Java Word Count MapReduce

Give Java mapreduce code for word count

Here's a simple Java MapReduce code for counting words using Hadoop. This example splits each line of input into words and then counts the occurrences of each word across all lines.

WordCount.java

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String[] words = value.toString().split("\\s+");
            for (String w : words) {
                word.set(w);
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable
> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation

1. **Mapper**:
   - **TokenizerMapper** splits each line of the input into words.
   - Each word is outputted as a key, with a value of `1` (indicating one occurrence of the word).
2. **Reducer**:
   - **IntSumReducer** sums up all values for each word key, giving the total count of occurrences for each word.
3. **Driver**:
   - Configures the job, sets the Mapper and Reducer classes, defines input and output formats, and runs the MapReduce job.

Compilation and Execution

1. **Compile**:

   ```bash
   javac -classpath $(hadoop classpath) -d . WordCount.java
   jar -cvf wordcount.jar *.class
   ```

2. **Run the Job**:

   ```bash
   hadoop jar wordcount.jar WordCount /input_path /output_path
   ```

Notes

- Ensure Hadoop is properly configured and paths are set.
- Replace `/input_path` and `/output_path` with your HDFS paths for input and output.

Give Java mapreduce code for searching and sorting

Here's a Java MapReduce code for both **searching** (filtering lines based on a keyword) and **sorting** (sorting results by frequency of occurrence). This example focuses on a specific keyword search and sorting based on count, which is useful when you want to list how frequently different words appear in files that contain a keyword.

Key Steps

1. **Filtering**: Mapper searches for lines containing a specific keyword.
2. **Counting**: The Reducer counts occurrences of each filtered word.
3. **Sorting by Count**: An additional job sorts the counts in descending order.

Code Overview

This code consists of:

1. `` `SearchMapper` ``: Filters lines based on a keyword.
2. `` `CountReducer` ``: Counts occurrences of each word.
3. `` `SortMapper` `` and `` `SortReducer` ``: Sorts the word count results.

Complete Code

java

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.jobcontrol.ControlledJob;
import org.apache.hadoop.mapreduce.lib.jobcontrol.JobControl;

public class SearchAndSort {

    // First Mapper: Search for keyword and emit words in lines containing it
    public static class SearchMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        private String keyword;

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            Configuration conf = context.getConfiguration();
            keyword = conf.get("keyword");  // Get keyword from configuration
        }

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String line = value.toString();
            if (line.contains(keyword)) {
                String[] words = line.split("\\s+");
                for (String w : words) {
                    word.set(w);
                    context.write(word, one);
                }
            }
        }
    }

    // First Reducer: Count occurrences of each word
    public static class CountReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    // Second Mapper: Flip key and value for sorting by count
    public static class SortMapper extends Mapper<Object, Text, IntWritable, Text> {
        private IntWritable count = new IntWritable();
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String[] wordAndCount = value.toString().split("\\t");
            word.set(wordAndCount[0]);
            count.set(Integer.parseInt(wordAndCount[1]));
            context.write(count, word);
        }
    }

    // Second Reducer: Output sorted counts
    public static class SortReducer extends Reducer<IntWritable, Text, Text, IntWritable>
{
        public void reduce(IntWritable key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
            for (Text word : values) {
                context.write(word, key);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("keyword", args[2]);  // Set the search keyword from command-line argumen
t

        // Job 1: Search and Count Job
        Job searchJob = Job.getInstance(conf, "search and count");
        searchJob.setJarByClass(SearchAndSort.class);
        searchJob.setMapperClass(SearchMapper.class);
        searchJob.setReducerClass(CountReducer.class);
        searchJob.setOutputKeyClass(Text.class);
        searchJob.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(searchJob, new Path(args[0]));
        Path tempOutputPath = new Path("temp_output");
        FileOutputFormat.setOutputPath(searchJob, tempOutputPath);

        // Job 2: Sorting Job
        Job sortJob = Job.getInstance(new Configuration(), "sort by count");
        sortJob.setJarByClass(SearchAndSort.class);
        sortJob.setMapperClass(SortMapper.class);
        sortJob.setReducerClass(SortReducer.class);
        sortJob.setMapOutputKeyClass(IntWritable.class);
        sortJob.setMapOutputValueClass(Text.class);
        sortJob.setOutputKeyClass(Text.class);
        sortJob.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(sortJob, tempOutputPath);
        FileOutputFormat.setOutputPath(sortJob, new Path(args[1]));

