Big Data Analytics
Lecture 7: The MapReduce Framework

Based on Slides by Dr. Tariq Mahmood

Fall 2025

Abstract

These notes provide a comprehensive examination of the MapReduce programming model, a core component of the original Hadoop ecosystem. We will explore the lifecycle of a MapReduce job, the mechanics of map and reduce tasks, the critical concept of data locality, and the role of YARN in scheduling. Special attention is given to the data flow between mappers and reducers, known as the "shuffle," and optimization techniques using combiner functions. The notes are supplemented with practical examples to illustrate these concepts.

Contents

1 The MapReduce Job . . . **2**
1.1 Job Execution and Task Scheduling . . . 2
1.2 Fault Tolerance . . . 2

2 The Map Phase in Detail . . . **2**
2.1 Input Splits: The Unit of Work for Mappers . . . 2
2.1.1 The Importance of Fine-Grained Splits for Load Balancing . . . 2
2.2 Optimal Split Size and HDFS Blocks . . . 3
2.3 Data Locality: Moving Computation to the Data . . . 3
2.4 Map Task Output . . . 3

3 The Shuffle and Reduce Phase . . . **3**
3.1 Reduce Tasks and Data Locality . . . 3
3.2 The Shuffle: Data Flow from Map to Reduce . . . 4
3.2.1 Multiple Reducers and Partitioning . . . 4
3.3 Reduce Task Output . . . 4

4 Optimization: Combiner Functions . . . **4**
4.0.1 Combiner Example: Finding the Maximum Temperature . . . 5

5 MapReduce in Practice: Code Examples . . . **5**
5.1 Java Job Configuration with a Combiner . . . 5
5.2 Python with Hadoop Streaming . . . 5
5.2.1 Map Function in Python . . . 5
5.2.2 Reduce Function in Python . . . 6

6 Summary of Key Hadoop Components . . . **6**

1 The MapReduce Job

MapReduce is a programming model for processing large datasets with a parallel, distributed algorithm on a cluster. A MapReduce job typically consists of three components:

1. **Input Data:** The large dataset to be processed, usually stored in HDFS.
2. **MapReduce Program:** The user-defined code containing the logic for the 'map' and 'reduce' functions.
3. **Configuration Information:** Parameters that control the job execution, such as the number of reducers, input/output paths, and data formats.

1.1 Job Execution and Task Scheduling

Hadoop runs a MapReduce job by breaking it down into smaller units of work called **tasks**. There are two types of tasks:

- **Map Tasks:** Process a piece of the input data and generate intermediate key-value pairs.
- **Reduce Tasks:** Process the intermediate data from the map tasks to produce the final output.

These tasks are scheduled and managed by **YARN (Yet Another Resource Negotiator)**, Hadoop's cluster resource management system. YARN's ApplicationMaster for MapReduce is responsible for negotiating resources (CPU, memory) from the ResourceManager and working with NodeManagers to launch and monitor the tasks on the nodes in the cluster.

1.2 Fault Tolerance

A key feature of the MapReduce framework is its built-in fault tolerance. In a large cluster, node or task failures are common.

- If a task fails (e.g., due to a software bug or a node crashing), YARN will detect the failure and **automatically reschedule the task to run on a different, healthy node**. This ensures that the overall job can complete even in the presence of hardware failures, without requiring manual intervention.

2 The Map Phase in Detail

2.1 Input Splits: The Unit of Work for Mappers

To enable parallel processing, Hadoop divides the input to a MapReduce job into fixed-size pieces called **input splits** or simply **splits**.

- Hadoop creates **one map task for each split**.
- The map function is then executed for each record within that split.
- Having many splits means the time taken to process a single split is small compared to the time required to process the entire input dataset.

2.1.1 The Importance of Fine-Grained Splits for Load Balancing

Processing splits in parallel across the cluster leads to better load balancing, especially when the splits are small and numerous.

- **Heterogeneous Clusters:** In a real-world cluster, some machines are faster than others. A faster machine will be able to process its assigned split more quickly and can then request a new split to process. This dynamic allocation ensures that faster machines perform proportionally more work, leading to a shorter overall job completion time.
- **Failure and Concurrency:** When a task fails and needs to be rescheduled, or when multiple jobs are running concurrently, good load balancing is essential to ensure resources are used efficiently and jobs are not starved.

- The quality of load balancing improves as the splits become more fine-grained, allowing for a more even distribution of work.

2.2 Optimal Split Size and HDFS Blocks

While small splits are good for load balancing, there is a trade-off.

- If splits are too small, the overhead of managing the splits and creating map tasks (scheduling, setup, etc.) can begin to dominate the total job execution time.
- A good split size tends to be **the size of an HDFS block**, which is 128 MB by default.

