduce scripts (mapper and reducer).

**Step-by-Step Guide**

**Step 1: Set Up the Hadoop Cluster**

Make sure your Hadoop cluster is properly set up and running. You should be able to access the Hadoop Distributed File System (HDFS) and run Hadoop jobs.

**Step 2: Prepare Your Python Scripts**

Create your mapper.py and reducer.py scripts. Ensure they are working correctly on your local machine before deploying them to the cluster.

**Mapper Script (mapper.py):**

python

Copy code

```
import sys

def main():
    for line in sys.stdin:
        line = line.strip()
        words = line.split()
        for word in words:
            print(f"{word}\t1")

if __name__ == "__main__":
    main()
```

**Reducer Script (reducer.py):**

python

Copy code

```
import sys

def main():
```

```
    current_word = None
    current_count = 0
    word = None

    for line in sys.stdin:
        line = line.strip()
        word, count = line.split('\t', 1)
        try:
            count = int(count)
        except ValueError:
            continue

        if current_word == word:
            current_count += count
        else:
            if current_word:
                print(f"{current_word}\t{current_count}")
            current_word = word
            current_count = count

    if current_word == word:
        print(f"{current_word}\t{current_count}")

if __name__ == "__main__":
    main()
```

**Step 3: Upload Data to HDFS**

Upload your input data to HDFS. For example, if you have a file named input.txt:

bash

Copy code

```
hdfs dfs -mkdir /input
hdfs dfs -put input.txt /input
```

**Step 4: Upload Python Scripts to HDFS**

Upload your Python scripts to HDFS so that they can be accessed by all nodes in the cluster.

bash

Copy code

```
hdfs dfs -put mapper.py /input
hdfs dfs -put reducer.py /input
```

**Step 5: Run the Hadoop Streaming Job**

Run the Hadoop Streaming job, specifying the paths to your Python scripts and input/output directories in HDFS.

bash

Copy code

```
hadoop jar /path/to/hadoop-streaming.jar \
  -input /input/input.txt \
  -output /output \
  -mapper /input/mapper.py \
  -reducer /input/reducer.py \
  -file /path/to/local/mapper.py \
  -file /path/to/local/reducer.py
```

Explanation of the command:

- -input /input/input.txt: Specifies the input file or directory in HDFS.
- -output /output: Specifies the output directory in HDFS.
- -mapper /input/mapper.py: Specifies the mapper script to use.
- -reducer /input/reducer.py: Specifies the reducer script to use.
- -file /path/to/local/mapper.py: Uploads the mapper script from the local file system.
- -file /path/to/local/reducer.py: Uploads the reducer script from the local file system.

**Step 6: Check the Output**

After the job completes, check the output in HDFS.

bash

Copy code

```
hdfs dfs -ls /output
hdfs dfs -cat /output/part-00000
```

**Step 7: Clean Up**

If needed, you can delete the output directory in HDFS to clean up.

bash

Copy code

```
hdfs dfs -rm -r /output
```

**Summary**

Deploying a MapReduce program written in Python on a Hadoop cluster involves preparing your Python scripts, uploading them and your input data to HDFS, and running the job using Hadoop Streaming. This allows you to leverage the power of Hadoop's distributed computing capabilities while writing your mapper and reducer in Python.