Unit # 2

- Map Reduce Data Flow
- Hadoop I/O ,Data Integrity
- Compression ,

Map Reduce – Data Flow

Map Reduce is a programming model introduced by Google to process and generate large datasets efficiently. It enables distributed computing across massive clusters of commodity hardware and is the core of many big data processing systems like Apache Hadoop.It breaks down complex data processing tasks into simple operations: **Map** and **Reduce**. The model provides a high level of abstraction while the system handles details like data distribution, task coordination, fault tolerance, and scalability.

The data flow in MapReduce can be complex, but it essentially follows a sequence of steps that transform raw input data into the desired output through a distributed processing model.

Understanding the data flow in MapReduce helps in optimizing the performance of Hadoop applications and making effective use of its computational resources.

Following are the Key phase of Map Reduce data flow:

- ✓ Input Reading
- ✓ Mapping
- ✓ Shuffling and Sorting
- ✓ Reducing
- ✓ Output Writing

Step- 1 : Input Reading

The first step in a MapReduce job is to read the input data from HDFS (or other file systems). The input data is divided into smaller pieces called "splits." Each split is passed to a RecordReader, which converts the data into key-value pairs suitable for processing by the Mapper.

Map Reduce – Data Flow (contd)

Example :

Say we have a large dataset of social media posts, and the goal is to determine the most popular hashtags used in those posts. The dataset is stored in a Hadoop Distributed File System (HDFS). Our intention is to task is to count the occurrences of each hashtag (say #TREMORS, #Taarangana, #XEBEC) and find the most popular ones.

Input Data:

1. I love using # Taarangana in my entire duration while at IGDTUW !
2. # TREMORS is transforming perspective for many students .
3. The future is in # TREMORS is great.
4. I feel # XEBEC is becoming the new competitor for # TREMORS.
….
….
n. Amazing breakthroughs in # XEBEC !
n+1. #XEBEC is powering the creativity # IGDTUW .
n+2. # XEBEC # TREMORS # Taarangana are the most loved of #IGDTUW events .

Step- 2 : Map Phase:

In the Input phase the input data is split into chunks, and each chunk is processed independently to generate intermediate key-value pairs. The key goal of the Mapper is to **extract, filter, and transform** data from a large dataset into a format that is more meaningful or relevant for the later stages of processing.

Map – Reduce (contd)

In the Mapper function, we will:

- Extract hashtags from the text of each post.
- Emit each hashtag as a key, with the value of 1 (indicating one occurrence of that hashtag).

for line in sys.stdin: line = line.strip() # Remove extra whitespace
hashtags = hashtag_pattern.findall(line) # Find all hashtags in the line

The mapper function shall use a regular expression (#\w+) to find all hashtags in the text. This pattern matches any word that starts with a # followed by alphanumeric characters.

Hadoop tries to perform data locality optimizations by scheduling map tasks on the nodes where data is already present, minimizing network transfers.

The Mapper processes each input key-value pair and produces zero or more output key-value pairs. The output key-value pairs could represent a transformation of the input data or could derive new data from the input.

Map – Reduce (contd)

For each hashtag found, the Mapper emits a key-value pair where the key is the hashtag (e.g., # Taarangana), and the value is 1 (indicating a single occurrence of that hashtag).

Each occurrence of a hashtag results in a key-value pair where the key is the hashtag, and the value is 1.

Step- 3 : Shuffle and Sort

In this step, which is automatically handled by the MapReduce framework, the intermediate key-value pairs emitted by the Mappers are shuffled and sorted by key (the hashtags). The goal here is to group all occurrences of the same hashtag together.

For example, after this phase, all occurrences of # Taarangana ,#XEBEX and #TREMORS would be grouped like as shown in fig.

```
PS D:\Python\python37\source_code\Som
hon310/python.exe d:/Python/python37/
#Taarangana     1
#TREMORS        1
#XEBEC      1
#TREMORS    1
#Taarangana     1
#TREMORS     1
#Taarangana     1
#XEBEC        1
#TREMORS    1
#XEBEC       1
#TREMORS    1
#Taarangana     1
```

```
PS D:\Python\python37\source_code\Som\Ver
hon310/python.exe d:/Python/python37/sour
#Taarangana [1,1,1,1]
#TREMORS  [1,1,1,1,1,1,1,1,1,1,1]
#XEBEC  [1,1,1,1,1,1,1]
```

