Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قسم الانظمة الطبية الذكية

Lecture (4 & 5): MapReduce

Subject: Big Data Analysis in Healthcare
Level: Fourth
Lecturer: Asst. Lecturer Qusai AL-Durrah
Duration: Two hours

1. Introduction

MapReduce represents the computational core of Hadoop, enabling massive datasets to be processed efficiently across clusters of commodity servers. Developed originally by Google and later adopted into Apache Hadoop, the model abstracts the complexity of distributed processing, fault tolerance, and synchronization.
Instead of writing intricate multi-threaded or network-aware code, developers define two simple functions — **Map** and **Reduce** — and Hadoop manages all the parallelization automatically.

In healthcare, where terabytes of clinical data, sensor streams, and genomic sequences are produced daily, MapReduce provides a scalable approach for data aggregation, pattern detection, and predictive analytics.
It is particularly relevant to the Smart Medical Systems Department because it supports large-scale processing of medical images, diagnostic reports, and continuous patient monitoring data.

2. Learning Outcomes

By the end of this lecture, students will be able to:

1. **Explain:** the workflow and key principles of the MapReduce model.
2. **Differentiate**: between the Map and Reduce phases in Hadoop's data processing pipeline.
3. **Describe**: the use of key–value pairs and the flow of data between mappers and reducers.
4. **Discuss**: the roles of combiners and partitioners in optimizing MapReduce jobs.
5. **Apply**: MapReduce to analyze sample healthcare datasets and derive aggregated results.

3. Conceptual Framework of MapReduce

The MapReduce paradigm is based on the principle of **divide and conquer**. A dataset is split into smaller chunks that are processed in parallel across many nodes.
Each node performs a *map* operation that transforms input records into intermediate key–value pairs, followed by a *reduce* operation that aggregates all values associated with the same key.

Formally:

- **Map Function:** (key1, value1) → list(key2, value2)
  Converts raw input data into intermediate representations.
- **Reduce Function:** shuffle (key2, list(value2)) → list(value3)
  Consolidates intermediate values to produce the final output.

This abstraction hides the complexity of communication, synchronization, and fault handling, allowing developers to focus purely on the computation logic.

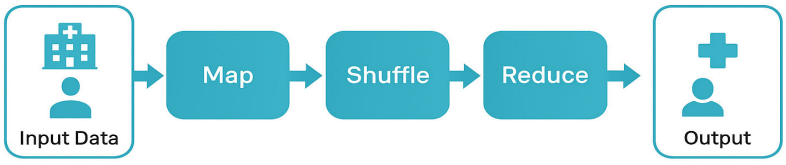


Figure 1: *Conceptual View of MapReduce Workflow*

4. Detailed Architecture and Data Flow

The Hadoop MapReduce engine executes jobs in distinct stages:

1. **Input Splitting**: Large input files in HDFS are divided into *InputSplits*, each typically 64 MB or 128 MB.
2. **RecordReader:** It communicates with the inputSplit. And then converts the data into key-value pairs suitable for reading by the Mapper. RecordReader

by default uses TextInputFormat to convert data into a key-value pair. Meaning it communicates to the InputSplit until the completion of file reading. It assigns byte offset to each line present in the file. Then, these key-value pairs are further sent to the mapper for further processing.

3. **Mapping**: Each split is processed by a mapper, which emits intermediate key–value pairs.
4. **Combining (optional):** Combiner is Mini-reducer which performs local aggregation on the mapper's output. It minimizes the data transfer between mapper and reducer. So, when the combiner functionality completes, framework passes the output to the partitioner for further processing.
5. **Partitioning:** Partitioner comes into the existence if we are working with more than one reducer. It takes the output of the combiner and performs partitioning. After that, each partition is sent to a reducer. Partitioning in MapReduce execution allows even distribution of the map output over the reducer.
6. **Shuffling and Sorting:** After partitioning, the output is shuffled to the reduce node. The shuffling is the physical movement of the data which is done over the network. As all the mappers finish and shuffle the output on the reducer nodes. Then framework merges this intermediate output and sort. This is then provided as input to reduce phase.
7. **Reducing**: Reducer then takes set of intermediate key-value pairs produced by the mappers as the input. After that runs a reducer function on each of them to generate the output. The output of the reducer is the final output. Then framework stores the output on HDFS.
8. **RecordWriter:** It writes these output key-value pair from the Reducer phase to the output files.

This process ensures *data locality*: computations occur on the same nodes that store the data, minimizing costly network transfer.

