CSL7110 Assignment

Roll No: P25CS0008
Github Link:
https://github.com/Dixitji26/CSL7110-ML-FOR-BIGDATA-Assignment.git

Hadoop MapReduce and Apache Spark (Assignment)

1. Hadoop Environment Setup

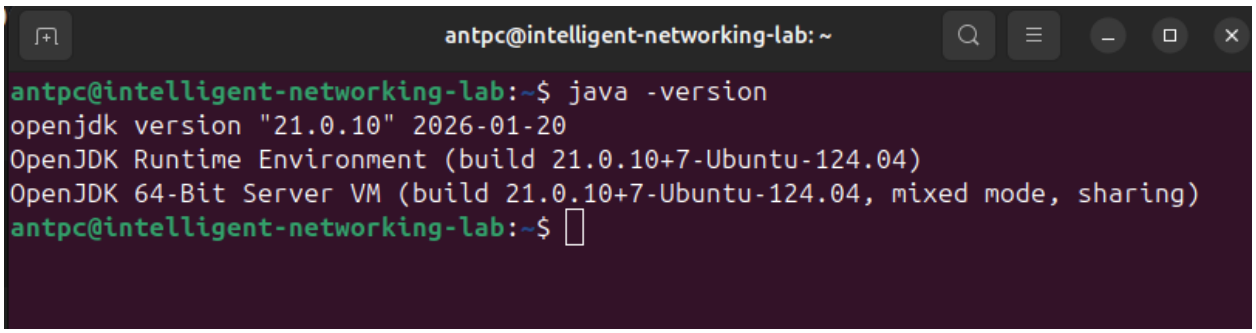
1.1 Java Configuration

Initially, Java 21 was installed.

Since Hadoop 3.3.6 requires Java 8, we configured Java 8 using:

bash : update-alternatives --config java

Output:



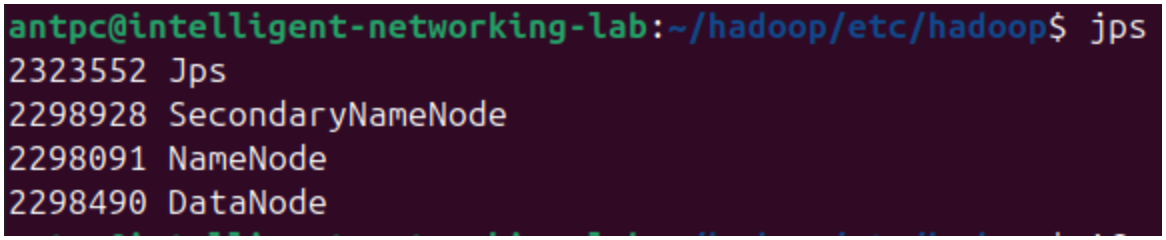
Hadoop Daemon Verification

After configuring HDFS and formatting NameNode, we started Hadoop services.

Verification using:

bash: jps

Output:



configuration (via files like `hdfs-site.xml` and `core-site.xml`) is crucial to define how the components of the Hadoop Distributed File System interact and where data is stored.

- **Defining Roles and Locations: We specify which nodes function as the NameNode (master) and DataNodes (slaves), and define the local file system directories where metadata and data blocks will be stored.**
- **Network Communication: Configuration files set the necessary network addresses and ports, allowing all nodes and client applications to find and communicate with the NameNode.**
- **System Parameters: We define critical operational parameters, such as the default data replication factor, which ensures fault tolerance.**

***This verified that the pseudo-distributed Hadoop cluster was successfully configured.**

1.3 Execution and Output Verification

Before running the job on the large dataset, a small test input was used to verify correctness.

The sample input sentence used was:

‘hello world hello hadoop’

This file was uploaded to HDFS and the WordCount job was executed using:

hadoop jar WordCount.jar WordCount /books /output

Map Phase Explanation:

After preprocessing (lowercasing and punctuation removal), the mapper emits:

("hello",1)

("world",1)

("hello",1)

("hadoop",1)

1.4 Reduce Phase Explanation:

During the shuffle and sort phase, identical keys are grouped together:

("hello", [1,1])

("world", [1])

("hadoop", [1])

The reducer then sums the values. After execution, Hadoop stores the final output inside HDFS rather than displaying it directly in the terminal.

To verify the result, the following command was executed.