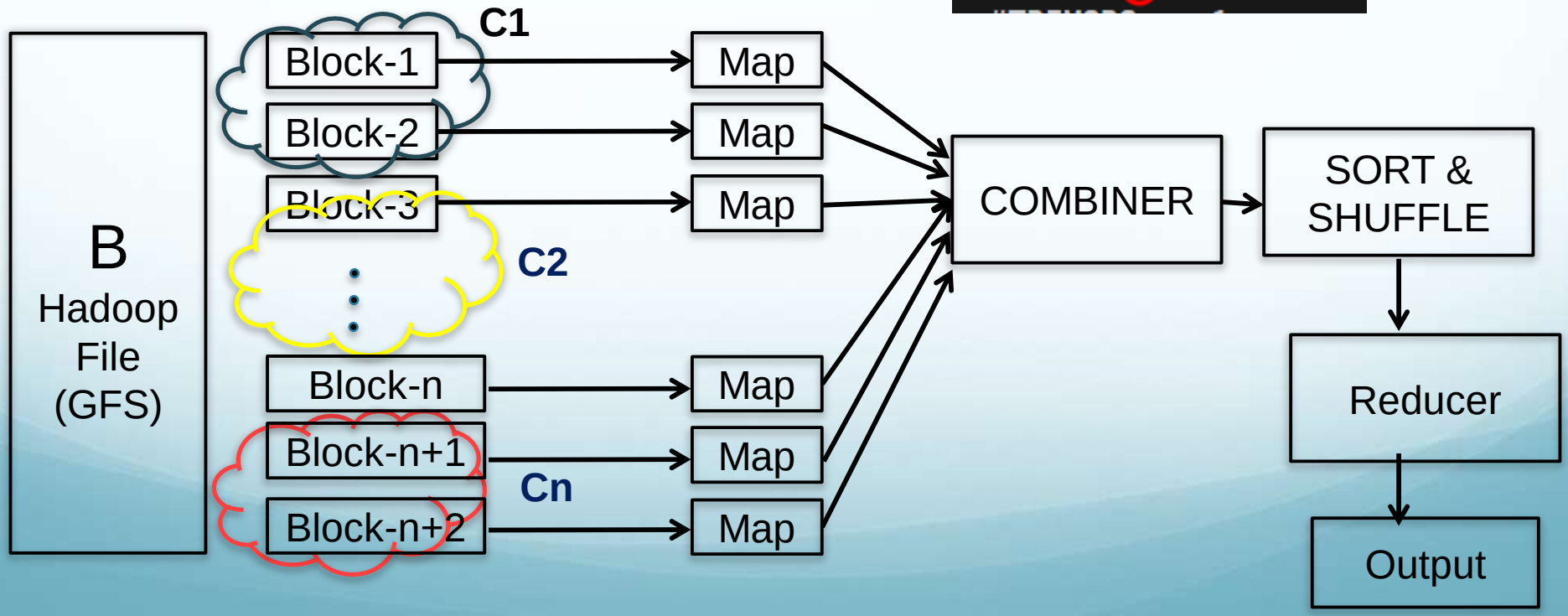
Map – Reduce (contd)

Step- 4 : The Reduce Phase

In the Reducer function, we group hashtags (key) along with the list of 1s (values).Sum up the values for each hashtag to calculate the total number of occurrences.

The final output contains the total counts for each hashtag. The output can be further processed to find the top N most popular hashtags by sorting the results in descending order of the count.

```
#TREMORS    1
#Taarangana    1
PS D:\Python\python3
hon310/python.exe d:
#Taarangana 4
#TREMORS  11
#XEBEC  7
```



Hadoop I/O

Hadoop I/O refers to the input/output mechanisms used by Hadoop to read and write data in its distributed computing environment. Since Hadoop is designed to process large-scale datasets distributed across clusters of commodity hardware, it requires efficient ways to handle data input (reading from sources like HDFS) and output (writing back to HDFS or other storage).

Hadoop provides specialized tools and APIs for performing I/O operations efficiently. These include mechanisms for serialization, compression, data integrity, and file formats, all of which are crucial for handling big data at scale.