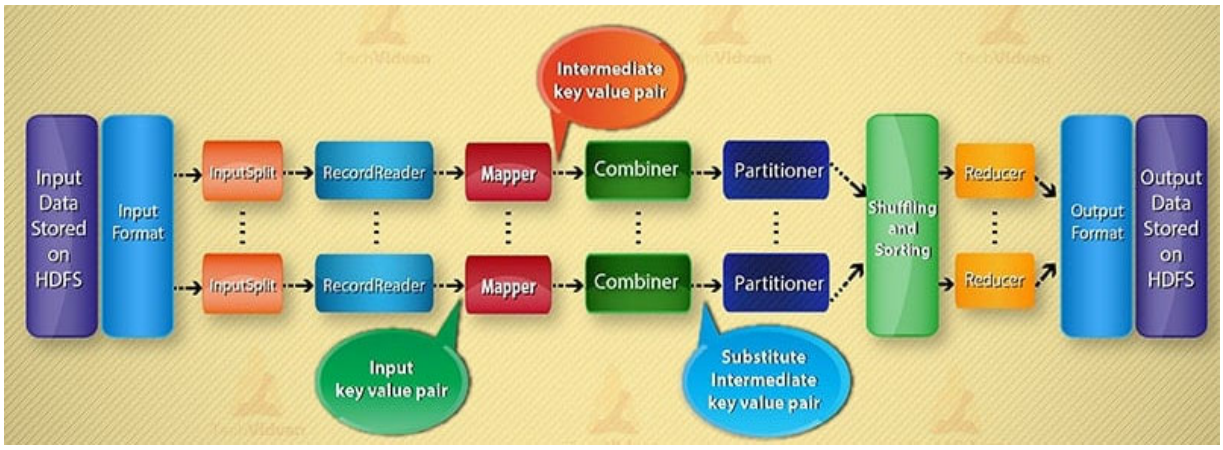


Figure 2: *MapReduce Execution Pipeline*

4. JobTracker and TaskTracker in Classic MapReduce

Before Hadoop 2.0 and YARN, Hadoop's MapReduce system relied on two main daemons — the **JobTracker** and the **TaskTracker** — which coordinated all distributed computations across the cluster.

4.1 JobTracker

The **JobTracker** acted as the **master** node responsible for the **coordination and management** of MapReduce jobs.
Its core functions included:

- **Task Scheduling:** Assigning map and reduce tasks to available TaskTrackers.
- **Job Monitoring:** Tracking task progress and updating overall job status.
- **Fault Tolerance:** Detecting node failures and reassigning unfinished tasks.
- **Resource Management:** Keeping record of available TaskTrackers and job queues.

Essentially, the JobTracker coordinated the entire cluster — ensuring jobs were balanced, monitored, and completed successfully.

4.2 TaskTracker

Each worker node in the cluster ran a **TaskTracker** daemon that performed the assigned tasks.
Its responsibilities included:

- **Task Execution:** Running map or reduce operations as instructed by the JobTracker.
- **Progress Reporting:** Sending regular *heartbeats* back to the JobTracker to confirm status.
- **Resource Control:** Managing a fixed number of *task slots* (e.g., two map slots and two reduce slots per node).
- **Error Handling:** Retrying failed tasks locally before reporting them as failed globally.
- **Job completion and cleanup:** Upon completion, TaskTrackers reported final status and cleaned up temporary data.

If a TaskTracker stopped responding, the JobTracker automatically reassigned its tasks to other healthy nodes.

4.3 Communication Flow

1. **Job Submission:** The client submits a job to the JobTracker.
2. **Task Assignment:** The JobTracker divides the job into smaller tasks and distributes them to TaskTrackers.
3. **Progress Updates:** TaskTrackers send *heartbeat* signals and task status reports back to the JobTracker.
4. **Completion:** When all tasks are finished, the JobTracker marks the job as complete and writes results to HDFS.

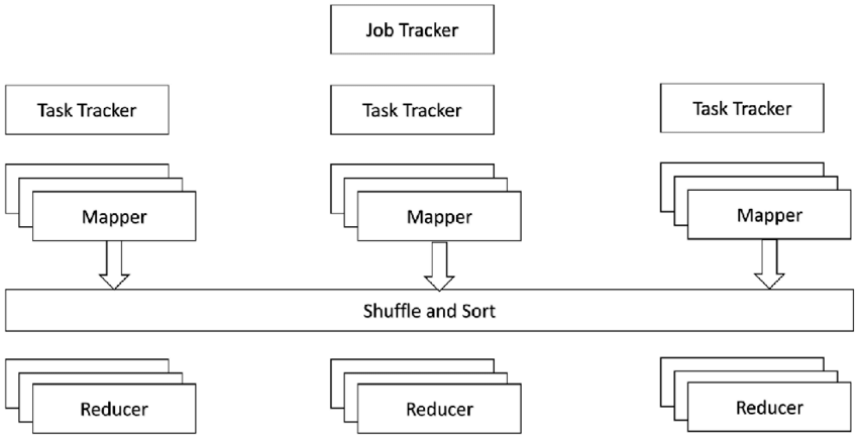


Figure 3: *JobTracker–TaskTracker Communication Flow*

4.4 Transition to YARN

As clusters grew larger (beyond 4,000 nodes), the JobTracker became a performance bottleneck.
Hadoop 2.0 replaced this architecture with **YARN (Yet Another Resource Negotiator)**, which divided the JobTracker's roles into:

- **ResourceManager:** Manages cluster-wide resource allocation.
- **ApplicationMaster:** Oversees the lifecycle of a single MapReduce job.

