Midterm Notes

Big Data Analytics - Midterm Exam Review

1. HDFS & Hadoop Cluster

HDFS Components

Hadoop Distributed File System (HDFS) is designed to store large amounts of data across multiple nodes in a distributed environment.

- **Data Node**: Stores actual file data in blocks.
    - Example: A 1GB file is split into 128MB blocks, each stored on different Data Nodes.
- **Name Node**: Manages metadata such as file names, locations, and block mappings.
    - Acts as the master for file system operations.
- **Replication**: Ensures fault tolerance by replicating data blocks across Data Nodes.
    - Default replication factor is **3**, meaning each block is stored on three different nodes.

Hadoop Cluster Architecture

A Hadoop cluster consists of different types of nodes:

- **Master Node**:
    - Runs the **Name Node**, which manages HDFS.
    - Runs the **Job Tracker**, which schedules jobs.
- **Slave Nodes**:
    - Run **Data Nodes**, which store data blocks.
    - Run **Task Trackers**, which execute MapReduce tasks.
- **Job Tracker**:
    - Assigns tasks to Task Trackers on Slave Nodes.
- **Task Tracker**:
    - Executes assigned tasks and reports status to the Job Tracker.

Storing and Retrieving Files

HDFS efficiently handles large files by splitting them into blocks.

1. **File Upload**:
    - A file is divided into **blocks** (e.g., 128MB each).
    - Blocks are stored on **multiple Data Nodes** for redundancy.
2. **Read Operation**:
    - The client requests metadata from the **Name Node**.
    - The client retrieves actual data directly from the **Data Nodes**.

Running a Job on a Hadoop Cluster

The process of executing a MapReduce job in Hadoop follows these steps:

1. The **client submits** a job to the **Job Tracker**.

2. The **Job Tracker** splits the job into **Map tasks** and **Reduce tasks**.
3. **Task Trackers** execute tasks where data is stored, reducing network latency.
4. The **Job Tracker** monitors task completion and manages failures.
5. The **final output** is written back to **HDFS**.

---

2. MapReduce

MapReduce Workflow

MapReduce processes data using the following steps:

1. **Input Splitting**: The input file is divided into smaller chunks.
2. **Mapping**: Each Mapper processes its chunk and emits key-value pairs.
3. **Shuffling & Sorting**: The framework sorts intermediate data and assigns it to Reducers.
4. **Reducing**: Reducers aggregate results based on keys.
5. **Final Output**: The output is stored back in HDFS.

Key Components

- **Mapper**:
    - Processes input data line by line.
    - Emits intermediate key-value pairs.
    - **Example:** Word count

    ```
    public class WordCountMapper 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);
            }
        }
    }
    ```

- **Combiner** (Optional):
    - Acts as a mini-reducer before data is sent to the final Reducer.
    - Helps in reducing data transfer between Mapper and Reducer.
- **Partitioner**:
    - Determines which Reducer receives a given key.
    - **Example:** Assigning logs by month to different reducers.
- **Shuffle & Sort**:
    - Intermediate key-value pairs are grouped by key before reaching the Reducer.
- **Reducer**:
    - Aggregates values associated with each key.
    - **Example:** Summing word counts

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, Int
erruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Writing MapReduce Programs in Java

1. **Write a Mapper class:** `map()` method processes input and emits key-value pairs.
2. **Write a Reducer class**: `reduce()` method aggregates results based on keys.
3. **Write a Driver class**:
    - Sets the Mapper, Reducer, and Partitioner.
    - Specifies input and output paths.
    - Submits the job to Hadoop.

MapReduce Flow & Combiners

- **Combiner:** Acts as a mini-reducer, performing local aggregation before sending data to the reducer.
    - Reduces data transfer between mapper and reducer.
    - Example: Word Count
        - Without Combiner: Each word count is sent individually.
        - With Combiner: Local aggregation occurs before sending to reducer.
- **Commutative & Associative Operations**
    - Commutative: Order of operation does not affect result (e.g., `a + b = b + a`).
    - Associative: Grouping of operation does not affect result (e.g., `(a + b) + c = a + (b + c)`).
    - **Examples:**
        - **Max function**: Can be used in a combiner (`max(max(A, B), max(C, D))`).
        - **Sum function**: Can also be used.
        - **Average function**: **Cannot** use a combiner directly because averaging partial values leads to incorrect results.

Partitioners

- **Role of Partitioner:**
    - Determines which reducer receives a key.
    - **Default:** HashPartitioner distributes keys evenly.
    - **Custom Partitioner:** Used when specific grouping is needed (e.g., logs by month).
- **Examples of Partitioner Use Cases:**
    - Word count with multiple reducers.
    - Retail sales report with **12 reducers** (one per month).
    - Web traffic by day of the week with **7 reducers**.

- **Implementing a Custom Partitioner:**
  - Extend `Partitioner<K, V>`.
  - Override `getPartition()` method.
  - Example:

    ```
    public int getPartition(K key, V value, int numReduceTasks) {
        return Math.abs(key.hashCode() % numReduceTasks);
    }
    ```

  - Can implement `Configurable` to set up variables before partitioning.

Reducers

- **Single Reducer:**
  - Advantage: Completely sorted output.
  - Disadvantage: Slow if large data.
- **Multiple Reducers:**
  - Example: Assign logs by month to 12 reducers.

---

3. Common MapReduce Algorithms

MapReduce is widely used for processing large-scale data in a distributed manner. Below are some common algorithms implemented using MapReduce, along with detailed explanations and examples.

Sorting

Sorting is one of the most fundamental MapReduce operations. It is useful when dealing with large datasets where sorting helps in efficient querying and retrieval.