The INPUT FORMAT and OUTPUT FORMAT classes in Hadoop determine how data is read into and written out from a Map Reduce job.

Both, INPUTFORMAT and OUTPUTFORMAT are critical components and define how Hadoop interacts with the data stored in a HDFS (Hadoop Distributed File System).

InputFormat

How the input files are split up and read in Hadoop is defined by the INPUTFORMAT. INPUTFORMAT is the first component in Map-Reduce, it is responsible for creating the splitting the input and dividing them into records.

We know , Initially, the data for a MapReduce task is stored in input files, and input files typically reside in HDFS. These files format are arbitrary, line-based log files and are in binary format. Using INPUTFORMAT we define how these input files are split and read. The INPUTFORMAT class is one of the fundamental classes in the Hadoop MapReduce framework.

Hadoop I/O (contd)

Types of InputFormat:-

1. FileInputFormat

It is the base class for all file-based INPUTFORMATS. It specifies input directory where data files are located. When Hadoop job is started, FILEINPUTFORMAT is provided with a path containing files to read. FILEINPUTFORMAT will read all files and divides these files into one or more INPUTSPLITS.

2. TextInputFormat

It is the default INPUTFORMAT of MapReduce. TEXTINPUTFORMAT treats each line of each input file as a separate record and performs no parsing. This is useful for unformatted data or line-based records like log files.

3. KeyValueTextInputFormat

It is similar to TEXTINPUTFORMAT as it also treats each line of input as a separate record. While TEXTINPUTFORMAT treats entire line as the value,the KEYV ALUE TEXTINPUTFORMAT breaks the line itself into key and value by a tab character ('/t').

4. SequenceFileInputFormat

Hadoop SEQUENCEFILEINPUTFORMAT is an INPUTFORMAT which reads sequence files. Sequence files are binary files that stores sequences of binary key-value pairs. Sequence files block-compress and provide direct serialization and deserialization of several arbitrary data types (not just text).

Hadoop I/O

5. NLineInputFormat

It is similar to TEXTINPUTFORMAT, but instead of reading the entire file line by line, it processes a fixed number of lines per split.

6. CombineTextInputFormat

It combines smaller input files into fewer splits to reduce the overhead of running many small Map tasks. It is used in scenarios like where working with large numbers of small files (e.g., sensor data or logs. In these cases the application may generate too many small input splits and slow down job performance. Here is COMBINETEXTINPUTFORMAT ideal.

7. DBInputFormat

This format allows reading from relational databases like MySQL, Oracle, or any database that supports JDBC.

OutputFormat

The OutputFormat and InputFormat functions are more or less same. The OUTPUTFORMAT class in Hadoop defines how the results of the MapReduce job are written to the storage system (typically HDFS). After the Reducer finishes processing, the output must be written back to HDFS or another storage system. The OutputFormat provides a RECORDWRITER to write the key-value pairs in a specific format to the output files.

Hadoop I/O (contd)

Hadoop supports various OUTPUTFORMATS to suit different use cases :

1. TextOutputFormat

This is the default standard OutputFomrat of Hadoop. It writes the output as plain text. Each record is written as a line of text, where the key and value are separated by a tab.

2. SequenceFileOutputFormat

This writes the output as SEQUENCEFILES, a Hadoop-specific binary file format for key-value pairs. This format is efficient for large-scale applications where the binary format provides better compression and faster reads.

3. MapFileOutputFormat

This writes the output as a sorted SequenceFile in the form of MapFiles, which are SequenceFiles with an associated index. This allows for fast lookups by key. It is useful for scenarios where fast lookups by key are required after the MapReduce job finishes, such as database indexing.

4. LazyOutputFormat

Ensures that the output files are only created if the job produces some output. It avoids creating empty output files, which can be common in large MapReduce jobs. Useful when your Reducers don't always generate output, helping to avoid creating unnecessary empty files in HDFS.