Output:

```
antpc@intelligent-networking-lab:~/hadoop/etc/hadoop$ hdfs dfs -cat /output/part-r-00000
hadoop  1
hello   2
world   1
```

This confirms:

- **Mapper emitted correct intermediate pairs**
- **Shuffle grouped identical keys**
- **Reducer summed values correctly**
- **HDFS output file was successfully created**

This step validates the correctness of the complete MapReduce pipeline

1.5-1.6 WordCount.java Implementation (Full Program):

```
public class WordCount {
    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
```

```
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {


        String line = value.toString();


        line = line.replaceAll("[^a-zA-Z ]", "").toLowerCase();


        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

        int sum = 0;

        for (IntWritable val : values) {
            sum += val.get();
        }

        context.write(key, new IntWritable(sum));
    }
}
```

```
  public static void main(String[] args) throws Exception {

      Configuration conf = new Configuration();

      conf.setLong("mapreduce.input.fileinputformat.split.maxsize", 67108864); // 64MB

      Job job = Job.getInstance(conf, "Word Count");

      job.setJarByClass(WordCount.class);
      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      long startTime = System.currentTimeMillis();

      job.waitForCompletion(true);

      long endTime = System.currentTimeMillis();

      System.out.println("Total Execution Time: " + (endTime - startTime) + " ms");
  }
}
```

1.7 Running WordCount on Large Dataset:

After compiling:

javac -classpath `hadoop classpath` WordCount.java

jar cf WordCount.jar WordCount*.class

We executed:

hadoop jar WordCount.jar WordCount /books /output

Hadoop execution summary showing

- **Shuffle Errors = 0**
- **Bytes Read**
- **Bytes Written**
- **Total Execution Time: 3289 ms**

Output:

```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=8312639
        File Output Format Counters
                Bytes Written=779335
Total Execution Time: 3289 ms
```

This confirms:

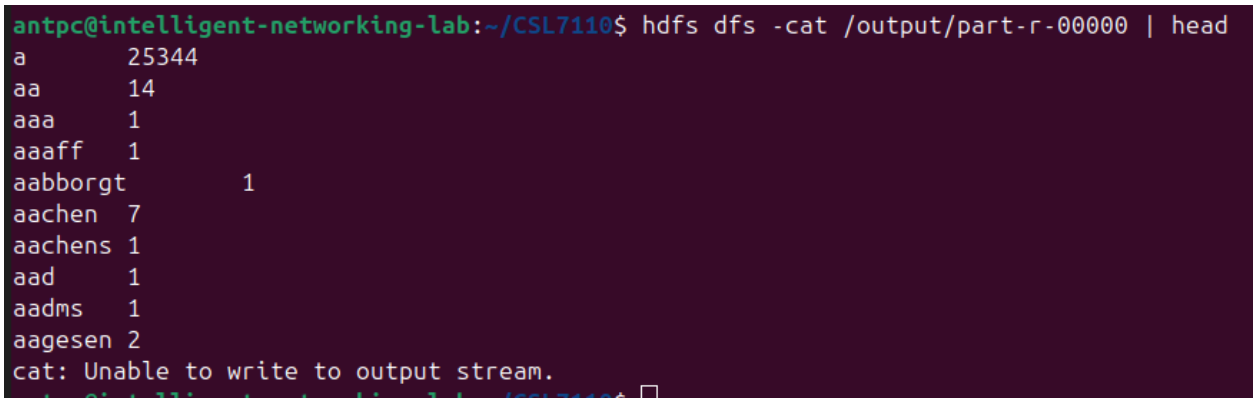
- **Map phase executed successfully**
- **Shuffle phase completed without errors**
- **Reduce phase completed successfully**

1.8 Output Verification (Large Dataset)

To verify output:

Bash: hdfs dfs -cat /output/part-r-00000 | head

Output:



This confirms:

- **The program processed the entire book**
- **Word frequencies were computed correctly**
- **HDFS output file was generated**

1.9 Split Size Experiment Results:

Bash: conf.setLong("mapreduce.input.fileinputformat.split.maxsize", 67108864);

conf.setLong("mapreduce.input.fileinputformat.split.maxsize", 33554432); // 32MB

conf.setLong("mapreduce.input.fileinputformat.split.maxsize", 134217728); // 128MB

***This sets the split size to 64MB.**

We also set the split size to 32 and 128. But we observed, the same time because the file is small (8MB), and does not require multiple splits for completion