This choice is logical and directly tied to the concept of data locality. HDFS stores a file as a sequence of blocks, and the locations of these blocks are known to the NameNode. By aligning the split size with the block size, Hadoop can ensure that a single split corresponds to a single block of data residing on a specific set of DataNodes.

2.3 Data Locality: Moving Computation to the Data

The guiding principle for MapReduce performance is **data locality**: running the computation on the node where the data resides, rather than moving large amounts of data across the network to the computation. The MapReduce scheduler in YARN strives to achieve this.

The scheduler attempts to place a map task according to the following hierarchy of preference:

1. **Data-Local:** The scheduler will first try to place the map task on a node that stores a replica of the input split's data. This is the most efficient scenario as no data is transferred over the network.
2. **Rack-Local:** If all nodes holding the data replica are busy running other tasks, the scheduler will look for a free slot on a different node within the **same rack** as one of the replicas. This requires transferring the data block over the rack's internal switch, which is faster than transferring it across racks.
3. **Off-Rack:** Very occasionally, if the local nodes and the entire rack are busy, the scheduler will be forced to run the task on a node in a different rack. This is the least efficient option, as it requires transferring the data block across the core network switches.

If a split spans two blocks, it is highly unlikely that a single HDFS node stores both blocks. Therefore, to process that split, part of its data would need to be transferred across the network to the node running the map task, undermining the goal of data locality. This is why aligning split size with block size is the optimal strategy.

2.4 Map Task Output

The output of a map task is intermediate data, meaning it is not the final result of the job.

- Map tasks write their output (key-value pairs) to the **local disk** of the node they are running on, not to HDFS.
- Since this data will be processed by the reduce tasks and then discarded, storing it in HDFS with the standard replication factor of three would be **overkill** and would generate unnecessary network traffic and storage overhead.
- **Fault Tolerance:** If a node running a map task fails before its output has been consumed by the reduce task, Hadoop simply reruns that map task on another node to re-create the intermediate map output.

3 The Shuffle and Reduce Phase

3.1 Reduce Tasks and Data Locality

Unlike map tasks, **reduce tasks do not have the advantage of data locality**. The input to a single reduce task is typically the output from *all* mappers. This means the intermediate data must be moved from where it was created (the mapper nodes) to where it will be processed (the reducer node).

3.2 The Shuffle: Data Flow from Map to Reduce

The entire process of moving data from the mappers to the reducers is colloquially known as the **"shuffle."** It is a complex and critical phase that has a major impact on the efficiency of a MapReduce job. figure[h!] MapReduce Data Flow with a Single Reduce Task

Description of the data flow (based on slide 8): The input data in HDFS is divided into three splits. Each split is processed by a separate map task. The output of each map task is first sorted locally by key. These sorted outputs are then copied (shuffled) across the network to a single location. There, they are merged into a single, larger sorted dataset. This merged data is then fed to a single reduce task. The output of the reducer ('part-0') is written back to HDFS, where it is replicated for fault tolerance.

3.2.1 Multiple Reducers and Partitioning

The number of reduce tasks is not determined by the input data size but is instead specified independently by the user in the job configuration.

- When there are multiple reducers, each map task must partition its output, creating one partition for each reduce task.
- The partitioning is controlled by a **partitioner function**. The default partitioner typically uses a hash function on the key ('hash(key) mod numReduceTasks') to determine which partition (and thus which reducer) a key-value pair should be sent to.
- The crucial guarantee is that **all records for any given key will end up in the same partition** and will therefore be processed by the same reduce task.

3.3 Reduce Task Output

The final output of the reduce tasks is the result of the entire job.

- This output is considered persistent and is normally stored in **HDFS for reliability**.
- For each HDFS block of the reduce output, the first replica is typically stored on the local node where the reduce task ran. Additional replicas are stored on off-rack nodes to protect against rack failure.
- This process consumes network bandwidth, but no more than a normal HDFS write operation.

4 Optimization: Combiner Functions

Many MapReduce jobs are limited by the network bandwidth available during the shuffle phase. It therefore pays to minimize the data transferred between map and reduce tasks. Hadoop provides an optimization for this called a **combiner function**.

- A combiner function is a user-specified function that is run on the map output before it is shuffled. It acts as a "mini-reducer" on the local map output.
- The output of the combiner function, which is typically smaller than the original map output, forms the input to the reduce function.
- Because the combiner is purely an optimization, Hadoop **does not provide a guarantee** of how many times it will call the combiner for a particular map output record, if at all (it could be zero, one, or many times).
- This implies a critical requirement: the operation must be **associative and commutative**. Calling the combiner function zero, one, or many times must produce the same final output from the reducer.