Data Integrity

- In Hadoop, data is processed in a distributed environment, meaning that pieces of data are distributed across many nodes (machines) in a cluster.
- Data integrity refers to the **Accuracy**, **Consistency,** and **Reliability** of data throughout its lifecycle. In distributed systems like Hadoop, ensuring data integrity is crucial because the data is often stored and processed across multiple machines in a cluster. Due to the large-scale nature of Hadoop, data corruption can occur due to hardware failures, network issues, or even software bugs, making data integrity a key concern.
- Hadoop addresses data integrity at various levels. The primary mechanisms used being :
  - ✓ Checksums
  - ✓ Replication
  - ✓ Data Recovery and Self-Healing
  - ✓ Fault Tolerance and DataNode Integrity Checks
  - ✓ Data Integrity in HDFS
  - ✓ Data Integrity During Processing

1. CheckSums

CheckSum is Primary Mechanism for Data Integrity in Hadoop. A checksum is a small, fixed-size block of data, typically a number, that is computed from a larger data block. This number acts as a "fingerprint" for that block of data. If any bit in the data block changes (due to corruption or failure), the checksum changes as well, allowing Hadoop to detect data corruption.

Data Integrity (contd)

2. Replication

Another method of Hadoop for data integrity is replication. Hadoop stores multiple copies of data blocks across different nodes in the cluster. If one copy of the data becomes corrupted or inaccessible due to hardware or network issues, Hadoop can retrieve the data from another replica. By default, HDFS stores **three replicas** of each data block (this can be configured). These replicas are stored on different DataNodes to prevent data loss due to node failure or corruption.

HDFS also uses **rack awareness**, meaning that it tries to distribute replicas across different racks to improve fault tolerance.

3. Data Recovery and Self-Healing

Hadoop has built-in mechanisms for self-healing when it detects data corruption or node failures. When HDFS detects corrupted blocks (via checksum validation), it initiates data recovery to ensure data integrity. The NameNode (the master node in HDFS) is notified of the corruption, and it instructs DataNodes that hold valid replicas to replicate the healthy blocks to other DataNodes. Once healthy replicas are created and the replication factor is restored, the corrupted block is deleted.

4. Fault Tolerance and DataNode Integrity Checks

This is mixing of two mechanisms. HDFS uses a **Block Scanner** to regularly scan all blocks on each DataNode to detect silent data corruption. The Block Scanner runs periodically and reads through the data blocks stored on the DataNode, verifying their checksums. If the Block Scanner finds that the checksum of a block does not match

Data Integrity (contd)

it marks the block as corrupted and informs the NameNode, which triggers the self-healing process to create a new replica from the healthy blocks.
Each **DataNode** sends regular **heartbeats** to the **NameNode** to report its status. If a DataNode fails to send a heartbeat within a specified time, the NameNode considers it dead and triggers replication of the blocks stored on that node to ensure data is not lost.

5. Checksums and Replication Together

HDFS combines checksums and replication to ensure robust data integrity. The checksum guarantees that individual blocks are not corrupted, while replication ensures that even if a block is lost or corrupted, another replica can be used to retrieve the data.

Data integrity in Hadoop is achieved through a combination of mechanisms that ensure the accuracy, consistency, and fault tolerance of data stored and processed in a distributed environment. The key mechanisms are:

Compression

Compression is a key technique in Hadoop for optimizing storage and improving the performance of data-intensive applications. Hadoop manages massive volumes of data . Compressing data can save significant amounts of disk space and reduce the time it takes to transfer data across the network. There are several stages in a Hadoop workflow where compression can be applied to optimize performance:

- I**nput Data Compression**:

As the name suggest this is compressing data before it is written to HDFS. This can significantly reduce the storage space and the time to read data during MapReduce jobs. This is done by compressing the data files before uploading them to HDFS.

- **Intermediate Data Compression (Map Output Compression)**:

The intermediate data generated by the Mapper (which is passed to the Reducer) can be compressed to save network bandwidth and reduce the time spent in the Shuffle and Sort phase.

- **Output Data Compression**:

The final output of a MapReduce job can be compressed before being written back to HDFS. This reduces storage requirements and can speed up subsequent jobs that read this output.

To enable compression in Hadoop, we need to configure several parameters in the Hadoop configuration files (mapred-site.xml, core-site.xml), depending on whether we want to apply compression to input, intermediate, or output data.