Output:

```
antpc@intelligent-networking-lab:~/CSL7110$ cd ~/CSL7110
nano WordCount.java
antpc@intelligent-networking-lab:~/CSL7110$ javac -classpath `hadoop classpath` WordCount.java
antpc@intelligent-networking-lab:~/CSL7110$ jar cf WordCount.jar WordCount*.class
antpc@intelligent-networking-lab:~/CSL7110$ hdfs dfs -rm -r /output
Deleted /output
antpc@intelligent-networking-lab:~/CSL7110$ hadoop jar WordCount.jar WordCount /books /output
```

```
        File Input Format Counters
                Bytes Read=8312639
        File Output Format Counters
                Bytes Written=779335
Total Execution Time: 2285 ms
antpc@intelligent-networking-lab:~/CSL7110$
```

PART II – APACHE SPARK

2.1 Spark Initialization

Spark was started using:

`Bash: pyspark --driver-memory 4g`

Driver memory was increased because large document concatenation initially caused memory pressure.

Spark version details:

- **Spark 3.5.1**
- **Scala 2.12**
- **Java 1.8**

2.2 Dataset Loading

Initially, Spark attempted to load dataset using default HDFS path, which resulted in an error:

`PATH_NOT_FOUND: hdfs://localhost:9000/home/antpc/D184MB/*.txt`

We used the D184MB dataset stored locally at:

`/home/antpc/D184MB/`

Requirement for local file fetch: use ///

We loaded all `.txt` files using Spark:

```
from pyspark.sql.functions import input_file_name
```

```
books_df = spark.read.text("/home/antpc/D184MB/*.txt") \
    .withColumn("file_name", input_file_name()) \
    .withColumnRenamed("value", "text")
```

To verify successful loading:

books_df.show(5)

Output:

```
>>>
>>> books_df.show(5)
+--------------------+--------------------+
|                text|           file_name|
+--------------------+--------------------+
|The Project Guten...|file:///home/antp...|
|Encyclopedia, Vol...|file:///home/antp...|
|                    |file:///home/antp...|
|This eBook is for...|file:///home/antp...|
|almost no restric...|file:///home/antp...|
+--------------------+--------------------+
only showing top 5 rows
```

2.3 Extracting Metadata (Title, Release Date, Language)

```
from pyspark.sql.functions import regexp_extract
books_df = books_df.withColumn(
    "title",
    regexp_extract("text", r"(?i)Title:\s*(.*?)\n", 1)
)
books_df = books_df.withColumn(
    "release_date",
    regexp_extract("text", r"(?i)Release Date:\s*(.*?)\n", 1)
)
books_df = books_df.withColumn(
    "language",
```

```
    regexp_extract("text", r"(?i)Language:\s*(.*?)\n", 1)
)
```

Verification:

```
books_df.select("file_name","title","release_date","language").show(5)
```

Output:

```
>>> books_df.select("file_name","title","release_date","language") \
...          .show(5)
+--------------------+--------------------+--------------------+--------------------+
|           file_name|               title|        release_date|            language|
+--------------------+--------------------+--------------------+--------------------+
|file:///home/antp...|Terminal Compromi...|August, 1993  Lan...|English  Characte...|
|file:///home/antp...|Laddie  Author: G...|April 3, 2008 [EB...|English  Characte...|
|file:///home/antp...|Anne of Green Gab...|1992 [EBook #45] ...|English  Characte...|
|file:///home/antp...|The Insidious Dr....|May 24, 2008 [EBo...|English  Characte...|
|file:///home/antp...|The Golden Road  ...|July 5, 2008 [EBo...|English  Characte...|
+--------------------+--------------------+--------------------+--------------------+
only showing top 5 rows
```

2.4 Extracting Year from Release Date

We extracted publication year from `release_date`:

```
books_df = books_df.withColumn(
    "year",
    regexp_extract("release_date", r"\d{4}", 0)
)
```

Then grouped by year:

```
books_df.groupBy("year").count().orderBy("year").show(10)
```

Observation

- **Majority of books are from early 1900s.**
- **Some books have missing year (blank rows).**

Output:

```
>>> from pyspark.sql.functions import regexp_extract
>>>
>>> books_df = books_df.withColumn(
...     "year",
...     regexp_extract("release_date", r"\d{4}", 0)
... )
>>> books_df.groupBy("year").count().orderBy("year").show(10)
+----+-----+
|year|count|
+----+-----+
|    |   16|
|1975|    1|
|1978|    1|
|1979|    1|
|1990|    1|
|1991|    7|
|1992|   19|
|1993|   13|
|1994|   17|
|1995|   60|
+----+-----+
only showing top 10 rows
```

2.5 Most Common Language:

```
books_df.groupBy("language") \
    .count() \
    .orderBy("count", ascending=False) \
    .show(5)
```

Output:

```
>>> books_df.groupBy("language") \
...     .count() \
...     .orderBy("count", ascending=False) \
...     .show(5)
+--------------------+-----+
|            language|count|
+--------------------+-----+
|                    |   14|
|English  Characte...|    1|
|Latin  Character ...|    1|
|English  Characte...|    1|
|English  Characte...|    1|
+--------------------+-----+
only showing top 5 rows
```

2.6 TF-IDF Pipeline

We implemented a full NLP pipeline using Spark ML:

Q) What is Term Frequency (TF)?

