MapReduce Keerthana Notes

Important Hadoop Concepts

- **Hadoop Framework**: Primarily written in Java, but it can also be developed using other languages such as Python or C++.
- **Hadoop Streaming**: This feature allows developers to write MapReduce programs in various languages like Python, C++, and Ruby. It supports any language that can read from standard input and write to standard output.

Introduction to MapReduce

MapReduce is a programming model used for processing large datasets across distributed systems, such as Hadoop clusters. The process consists of two main stages:

1. **Mapper**: Processes input data and emits key-value pairs. For instance, each word in a sentence serves as a key, with the value set to 1 (indicating its occurrence).
2. **Reducer**: Aggregates data emitted by the mapper by summing values for each unique key, resulting in a count for each word.

Hadoop Streaming API

In the Hadoop Streaming API:

- Mappers read data line by line from standard input (stdin).
- Reducers read from the mappers' output, which is sorted by keys (words) before being passed to the reducer.
- Both mappers and reducers write their output to standard output (stdout).

Step-by-Step Guide to Word Count Program

Step 1: Create Input Data File

Create a file named `word_count_data.txt` and add sample data.

```
cd Documents/ # to change the directory to /Documents
touch word_count_data.txt  # touch is used to create an emp
```

```
ty file
nano word_count_data.txt # nano is a command line editor to
edit the file
cat word_count_data.txt # cat is used to see the content of
the file
```

Step 2: Create Mapper Script

Create a `mapper.py` file that implements the mapper logic. This script will read from STDIN, split lines into words, and output each word with its count.

```
cd Documents/
touch mapper.py
cat mapper.py
```

Mapper Code Example:

```
#!/usr/bin/env python3
import sys

# Mapper function
def map_function(sentence):
    words = sentence.strip().split()
    for word in words:
        print(f"{word}\\t1")  # Output each word with a cou
nt of 1

for line in sys.stdin:
    map_function(line)


#ALTERNATIVELY USE THIS CODE
#!/usr/bin/env python

# import sys because we need to read and write data to STDI
N and STDOUT
import sys
```

```
# reading entire line from STDIN (standard input)
for line in sys.stdin:
    # to remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()

    # we are looping over the words array and printing the
word
    # with the count of 1 to the STDOUT
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        print '%s\t%s' % (word, 1)
```

▼ Shebang: #!/usr/bin/env python3 lets the system know to use Python 3 to run the script.

▼ import sys: Imports the sys module, which provides access to system-level input/output.

💡

- **sentence.strip().split(): strip() removes any leading or trailing whitespace, and split() splits the sentence by spaces into a list of words.**
- **Looping through words: For each word, print(f"{word}\t1") writes a line to standard output, where word is followed by a tab (\t) and the number 1.**
- **This format (word \t 1) is essential for Hadoop Streaming to interpret the output correctly.**
- **Reading Input Line-by-Line: At the bottom of the script, for line in sys.stdin: reads each line from standard input.**
- **This is a Hadoop Streaming requirement, as it streams data through standard input/output.**
  Example Output from mapper.py: If the input is "This is a sentence", the mapper will output:
  This 1
  is 1
  a 1
  sentence 1

💡 Here in the above program #! is known as shebang and used for interpreting the script. The file will be run using the command we are specifying.

Note: test our mapper.py locally that it is working fine or not.

💡 **Syntax**:
cat <text_data_file> | python <mapper_code_python_file>
e.g: cat word_count_data.txt | python
mapper.py

Step 3: Create Reducer Script

Create a `reducer.py` file that implements the reducer logic. This script will read the output of `mapper.py` from STDIN and aggregate occurrences of each word and will write the final output to STDOUT. .

```
cd Documents/
touch reducer.py
cat reducer.py
```

Reducer Code Example:

```
#!/usr/bin/env python3
import sys

# Reducer function to sum up values
current_word = None
current_count = 0

# Read input line by line from standard input
for line in sys.stdin:
    word, count = line.strip().split("\\t")
    count = int(count)

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