Example: Sorting Employee Salaries

Input:

```
Alice, 60000
Bob, 50000
Charlie, 70000
David, 60000
```

Map Phase:

Each key-value pair is transformed to make salary the key so that sorting can be performed.

```
(60000, Alice)
(50000, Bob)
(70000, Charlie)
(60000, David)
```

Shuffle & Sort Phase:

MapReduce automatically sorts the intermediate key-value pairs.

Reduce Phase:

The reducer outputs the sorted data in ascending order.

```
(50000, Bob)
(60000, Alice)
(60000, David)
(70000, Charlie)
```

Sorting can also be used in conjunction with secondary sorting when sorting within partitions is needed.

Searching

Searching allows us to scan large text datasets and find relevant information efficiently.

Example: Finding a Keyword in Log Files

Input: Log file contents

```
log1.txt: Error at line 45
log1.txt: Warning at line 100
log2.txt: Error at line 200
```

Search pattern: `"Error"`

Map Phase:

Each mapper scans a line and emits results if the search pattern is found.

```
(Error, log1.txt:45)
(Error, log2.txt:200)
```

Reduce Phase:

The reducer aggregates the results and outputs matching occurrences.

```
Error found in:
- log1.txt: Line 45
- log2.txt: Line 200
```

This approach is useful for searching error messages in logs, detecting spam in emails, or retrieving documents matching a keyword.

Building an Inverted Index

An inverted index is a key data structure used in search engines to quickly find documents containing a given word.

Example: Creating an Inverted Index for Documents

Input:

```
doc1.txt: Hadoop is a big data framework
doc2.txt: Hadoop uses MapRedu
```

Map Phase:

The mapper emits word-document pairs.

```
(Hadoop, doc1.txt)
(Hadoop, doc2.txt)
(big, doc1.txt)
```

```
(data, doc1.txt)
(framework, doc1.txt)
(uses, doc2.txt)
(MapReduce, doc2.txt)
```

Reduce Phase:

The reducer groups words and outputs a list of documents where each word appears.

```
Hadoop → [doc1.txt, doc2.txt]
big → [doc1.txt]
data → [doc1.txt]
framework → [doc1.txt]
uses → [doc2.txt]
MapReduce → [doc2.txt]
```

This structure helps search engines like Google retrieve relevant documents based on keyword searches.

Word Co-Occurrence

Word co-occurrence measures how often two words appear together in a dataset. This is useful for NLP applications, topic modeling, and recommendation systems.

Example: Measuring Co-Occurrence in a Sentence

Input:

```
doc1.txt: data science is fun
doc2.txt: big data is useful
```

Map Phase:

The mapper emits word pairs with a count of 1.

```
(data, science) → 1
(science, is) → 1
(is, fun) → 1
(big, data) → 1
(data, is) → 1
(is, useful) → 1
```

Reduce Phase:

The reducer sums up the occurrences.

```
(data, science) → 1
(science, is) → 1
(is, fun) → 1
(big, data) → 1
(data, is) → 1
(is, useful) → 1
```

This helps in building word association models, such as for predicting the next word in a sentence.

TF-IDF Computation

TF-IDF (Term Frequency-Inverse Document Frequency) is a key algorithm in text mining and information retrieval. It measures the importance of a word in a document relative to a collection of documents.

Formula:

- **Term Frequency (TF):** TF=Total words in the documentNumber of times term appears in a document

  TF=Number of times term appears in a documentTotal words in the document

- **Inverse Document Frequency (IDF):** IDF=log(Number of documents containing the termTotal number of documents)

  IDF=log(Total number of documentsNumber of documents containing the term)

- **TF-IDF Score:** TF-IDF=TF×IDF

  TF-IDF=TF×IDF

Example: Computing TF-IDF for "Hadoop"

Documents:

```
doc1.txt: Hadoop is a big data tool
doc2.txt: Hadoop and MapReduce work together
doc3.txt: Big data uses MapReduce
```

Step 1: Compute TF

```
TF(Hadoop, doc1) = 1/6
TF(Hadoop, doc2) = 1/5
TF(Hadoop, doc3) = 0
```

Step 2: Compute IDF

```
IDF(Hadoop) = log(3 / 2) = 0.176
```

Step 3: Compute TF-IDF

```
TF-IDF(Hadoop, doc1) = 1/6 * 0.176
TF-IDF(Hadoop, doc2) = 1/5 * 0.176
TF-IDF(Hadoop, doc3) = 0
```

The words with the highest TF-IDF values are considered more important for identifying the topic of each document.

Secondary Sort

Secondary sorting allows sorting within a key's partition, often using a composite key.

Example: Sorting People by Last Name and Birth Year

Input:

```
John Smith, 1985
Alice Brown, 1990
Bob Smith, 1983
Charlie Brown, 1988
```

Map Phase:

The mapper outputs composite keys for sorting.

```
(Smith, 1985) → John
(Smith, 1983) → Bob
(Brown, 1990) → Alice
(Brown, 1988) → Charli
```

Partitioning and Sorting:

- **Partitioner** ensures all people with the same last name go to the same reducer.
- **Sort Comparator** sorts by birth year.

Reduce Phase:

The reducer receives sorted values within each partition.

```
Brown:
  - Charlie (1988)
  - Alice (1990)
Smith:
  - Bob (1983)
  - John (1985)
```

This approach is useful when sorting within a category, such as sorting transactions within a user's account.

These MapReduce algorithms form the backbone of big data processing in distributed systems, enabling tasks such as sorting, searching, indexing, and text analysis at scale.