This design allowed for improved scalability, better fault isolation, and support for multiple processing frameworks (e.g., Spark, Tez, MapReduce v2) running on the same cluster.

6. Key System Components

Component	Function
JobTracker/ ApplicationMaster	Coordinates MapReduce job execution (in YARN, ApplicationMaster replaces JobTracker).
TaskTracker / NodeManager	Executes individual map and reduce tasks on worker nodes.
Mapper	Processes input records and produces intermediate key–value pairs.
Combiner	Performs local aggregation to reduce data sent over the network.
Partitioner	Determines how intermediate keys are assigned to reducers.
Reducer	Aggregates all values for each key and writes final output to HDFS.

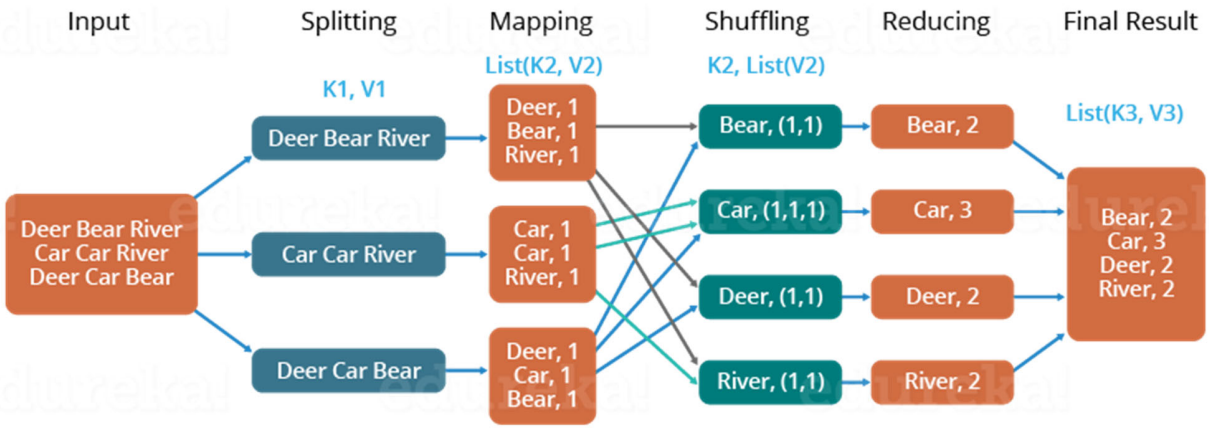
7. MapReduce Examples

7.1 Word Count Example:

Let us understand, how a MapReduce works by taking an example where I have a text file called example.txt whose contents are as follows:

Dear, Bear, River, Car, Car, River, Deer, Car and Bear

Now, suppose, we have to perform a word count on the sample.txt using MapReduce. So, we will be finding the unique words and the number of occurrences of those unique words.



- First, we divide the input into three splits as shown in the figure. This will distribute the work among all the map nodes.
- Then, we tokenize the words in each of the mappers and give a hardcoded value (1) to each of the tokens or words. The rationale behind giving a hardcoded value equal to 1 is that every word, in itself, will occur once.
- Now, a list of key-value pair will be created where the key is nothing but the individual words and value is one. So, for the first line (Dear Bear River) we have 3 key-value pairs – Dear, 1; Bear, 1; River, 1. The mapping process remains the same on all the nodes.
- After the mapper phase, a partition process takes place where sorting and shuffling happen so that all the tuples with the same key are sent to the corresponding reducer.
- So, after the sorting and shuffling phase, each reducer will have a unique key and a list of values corresponding to that very key. For example, Bear, [1,1]; Car, [1,1,1].., etc.
- Now, each Reducer counts the values which are present in that list of values. As shown in the figure, reducer gets a list of values which is [1,1] for the key Bear. Then, it counts the number of ones in the very list and gives the final output as – Bear, 2.

- Finally, all the output key/value pairs are then collected and written in the output file.

7.2 Applying MapReduce on MovieLens Dataset

The **MovieLens Dataset** is commonly used to demonstrate distributed data processing concepts such as aggregation, sorting, and reduction. The dataset contains the following columns:

USER_ID	MOVIE_ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Step 1: First we have to map the values , it is happen in 1st phase of Map Reduce model.