4.0.1 Combiner Example: Finding the Maximum Temperature

Consider a job to find the maximum temperature reading for the year 1950. The data is processed by two map tasks.

- **Map 1 Output:** ‘(1950, 0), (1950, 20), (1950, 10)‘
- **Map 2 Output:** ‘(1950, 25), (1950, 15)‘

Scenario 1: Without a Combiner

1. All 5 key-value pairs are transferred across the network during the shuffle.
2. The reducer receives: ‘(1950, [0, 20, 10, 25, 15])‘.
3. The reducer computes ‘max(0, 20, 10, 25, 15)‘ and outputs ‘(1950, 25)‘.

Scenario 2: With a Combiner (using ‘max‘ as the function)

1. The combiner runs on Map 1’s output, computing ‘max(0, 20, 10) = 20‘. It outputs ‘(1950, 20)‘.
2. The combiner runs on Map 2’s output, computing ‘max(25, 15) = 25‘. It outputs ‘(1950, 25)‘.
3. Only these 2 aggregated key-value pairs are transferred during the shuffle.
4. The reducer receives: ‘(1950, [20, 25])‘.
5. The reducer computes ‘max(20, 25)‘ and outputs ‘(1950, 25)‘.

The final result is identical, but the amount of data shuffled across the network was significantly reduced. In this case, the ‘max‘ function is both associative and commutative, so the reducer logic can be reused as the combiner. This is also true for functions like ‘sum‘ and ‘count‘, but not for functions like ‘average‘.

5 MapReduce in Practice: Code Examples

5.1 Java Job Configuration with a Combiner

In a Java MapReduce program, specifying a combiner is a single line in the job driver code. The same Reducer class can often be used as the Combiner.

```java
// ... job setup ...
job.setMapperClass(MaxTemperatureMapper.class);

// Use the same Reducer logic for the Combiner
job.setCombinerClass(MaxTemperatureReducer.class);

job.setReducerClass(MaxTemperatureReducer.class);
// ... output setup and job submission ...
```

Listing 1: Setting a Combiner in a Java MapReduce Job

5.2 Python with Hadoop Streaming

Hadoop Streaming allows developers to use any executable or script as a mapper or reducer. The framework communicates with the scripts via standard input and standard output.

5.2.1 Map Function in Python

The following script reads weather data line by line from standard input, extracts the year and temperature, and prints them to standard output as a tab-separated key-value pair.

```python
#!/usr/bin/env python
import re
import sys

for line in sys.stdin:
    val = line.strip()
    year = val[15:19]
    temp = val[87:92]
    quality = val[92:93]

    # Check for valid temperature and quality readings
    if (temp != "+9999" and re.match("[01459]", quality)):
        print "%s\t%s" % (year, temp)
```

Listing 2: max_temperature_map.py

5.2.2 Reduce Function in Python

The reduce script receives the sorted output from the mappers via standard input. It iterates through the lines, keeping track of the maximum temperature for the current key (year). When the key changes, it prints the maximum value for the previous key.

```python
#!/usr/bin/env python
import sys

last_key = None
max_val = -sys.maxint - 1

for line in sys.stdin:
    (key, val_str) = line.strip().split("\t")
    val = int(val_str)

    if last_key and last_key != key:
        print "%s\t%s" % (last_key, max_val)
        last_key = key
        max_val = val
    else:
        last_key = key
        max_val = max(max_val, val)

if last_key:
    print "%s\t%s" % (last_key, max_val)
```

Listing 3: max_temperature_reduce.py

Note: The 'sort' step in local testing ('cat data — map — sort — reduce') is crucial because it simulates the Hadoop Shuffle phase, which guarantees that the input to the reducer is sorted by key.

6 Summary of Key Hadoop Components

HDFS A distributed, scalable, and fault-tolerant file system. **Advantages:** Handles massive datasets on commodity hardware, provides high throughput for streaming reads, and enables data locality. **Disadvantages:** High latency for random access, not efficient for small files.

YARN The cluster resource manager for Hadoop. **Advantages:** Decouples resource management from the processing model, allowing multiple frameworks (MapReduce, Spark, Flink, etc.) to run on the same cluster. Manages scheduling and fault tolerance of tasks. **Disadvantages:** Can be complex to configure and tune.

MapReduce A programming model for batch processing. **Advantages:** Simplifies parallel programming, provides automatic scalability and fault tolerance. **Disadvantages:** Inherently high latency (batch-oriented), can be verbose, and is often less efficient than modern in-memory frameworks like Spark for many use cases.

References

[1] Apache Hadoop Documentation. https://hadoop.apache.org/

[2] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 29-43. 2003.

[3] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51, no. 1 (2008): 107-113.