        // Control job dependencies
        ControlledJob controlledSearchJob = new ControlledJob(searchJob.getConfiguration
());
```

```
        ControlledJob controlledSortJob = new ControlledJob(sortJob.getConfiguration());
        controlledSortJob.addDependingJob(controlledSearchJob);

        JobControl jobControl = new JobControl("Search and Sort");
        jobControl.addJob(controlledSearchJob);
        jobControl.addJob(controlledSortJob);

        Thread jobControlThread = new Thread(jobControl);
        jobControlThread.start();

        while (!jobControl.allFinished()) {
            Thread.sleep(1000);
        }

        System.exit(jobControl.getFailedJobList().isEmpty() ? 0 : 1);
    }
}
```

Explanation of Code

1. **SearchMapper**: Filters lines containing a specific keyword (set by `` `conf.set("keyword", args[2])` ``) and emits each word in those lines with a count of `` `1` ``.
2. **CountReducer**: Sums counts for each word from `` `SearchMapper` ``.
3. **SortMapper**: Flips each word and count (outputs `` `count` `` as key and `` `word` `` as value) for sorting.
4. **SortReducer**: Outputs words sorted by count.

Steps to Compile and Execute

1. **Compile**:

   ```bash
   javac -classpath $(hadoop classpath) -d . SearchAndSort.java
   jar -cvf searchandsort.jar *.class
   ```

2. **Run the Job**:

   ```bash
   hadoop jar searchandsort.jar SearchAndSort /input_path /output_path keyword
   ```

   Replace `` `/input_path` ``, `` `/output_path` ``, and `` `keyword` `` with actual values.

Notes

- This code runs two MapReduce jobs in sequence: the first counts word occurrences filtered by the keyword, and the second sorts the results by count.
- Ensure Hadoop is configured properly, and use appropriate paths for `` `input` ``, `` `output` ``, and `` `temp_output` ``.

Give Java mapreduce code for TF-IDF

Here's a Java MapReduce example for computing **TF-IDF (Term Frequency-Inverse Document Frequency)**. The TF-IDF measure is widely used in information retrieval to evaluate the importance of a word in a document relative to a collection of documents.

TF-IDF Calculation Steps

1. **Term Frequency (TF)**: Measures the frequency of a word in a document.
    - Formula: $\text{TF} = \frac{\text{Number of occurrences of word in document}}{\text{Total words in document}}$
2. **Inverse Document Frequency (IDF)**: Measures the importance of the word across all documents.
    - Formula: $\text{IDF} = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing the word}}\right)$
3. **TF-IDF**: Product of TF and IDF.
    - Formula: $\text{TF-IDF} = \text{TF} \times \text{IDF}$

Implementation in MapReduce

This Java MapReduce code performs TF-IDF in three main jobs:

1. **Job 1**: Compute term frequency (TF) for each word in each document.
2. **Job 2**: Count the number of documents containing each word (to compute IDF).
3. **Job 3**: Calculate TF-IDF.

Code Structure

1. **TFMapper** and **TFReducer**: Calculate TF for each word in each document.
2. **IDFMapper** and **IDFReducer**: Count documents containing each word.
3. **TFIDFMapper** and **TFIDFReducer**: Calculate TF-IDF using TF and IDF values.

Complete Code

```java
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class TFIDF {
    // Job 1: Mapper for Term Frequency
    public static class TFMapper extends Mapper<Object, Text, Text, IntWritable> {
        private IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String[] docWords = value.toString().split("\\s+");
            String docID = docWords[0]; // Assuming docID is the first word
            for (int i = 1; i < docWords.length; i++) {
                word.set(docWords[i] + "@" + docID);
                context.write(word, one);
            }
        }
    }

    public static class TFReducer extends Reducer<Text, IntWritable, Text, FloatWritable>
{
        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 FloatWritable(sum));
        }
    }

    // Job 2: Mapper for Document Frequency (IDF)
    public static class IDFMapper extends Mapper<Object, Text, Text, Text> {
        private Text word = new Text();
        private Text docID = new Text();