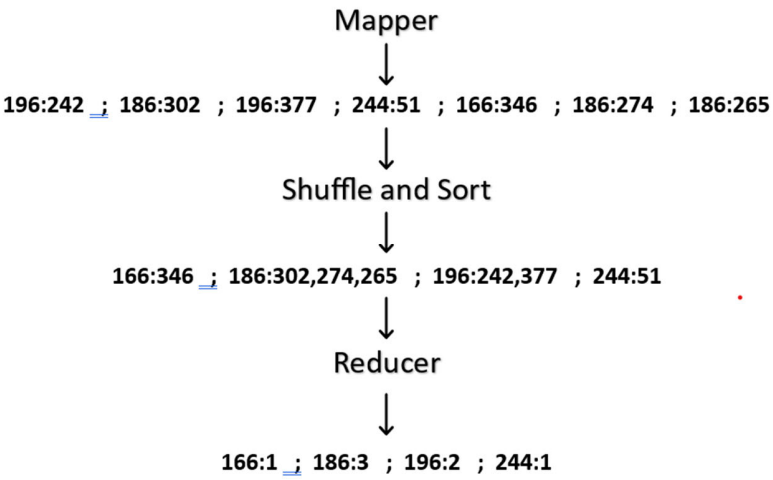
196:242 ; 186:302 ; 196:377 ; 244:51 ; 166:346 ; 186:274 ; 186:265

Step 2*:* After Mapping we have to shuffle and sort the values.

166:346 ; 186:302,274,265 ; 196:242,377 ; 244:51

Step 3*:* After completion of step1 and step2 we have to reduce each key's values.

Now, put all values together



Code:

```
from mrjob.job import MRJob
from mrjob.step import MRStep
class RatingsBreak(MRJob):
    def steps(self):
        return [
            MRstep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]
        # MAPPER CODE
    def mapper_get_ratings(self, _, line):
         (User_id, Movie_id, Rating, Timestamp) = line.split('/t')
         yield rating,
        # REDUCER CODE
```

```
def reducer_count_ratings(self, key, values):
    yield key, sum(values)
```

8. Performance Optimization Concepts

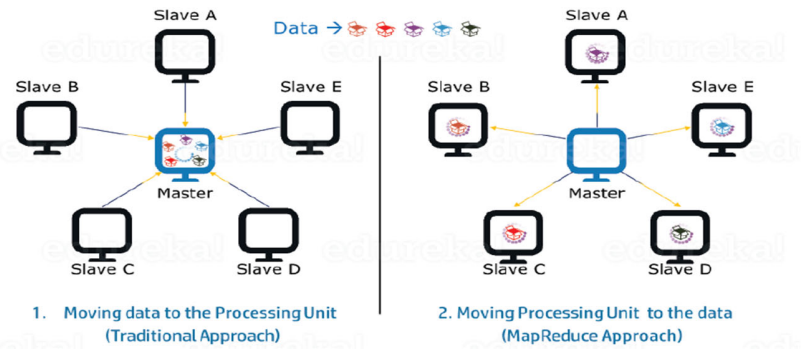
1. **Combiners** – Act as "mini-reducers" to minimize shuffle volume.
2. **Partitioners** – Ensure balanced distribution of keys across reducers.
3. **Speculative Execution** – Hadoop may duplicate slow tasks to avoid bottlenecks.
4. **Counters and Metrics** – Monitor job progress and collect statistics for analysis.

8. Advantages of MapReduce

The two biggest advantages of MapReduce are:

1. Parallel Processing:

In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines. As the data is processed by multiple machines instead of a single machine in parallel, the time taken to process the data gets reduced by a tremendous amount as shown in the figure below.



2. Data Locality:

Instead of moving data to the processing unit, we are moving the processing unit to the data in the MapReduce Framework. In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed the following issues:

- Moving huge data to processing is costly and deteriorates the network performance.
- Processing takes time as the data is processed by a single unit which becomes the bottleneck.
- The master node can get over-burdened and may fail.

Now, MapReduce allows us to overcome the above issues by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

- It is very cost-effective to move processing unit to the data.
- The processing time is reduced as all the nodes are working with their part of the data in parallel.
- Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

9. Summary

MapReduce abstracts the complexities of parallel computation into two clear functions — Map and Reduce — that enable high-throughput processing of massive datasets.
It provides automatic load balancing, data locality, and fault tolerance, making it a revolutionary approach for Big Data analytics.
In healthcare, MapReduce empowers researchers to integrate and analyze multi-

institutional datasets, unlocking patterns that drive clinical insight and improve patient care.