Compression (contd)

Hadoop supports various compression codecs (algorithms) that can be used depending on the specific use case. These codecs offer different tradeoffs between compression speed and compression ratio (the amount of data reduction). The key codecs used in Hadoop are:

1. Gzip (GzipCodec)
2. BZip2 (BZip2Codec)
3. Snappy (SnappyCodec)
4. LZO (LzoCodec)
5. Deflate (DeflateCodec)

1. Gzip (GzipCodec)

Gzip (GzipCodec) is one of the most commonly used compression codecs in Hadoop. Gzip stands for GNU zip and is a very popular compression tool designed for flexibility and widespread compatibility. In Hadoop, it is typically used to compress and decompress single files, offering a good balance between compression ratio and the speed of decompression and compression.

It is typically effective for text-based data common in many big data applications. While not the fastest among compression codecs, Gzip offers a reasonable trade-off between speed and efficiency.

One significant limitation of Gzip in Hadoop contexts is that it is **not splittable** This means that a large Gzip-compressed file cannot be processed in parallel by multiple Mappers in a Hadoop job and has to be processed by one Mapper in one go.

Compression (contd)

2. bzip2 (bzip2Codec)

bzip2 is a freely available, patent free (see below), high-quality data compressor. It typically compresses files to within 10% to 15% of the best available techniques (the PPM family of statistical compressors), whilst being around twice as fast at compression and six times faster at decompression.

One of the most significant advantages of BZip2 in the context of Hadoop is its splittability. Despite being a compressed format, BZip2 files can be split into multiple parts.

3. Snappy (Snappy (SnappyCodec)

Snappy (SnappyCodec), developed by Google, is a popular compression codec used extensively within the Hadoop ecosystem and beyond. It is designed to be very fast and efficient, providing a reasonable compression ratio without the computational overhead associated with algorithms like BZip2 or Gzip.

It does not aim for maximum compression instead, it aims for very high speeds. So snappy does not provide as high a compression ratio as Gzip or BZip2 but it still offers significant size reduction, typically better than that of algorithms like LZO. Its balance between speed and compression efficiency makes it suitable for scenarios where moderate compression is acceptable but speed is paramount.

Snappy is widely used inside Google, in everything from BigTable to MapReduce.

Compression (contd)

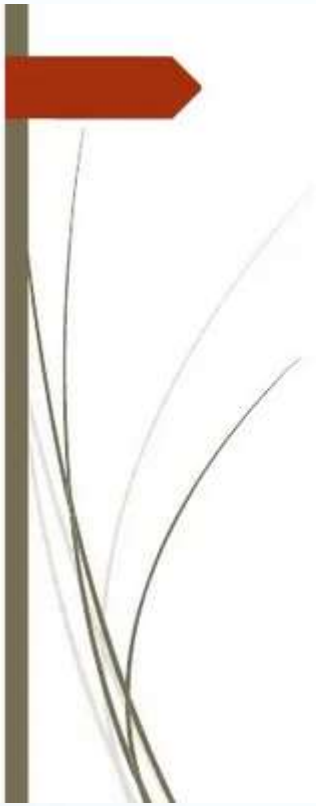
4. LZO(lzoCodec)

LZO stands for Lempel-Ziv-Oberhumer, named after its algorithm designers. It is particularly favored in scenarios where both read and write speeds are crucial, making it an excellent choice for real-time data processing and interactive workloads in Hadoop. It outperforms many other compression codecs in speed, particularly in decompression, making it highly suitable for systems that require rapid access to data. Further , it is splittable and with an addition of an index file, large LZO-compressed files can be processed in parallel by multiple Mappers in a MapReduce job.

3. Deflate (DeflateCodec)

Deflate is a widely used compression codec implemented in many software applications, including Hadoop. It forms the basis for many other file formats such as ZIP and GZIP, but is distinct in its combination of the LZ77 algorithm and Huffman coding. In Hadoop, the Deflate codec is often used because it offers a good balance between compression ratio and speed, and is very effective for compressing text data. It's not as fast as lightweight algorithms like Snappy or LZO, but typically offers faster compression and decompression speeds than more intensive codecs like BZip2.

Like GZIP, files compressed using Deflate are not splittable when stored as a single stream. This is a significant limitation with Deflate but Due to its use in popular formats like ZIP and GZIP, Deflate is widely supported across various platforms and languages, which ensures good interoperability when exchanging data between different systems.



Thanks