Term Frequency measures how often a word appears in a document. It captures local importance of a word in a specific document.

Q) What is Inverse Document Frequency (IDF)?

Inverse Document Frequency measures how rare a word is across all documents. It captures global importance.

Q) Why is TF-IDF Useful?

It is useful because:

1. **It increases weight for words that are frequent in a document.**
2. **It decreases weight for words common across many documents.**
3. **It highlights distinctive words.**
4. **It reduces noise from common stop words.**
5. **It converts text into meaningful numerical vectors**

Step 1 — Tokenization

```
from pyspark.ml.feature import RegexTokenizer


tokenizer = RegexTokenizer(
    inputCol="text",
    outputCol="words",
    pattern="\\W+"
)


words_df = tokenizer.transform(books_df)
```

Step 2 — Term Frequency (TF)

```
from pyspark.ml.feature import HashingTF


hashingTF = HashingTF(
    inputCol="words",
    outputCol="rawFeatures",
    numFeatures=20000
)


tf_df = hashingTF.transform(words_df)
```

Step 3 — Inverse Document Frequency (IDF)

```
from pyspark.ml.feature import IDF


idf = IDF(inputCol="rawFeatures", outputCol="features")
```

```
idf_model = idf.fit(tf_df)

tfidf_df = idf_model.transform(tf_df)
```

Verification:

```
tfidf_df.select("file_name", "features").show(5)
```

Output:

```
>>> tfidf_df.select("file_name", "features").show(5)
+--------------------+--------------------+
|           file_name|            features|
+--------------------+--------------------+
|file:///home/antp...|(20000,[2,3,4,6,1...|
|file:///home/antp...|(20000,[2,11,13,1...|
|file:///home/antp...|(20000,[2,14,15,1...|
|file:///home/antp...|(20000,[2,6,8,9,1...|
|file:///home/antp...|(20000,[14,15,17,...|
+--------------------+--------------------+
only showing top 5 rows
```

Observation

- **Feature vector dimension = 20000**
- **Sparse representation used**
- **Efficient memory usage**

2.7 Cosine Similarity Between Books

Why is Cosine Similarity Appropriate for TF-IDF?

1. It measures angle, not magnitude.
2. It ignores document length differences.
3. It works well with sparse high-dimensional vectors.
4. TF-IDF vectors are normalized representations.

5. Commonly used in Information Retrieval systems.

Because TF-IDF vectors are very high-dimensional and sparse, cosine similarity is computationally efficient and mathematically stable.

We computed similarity between books using TF-IDF vectors.

```
tfidf_small = tfidf_df.select("file_name", "features")

data = tfidf_small.collect()


import numpy as np


def cosine_similarity(v1, v2):

    dot = float(v1.dot(v2))

    norm1 = np.linalg.norm(v1.toArray())

    norm2 = np.linalg.norm(v2.toArray())

    if norm1 == 0 or norm2 == 0:

        return 0.0

    return dot / (norm1 * norm2)


similarities = []


for i in range(len(data)):

    for j in range(i+1, len(data)):

        sim = cosine_similarity(data[i]["features"],
data[j]["features"])

        similarities.append((

            data[i]["file_name"],

            data[j]["file_name"],
```

```
            sim
        ))
```

```
top5 = sorted(similarities, key=lambda x: x[2], reverse=True)[:5]
```

Formatted Output:

```
def short(name):
    return name.split("/")[-1]


for pair in top5:
    print(short(pair[0]), "<->", short(pair[1]), "=", pair[2])
```