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String[] wordDocAndCount = value.toString().split("\\t");
            String[] wordAndDoc = wordDocAndCount[0].split("@");
            word.set(wordAndDoc[0]);
            docID.set(wordAndDoc[1]);
            context.write(word, docID);
        }
    }

    public static class IDFReducer extends Reducer<Text, Text, Text, FloatWritable> {
        private IntWritable totalDocs = new IntWritable(1000);  // Set total document coun
t manually

        public void reduce(Text key, Iterable<Text> values, Context context) throws IOExce
ption, InterruptedException {
            int docCount = 0;
            for (Text val : values) {
                docCount++;
            }
            float idf = (float) Math.log((float) totalDocs.get() / (docCount + 1));
            context.write(key, new FloatWritable(idf));
        }
    }

    // Job 3: Mapper for TF-IDF
    public static class TFIDFMapper extends Mapper<Object, Text, Text, Text> {
        private Text wordDoc = new Text();
        private Text tfOrIdfValue = new Text();

        public void map(Object key, Text value, Context context) throws IOException, Inter
ruptedException {
            String[] wordAndDocOrIdf = value.toString().split("\\t");
            wordDoc.set(wordAndDocOrIdf[0]);
            tfOrIdfValue.set(wordAndDocOrIdf[1]);
            context.write(wordDoc, tfOrIdfValue);
        }
    }

    public static class TFIDFReducer extends Reducer<Text, Text, Text, FloatWritable> {
        public void reduce(Text key, Iterable<Text> values, Context context) throws IOExce
ption, InterruptedException {
            float tf = 0;
            float idf = 0;
            for (Text val : values) {
                String[] tfOrIdf = val.toString().split("=");
                if (tfOrIdf[0].equals("tf")) {
                    tf = Float.parseFloat(tfOrIdf[1]);
                } else if (tfOrIdf[0].equals("idf")) {
                    idf = Float.parseFloat(tfOrIdf[1]);
                }
            }
            float tfidf = tf * idf;
            context.write(key, new FloatWritable(tfidf));
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        // Job 1: Calculate Term Frequency
        Job tfJob = Job.getInstance(conf, "term frequency");
        tfJob.setJarByClass(TFIDF.class);
        tfJob.setMapperClass(TFMapper.class);
        tfJob.setReducerClass(TFReducer.class);
        tfJob.setOutputKeyClass(Text.class);
        tfJob.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(tfJob, new Path(args[0]));
        Path tfOutputPath = new Path("tf_output");
        FileOutputFormat.setOutputPath(tfJob, tfOutputPath);

        if (!tfJob.waitForCompletion(true)) {
            System.exit(1);
        }

        // Job 2: Calculate Document Frequency for IDF
        Job idfJob = Job.getInstance(conf, "document frequency");
        idfJob.setJarByClass(TFIDF.class);
        idfJob.setMapperClass(IDFMapper.class);
        idfJob.setReducerClass(IDFReducer.class);
        idfJob.setOutputKeyClass(Text.class);
        idfJob.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(idfJob, tfOutputPath);
        Path idfOutputPath = new Path("idf_output");
        FileOutputFormat.setOutputPath(idfJob, idfOutputPath);

        if (!idfJob.waitForCompletion(true)) {
            System.exit(1);
        }

        // Job 3: Calculate TF-IDF
        Job tfidfJob = Job.getInstance(conf, "tf-idf");
        tfidfJob.setJarByClass(TFIDF.class);
        tfidfJob.setMapperClass(TFIDFMapper.class);
        tfidfJob.setReducerClass(TFIDFReducer.class);
        tfidfJob.setOutputKeyClass(Text.class);
        tfidfJob.setOutputValueClass(FloatWritable.class);
        FileInputFormat.addInputPath(tfidfJob, idfOutputPath);
        FileOutputFormat.setOutputPath(tfidfJob, new Path(args[1]));