```
# Output the last word count if needed
if current_word:
    print(f"{current_word}\\t{current_count}")
#The reducer.py script takes the output from the mapper (so
rted by keys) and aggregates counts for each word.
```

💡 **Initializing Variables:**

- **current_word: Tracks the word being processed.**
  current_count: Accumulates the count for each word.
  Reading Mapper Output Line-by-Line:
- **For each line, word, count = line.strip().split("\t") splits the line by a tab into the word and its count.**
  count = int(count) converts the count to an integer for summation.
  Aggregating Word Counts:
- **If word matches current_word, it means the mapper has emitted multiple entries for the same word, so current_count += count increments the total count.**
- **If word is different from current_word, it means the reducer has finished counting occurrences of the previous word. It prints the total for current_word and resets current_word and current_count to the new word and its count.**
- **Final Output:**
- **After the loop, if current_word: handles the last word, as it may not have been printed in the loop.**
- **Example Output from reducer.py: If the mapper's output for "This is a sentence" and "This is another sentence" was:**
  This 1
  This 1
  is 1
  is 1
  a 1
  another 1
  sentence 1
  sentence 1
  The reducer output would be:
  This 2
  is 2
  a 1
  another 1
  sentence 2

💡 **Note: let's check our reducer code reducer.py with mapper.py is it working properly or not with the help of the below command.**
cat word_count_data.txt | python mapper.py | sort -k1,1 | python reducer.py

💡

Step 4: Run Hadoop Streaming Job - Now let's start all our Hadoop daemons

1. **Start Hadoop Daemons**: Ensure all necessary Hadoop services are running.

```
start-all.sh
```

2. **Create HDFS Directory**: make a directory word_count_in_python in our HDFS in the root directory that will store our word_count_data.txt file with the below command.

```
hdfs dfs -mkdir /word_count_in_python
```

3. **Copy Input File to HDFS** :

```
hdfs dfs -copyFromLocal /path/to/local/word_count_data.t
xt /word_count_in_python/
#hdfs dfs -copyFromLocal /path 1 /path 2 .... /path n /d
estination
*/
Now our data file has been sent to HDFS successfully. we
can check whether it sends or not by using the below com
mand or by manually visiting our HDFS.
 */
```

```
hdfs dfs -ls /       # list down content of the root dir
ectory

hdfs dfs -ls /word_count_in_python    # list down conten
t of /word_count_in_python directory
```

Copy word_count_data.txt to this folder in our HDFS with help of copyFromLocal command.

1. **Make Scripts Executable**: give executable permission to our mapper.py and reducer.py with the help of below command.

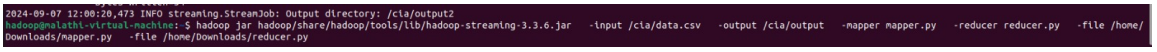
```
cd Documents/
chmod +x mapper.py reducer.py
# OR
cd Documents/
chmod 777 mapper.py reducer.py     # changing the permis
sion to read, write, execute for user, group and others
# OR
chmod +x mapper.py
chmod +x reducer.py
```

2. **Run the Hadoop Streaming Job**:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-st
reaming-*.jar \\
    -input /word_count_in_python/word_count_data.txt \\
    -output /word_count_in_python/output \\
    -mapper /path/to/mapper.py \\
    -reducer /path/to/reducer.py
e.g
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-st
reaming-*.jar \
-input /path/to/input.txt \ (/word_count_in_python/word_
count_data.txt)
-output /path/to/output \ (/word_count_in_python/output)
-mapper /path/to/mapper.py \ (/home/dikshant/Documents/m
apper.py)
-reducer /path/to/reducer.py (/home/dikshant/Documents/r
```

```
educer.py)


I should be giving
 hadoop jar hadoop_user/share/hadoop/tools/lib/hadoop-st
reaming-3.3.6.jar \\
    -input /word_count/data.csv \\
    -output /word_count/op \\
    -mapper mapper.py \\
    -reducer reducer.py
```



3. **Check Output**: In the above command in -output, we will specify the location in HDFS where we want our output to be stored. So let's check our output in output file at location /word_count_in_python/output/part-00000. We can check results by manually vising the location in HDFS or with the help of cat command as shown below

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

Understanding WordCount Example

- The program reads a text file and outputs a text file.
- Ensure the Hadoop cluster is up and running.
- Understand the differences between single-node and multi-node clusters.
- Data formats supported by Hadoop should be considered before running jobs.
- Before we run the actual MapReduce job, we first have to copy the files from our local file system to Hadoop's HDFS.
- Hadoop Streaming facilitates data passing between Map and Reduce code via STDIN and STDOUT.

https://www.geeksforgeeks.org/hadoop-streaming-using-python-word-count-problem/

https://www.geeksforgeeks.org/hadoop-streaming-using-python-word-count-problem/

EXAMPLE PRPGRAM:

Initial Hadoop Setup and Starting Services

1. **Start Hadoop Services**:
   - **Command**: `start-all.sh`
   - **Purpose**: This command starts all essential Hadoop daemons, including:
     - **Namenode**: Manages the file system namespace and regulates access to files by clients.
     - **Datanode(s)**: Store actual data and handle read/write requests.
     - **Secondary Namenode**: Assists in housekeeping tasks and checkpoints the filesystem metadata.
     - **Resource Manager**: Manages the allocation of compute resources.
     - **Node Manager(s)**: Manages containers on each node, monitoring their resource usage.
2. **Verify Daemons**:
   - **Command**: `jps`
   - **Purpose**: This command lists all Java processes, allowing you to confirm that each Hadoop daemon is running.
3. **Check Network Configuration**:
   - **Install `net-tools` (if not already installed)**:
     - **Command**: `sudo apt install net-tools`
     - This installs tools like `ifconfig`, which can check network interfaces.
   - **Edit sudoers file (optional)**:

- **Commands**:
  Add
  `hadoop ALL=(ALL) ALL` if the Hadoop user lacks required permissions.

  ```
  su
  visudo
  ```

4. **Get the Inet Address**:
   - **Command**: `ifconfig`
   - **Purpose**: Displays network interfaces, including the **inet address** (IP address) which can be used to access the Hadoop Namenode web interface at `<inet_address>:9870`.

Setting Up and Navigating Directories for Hadoop Programs

1. **Save Program in the Hadoop Directory**:
   - Save MapReduce or other Hadoop programs within the Hadoop directory. This can be done through:
     - **GUI**: Directly moving files in the GUI.
     - **Commands**: For example, use `scp` or `mv` commands to move files.
2. **Understanding Directory Navigation**:
   - When you open a terminal, you may be in the **Hadoop user's home directory**, not the **root directory**.
   - **Command**: `cd ..`
     - This moves you up one level in the directory tree.
     - Example: `hadoop@LAPTOP-JHB2JBGQ:/home$` indicates that you're in the home directory, and moving up takes you to `/`.

Working with HDFS (Hadoop Distributed File System)

1. **Check Directory Structure**:
   - **Command**: `ls`
   - This lists files and folders in the current directory.
2. **Create HDFS Directory**:

- **Command**: `hdfs dfs -mkdir /user`
- Purpose: Creates a `/user` directory within HDFS, which is a common starting directory for storing data files.
- **Common Issue**: `hdfs dfs -mkdir user` (without a leading `/`) is incorrect, as HDFS paths should start from the root `/`.

3. **Listing Users in HDFS**:
   - **Explanation**: Running `hdfs dfs -ls /user` will display directories (e.g., `hadoop`, `keerthana`, `keerthana23`) that represent user directories within HDFS. These are separate from the Linux system users; they're HDFS directories created within the `/user` path.
4. **Transferring Files to HDFS**:
   - **Path of the Default User**:
     - You can find the default user's home directory path with `echo $HOME` or `pwd` if you're in the default user's directory.
   - **Transfer File**:
     - **Command**: `hdfs dfs -put /path/to/data.csv /user/hadoop/`
     - Purpose: Uploads `data.csv` to the `/user/hadoop` directory in HDFS.
5. **Displaying Contents of a File in HDFS**:
   - **Command**: `hdfs dfs -cat /user/hadoop/data.csv`
   - **Purpose**: Outputs the contents of `data.csv` to the console, allowing you to confirm that the file has been uploaded correctly.

Example Workflow for Running a Hadoop MapReduce Program

Here's a complete workflow for a MapReduce job using the steps outlined:

1. **Start Hadoop Services**:

```
start-all.sh
```

2. **Check Daemons**:

```
jps
```

3. **Get Inet Address**:

   ```
   ifconfig
   ```

   Use `<inet_address>:9870` to access the Namenode UI.

4. **Set Up Directory in HDFS**:

   ```
   hdfs dfs -mkdir /user/hadoop
   ```

5. **Transfer Data to HDFS**:

   ```
   hdfs dfs -put /path/to/data.csv /user/hadoop/
   ```

6. **Run MapReduce Program**:

   ```
   hadoop jar /path/to/hadoop-streaming-*.jar -input /user/
   hadoop/data.csv -output /user/hadoop/output -mapper /pat
   h/to/mapper.py -reducer /path/to/reducer.py
   ```

7. **Check Output**:

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

This setup will allow you to run and verify Hadoop MapReduce jobs with custom data files and Python scripts as mapper and reducer.

ADDITIONAL

Yes, you can use CSV data with the same Hadoop Streaming code, but there are a few adjustments to keep in mind. Since your existing code processes text data by splitting lines into words, you'll need to handle the CSV format appropriately in the `mapper.py` file.

Adjusting the Mapper Code for CSV Data

In CSV files, data is typically structured in columns, so you may need to modify your mapper to parse CSV lines correctly and extract relevant information. For example, if you want to count word occurrences across different fields in the CSV, you'll need to:

1. **Read the CSV line by line**.
2. **Split each line by the CSV delimiter** (often a comma `,`).
3. **Extract the desired columns** or split each column's text into words.
4. **Emit each word with a count of** `1`, just like in your existing code.