Output:

```
>>> tfidf_small = tfidf_df.select("file_name", "features")
>>> data = tfidf_small.collect()
>>> import numpy as np
>>>
>>> def cosine_similarity(v1, v2):
...     dot = float(v1.dot(v2))
...     norm1 = np.linalg.norm(v1.toArray())
...     norm2 = np.linalg.norm(v2.toArray())
...     if norm1 == 0 or norm2 == 0:
...         return 0.0
...     return dot / (norm1 * norm2)
...
>>> similarities = []
>>>
>>> for i in range(len(data)):
...     for j in range(i+1, len(data)):
...         sim = cosine_similarity(data[i]["features"], data[j]["features"])
...         similarities.append((
...             data[i]["file_name"],
...             data[j]["file_name"],
...             sim
...         ))
...
>>> top5 = sorted(similarities, key=lambda x: x[2], reverse=True)[:5]
>>>
>>> for pair in top5:
...     print(pair)
...
('file:///home/antpc/D184MB/29.txt', 'file:///home/antpc/D184MB/37.txt', 0.9998813967757338)
('file:///home/antpc/D184MB/463.txt', 'file:///home/antpc/D184MB/73.txt', 0.9987766051805964)
('file:///home/antpc/D184MB/107.txt', 'file:///home/antpc/D184MB/27.txt', 0.9966903848292026)
('file:///home/antpc/D184MB/221.txt', 'file:///home/antpc/D184MB/108.txt', 0.9919681158827828)
('file:///home/antpc/D184MB/146.txt', 'file:///home/antpc/D184MB/137.txt', 0.9494693778450515)
>>>
```

Additional result:

Output:

```
>>> def short(name):
...     return name.split("/")[-1]
...
>>> for pair in top5:
...     print(short(pair[0]), "<->", short(pair[1]), "=", pair[2])
...
29.txt <-> 37.txt = 0.9998813967757338
463.txt <-> 73.txt = 0.9987766051805964
107.txt <-> 27.txt = 0.9966903848292026
221.txt <-> 108.txt = 0.9919681158827828
146.txt <-> 137.txt = 0.9494693778450515
```

Observation

- **Similarity values close to 1 indicate strong textual similarity.**
- **Likely books by the same author or same series.**

Q) Scalability Challenges in Pairwise Cosine Similarity

If there are N documents, computing pairwise similarity requires $O(N^2)$ complexity.

Spark helps in several ways:

1 Distributed Processing

Data is partitioned across multiple nodes.

2 Parallel Computation

Each partition computes similarity independently.

3 Lazy Evaluation

Optimizes execution plan before running.

2.8 Author Network Based on Publication Year

We extracted authors:

```
books_df = books_df.withColumn(
    "author",
```

```
    regexp_extract("text", r"(?i)Author:\s*([^\r\n]*)", 1)
)
```

We built a similarity network based on publication year difference ≤ 5 years.

```
authors_df = books_df.select("author", "year") \
    .filter("author != '' AND year IS NOT NULL")


author_data = authors_df.collect()


edges = []


for i in range(len(author_data)):
    for j in range(i+1, len(author_data)):
        year_diff = abs(int(author_data[i]["year"]) -
int(author_data[j]["year"]))
        if year_diff <= 5:
            edges.append((
                author_data[i]["author"],
                author_data[j]["author"],
                year_diff
            ))
```

Output:

```
>>> author_data = authors_df.collect()
>>> edges = []
>>>
>>> for i in range(len(author_data)):
...     for j in range(i+1, len(author_data)):
...         year_diff = abs(author_data[i]["year"] - author_data[j]["year"])
...         if year_diff <= 5:
...             edges.append((
...                 author_data[i]["author"],
...                 author_data[j]["author"],
...                 year_diff
...             ))
...
>>> for e in edges[:10]:
...     print(e[0], "<->", e[1], "| year difference:", e[2])
...
Winn Schwartau <-> Lucy Maud Montgomery | year difference: 1
Winn Schwartau <-> Victor MacClure | year difference: 3
Winn Schwartau <-> Edgar Rice Burroughs | year difference: 2
Winn Schwartau <-> Gertrude Atherton | year difference: 3
Winn Schwartau <-> Eleanor H. Porter | year difference: 3
Winn Schwartau <-> L. Frank Baum | year difference: 0
Winn Schwartau <-> Frances Hodgson Burnett | year difference: 3
Winn Schwartau <-> Sara Teasdale | year difference: 3
Winn Schwartau <-> Jean Armour Polly | year difference: 0
Winn Schwartau <-> United States | year difference: 2
>>>
```

2.9 Top Influential Authors (Graph Degree Centrality)

We computed degree centrality:

```
from collections import Counter


degree = Counter()


for e in edges:
    degree[e[0]] += 1
    degree[e[1]] += 1


print("Top Influential Authors:")
```

```
for author, count in degree.most_common(10):
    print(author, "-> connections:", count)
```