Modified Example Code for `mapper.py`

Here's how you can adjust the `mapper.py` code to handle CSV data:

```
import sys
import csv

def map_function(line):
    words = line.split()  # Modify this line if specific co
lumns are needed
    mapped_values = []
    for word in words:
        mapped_values.append((word, 1))
    return mapped_values

# Read each line from standard input
for line in sys.stdin:
    # Parse CSV row
    row = line.strip().split(',')

    # Assuming you want to process all columns; adjust as n
eeded
    for field in row:
        words = field.split()  # Split each field into word
s
        for word in words:
            # Print each word with a count of 1
            print(f"{word}\\t1")
```

Notes on Modifications

1. **CSV Parsing**:

- If your data has commas within fields (like `"Smith, John"`), consider using the `csv.reader` to handle parsing correctly:

```
reader = csv.reader(sys.stdin)
for row in reader:
    for field in row:
        words = field.split()
        for word in words:
            print(f"{word}\\t1")
```

2. **Target Specific Columns**:
   - If you only need specific columns (e.g., `row[0]` for the first column), select them accordingly to reduce processing time.

Running the Code with Hadoop Streaming

You can execute this MapReduce job in Hadoop Streaming as usual. Ensure the CSV file is stored in HDFS, then run the Hadoop Streaming command:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-strea
ming-*.jar \\
  -input /user/hadoop/data.csv \\
  -output /user/hadoop/output \\
  -mapper /path/to/mapper.py \\
  -reducer /path/to/reducer.py
```

Reducer Code (No Change Needed)

The `reducer.py` code remains the same since it only needs to sum up counts for each word, which is independent of the input data format.

By making these minor adjustments to your mapper code, you can seamlessly process CSV data in Hadoop MapReduce!

The CSV parser should be used in your `mapper.py` script where the data is read line by line. Using a CSV parser, such as Python's built-in `csv` module, ensures that each line in the CSV file is correctly split into fields, even if the fields contain commas within quotes. This is especially helpful for more complex CSV

files, as simply splitting by commas could lead to incorrect parsing if there are commas within the field values.

Example of Using `csv.reader` in Mapper

Here's how to incorporate `csv.reader` in your `mapper.py` file to handle CSV input:

```
import sys
import csv

# Use csv.reader to parse each line of input as CSV
reader = csv.reader(sys.stdin)

for row in reader:
    # Process each row of CSV data; `row` is a list of fiel
ds in the row
    for field in row:
        words = field.split()  # Split each field into word
s
        for word in words:
            # Emit each word with a count of 1
            print(f"{word}\\t1")
```

Explanation of the Code

1. `csv.reader(sys.stdin)`: Reads each line of input from standard input (usually provided by Hadoop Streaming) and parses it as CSV. Each line is split into a list of fields.
2. **Iterate Over Rows and Fields**: For each `row`, `row` is a list of columns in that row. We then iterate through each `field` in the row.
3. **Split and Emit Words**: For each field, split it into words and emit each word with a count of `1`.

Why Use `csv.reader`

- **Handles Quoted Fields**: `csv.reader` correctly handles commas within quoted fields (e.g., `"Smith, John"`).
- **Flexible Delimiters**: While CSV typically uses commas, `csv.reader` can be configured to use other delimiters if necessary.

Example Input and Output

Consider the following CSV data in `data.csv`:

```
Name,Occupation
"Smith, John",Engineer
Jane,Dentist
```

With the modified `mapper.py` using `csv.reader`, the mapper would output:

```
Smith   1
John    1
Engineer 1
Jane    1
Dentist 1
```

This setup ensures that the data is processed correctly, without misinterpreting commas within fields as separators.

How to Run in Hadoop Streaming

Place `mapper.py` and `reducer.py` in your working directory. Then, execute the following command:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-strea
ming-*.jar \\
  -input /user/hadoop/data.csv \\
  -output /user/hadoop/output \\
  -mapper /path/to/mapper.py \\
  -reducer /path/to/reducer.py

 hadoop jar hadoop/share/hadoop/tools/lib/hadoop-streaming-
3.3.6.jar -input /user/data.csv -output /word_count/op -map
per mapper.py -reducer reducer.py -file /mnt/emapper.py -fi
le /mnt/e/reducer.py
```

With this setup, the `mapper.py` script will process each line of CSV data correctly, and the `reducer.py` script will aggregate word counts as usual.

Move command]

```
mv file1.txt file2.txt file3.txt /path/to/destination/
```

Copy

```
cp file.txt /path/to/destination/
```

Allow to edit in hdfs

```
hdfs dfs -chmod 777 /word_count
```