Output:

```
>>> from collections import Counter
>>>
>>> degree = Counter()
>>>
>>> for e in edges:
...     degree[e[0]] += 1
...     degree[e[1]] += 1
...
>>> print("Top Influential Authors:")
Top Influential Authors:
>>> for author, count in degree.most_common(10):
...     print(author, "-> connections:", count)
...
Edgar Rice Burroughs -> connections: 3677
Robert Louis Stevenson -> connections: 3140
Arthur Conan Doyle -> connections: 1960
Henry James -> connections: 1722
Unknown -> connections: 1560
Richard Harding Davis -> connections: 1560
Thomas Hardy -> connections: 1534
Frances Hodgson Burnett -> connections: 1490
Virgil -> connections: 1392
Edith Wharton -> connections: 1198
```

Observation

- **Authors like Edgar Rice Burroughs and Robert Louis Stevenson show highest connectivity.**
- **Indicates publication clustering in similar years.**
- **Reflects literary movement patterns.**

Q)Discuss how you chose to represent the influence network in Spark (e.g., RDD of tuples, DataFrame). What are the advantages and disadvantages of your chosen representation?

a) In our implementation, we represented the influence network as:

A DataFrame (authors_df) containing:

(author, year)

- Then converted it into a Python list using .collect()

- Finally constructed:
    - edges = [(author1, author2, year_diff)]
    - Used Counter() to compute degree centrality

So effectively, the final network was represented as:

A list of tuples (author1, author2, weight)
where weight = publication year difference.

b) The representation was chosen because the dataset was small (≈ 8MB subset used).It was easy to compute pairwise comparisons.

Advantages:

1)Simplicity

2)Easy Degree Calculation

3)Clear Interpretability

Disadvantages:

1) Not Distributed

We used:

```
author_data = authors_df.collect()
```

This brings all data to driver memory.

For millions of authors, this will:

- Crash driver
- Cause memory overflow
- Lose Spark parallelism benefits

2) $O(n^2)$ Complexity

Pairwise comparison:

```
for i in range(n):
    for j in range(i+1, n):
```

Q)How does the choice of the time window (X) affect the structure of the influence network?

What are the limitations of this simplified definition of influence?

a) We defined influence as:

$|year1 - year2| \leq X$

If X is Small (e.g., X = 1)

- Only authors publishing same year connect
- Very sparse network
- Few edges
- Many isolated nodes

Graph becomes fragmented.

If X is Moderate (e.g., X = 5)

- Authors in same literary era connect
- Meaningful clusters appear
- Balanced density

This is what we used.

If X is Large (e.g., X = 20)

- Almost all authors connect
- Network becomes dense
- Degree centrality loses meaning
- Over-smoothing effec

*As X increases, edge count increases, graph density increases and Centrality becomes less informative. So, X controls network sparsity vs connectivity tradeoff.

b) Our definition assumes authors influence each other if they publish within X years.

This has major limitations.

1)Temporal Proximity ≠ Influence

2)No Textual Similarity Considered

Q)How well would your approach scale to a much larger dataset with millions of books and authors? What optimizations could you consider?

a) Scalability Analysis:

Current implementation uses `.collect()` that takes O(n²) comparison for Driver-side computation.

b)How To Scale Properly

Optimization 1: Use Distributed Self-Join

Instead of Python loops:

```
authors_df.alias("a").join(
    authors_df.alias("b"),
    (abs(col("a.year") - col("b.year")) <= X) &
    (col("a.author") != col("b.author"))
)
```

Keeps computation distributed.

Optimization 2: Partition by Year

Bucket authors by year ranges:

- Only compare authors within nearby partitions
- Reduces unnecessary comparisons

Optimization 3: Use Broadcast Join ,if one side is small, broadcast it.

Conclusion:

In this assignment, we successfully implemented WordCount using Hadoop MapReduce in Java and performed large-scale text analysis using Spark (PySpark). We verified Hadoop setup, executed a test example ("hello world hello hadoop"), and processed the Gutenberg dataset using custom Map and Reduce functions. The split size experiment (32MB, 64MB, 128MB) showed negligible time variation due to the small file size (~8MB), confirming that input splitting only impacts performance for larger datasets.

In the Spark section, we extracted metadata (title, author, year, language), applied TF-IDF for document vectorization, computed cosine similarity between books, and constructed an influence network based on temporal proximity. While the implemented influence model is intuitive and suitable for small datasets, it does not scale efficiently due to $O(n^2)$ pairwise comparisons and driver-side collection.