        System.exit(tfidfJob.waitForCompletion(true) ? 0 : 1);
    }
}
```

Explanation of the Code

1. **Job 1: TF Calculation**
    - **TFMapper**: Reads each document, counts occurrences of each word, and associates it with the document ID.
    - **TFReducer**: Sums the word occurrences to calculate TF for each word in each document.

2. **Job 2: IDF Calculation**
   - **IDFMapper**: Maps each word to each document containing it.
   - **IDFReducer**: Counts documents containing each word and calculates IDF using the total document count.

3. **Job 3: TF-IDF Calculation**
   - **TFIDFMapper**: Maps TF and IDF values for each word in each document.
   - **TFIDFReducer**: Multiplies TF and IDF to get the TF-IDF value.

Compiling and Running

1. **Compile**:

   ```bash
   javac -classpath $(hadoop classpath) -d . TFIDF.java
   jar -cvf tfidf.jar *.class
   ```

2. **Run the Jobs**:

   ```bash
   hadoop jar tfidf.jar TFIDF /input_path /output_path
   ```

This setup should calculate and output the TF-IDF values for each word in each document in the collection. Adjust `/input_path` and `/output_path` to your HDFS locations.

Give pyspark code to implement mllib package

Here's an example of how to use the **MLlib** package in **PySpark** to implement **machine learning algorithms**. MLlib in PySpark provides a powerful set of tools for scalable machine learning and covers various algorithms, including classification, regression, clustering, collaborative filtering, and dimensionality reduction.

Let's walk through a simple example of a **classification task** using the **Logistic Regression** algorithm on a sample dataset.

Example: Logistic Regression using MLlib in PySpark

This example demonstrates:

1. Loading and preprocessing data.

2. Using MLlib's `LogisticRegression` model.
3. Training and evaluating the model.

Sample Code

```python
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("MLlib Example - Logistic Regression") \
    .getOrCreate()

# Load dataset (for example, using Iris dataset format)
data = spark.read.csv("data/iris.csv", header=True, inferSchema=True)

# Show data schema
data.printSchema()

# Preprocess data - assemble features
feature_columns = [col for col in data.columns if col != "label"]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(data)

# Split data into training and test sets
train_data, test_data = data.randomSplit([0.7, 0.3], seed=42)

# Initialize the Logistic Regression model
lr = LogisticRegression(featuresCol="features", labelCol="label")

# Create a parameter grid for cross-validation (for tuning)
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.01, 0.1, 0.5]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
    .build()

# Cross-validation with 3-fold
crossval = CrossValidator(estimator=lr,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=3)

# Fit the model on training data
cvModel = crossval.fit(train_data)

# Make predictions on the test data
predictions = cvModel.transform(test_data)

# Show predictions
predictions.select("label", "prediction", "probability").show(5)

# Evaluate the model
evaluator = BinaryClassificationEvaluator()
accuracy = evaluator.evaluate(predictions)
print(f"Test Accuracy = {accuracy}")

# Stop the SparkSession
spark.stop()
```

Explanation of Code Components

1. **Data Loading**:
   - We load data from a CSV file. Here, you would replace `` `"data/iris.csv"` `` with your dataset file path.
2. **Feature Engineering**:
   - Use `` `VectorAssembler` `` to combine multiple feature columns into a single vector column named `` `"features"` ``.
3. **Data Splitting**:
   - Split the data into **training** and **testing** sets.
4. **Model Setup**:
   - Initialize a **Logistic Regression** model.
5. **Hyperparameter Tuning with Cross-Validation**:
   - Use `` `CrossValidator` `` with `` `ParamGridBuilder` `` to specify a range of values for hyperparameters, tuning the model to achieve the best performance.
6. **Model Training**:
   - Train the model on the training data with cross-validation.
7. **Model Evaluation**:
   - Make predictions on the test data and evaluate the model accuracy using `` `BinaryClassificationEvaluator` ``.

Important Notes

- **MLlib** provides other classifiers, such as **RandomForestClassifier**, **DecisionTreeClassifier**, and **NaiveBayes**, which can be used similarly.
- Replace `` `"label"` `` and feature columns as per your dataset.
- Use `` `spark.read.parquet` `` or other file formats as needed.

Implement graphx package in pyspark

PySpark does not directly support GraphX (which is a Scala-based library for graph processing). However, **GraphFrames**, a library built on top of Apache Spark, provides similar capabilities for graph processing and is fully compatible with PySpark. You can use GraphFrames to perform graph-based computations such as finding connected components, running PageRank, and more.

Here's how you can use **GraphFrames** in PySpark to perform some common graph processing tasks.

Step 1: Install GraphFrames

First, you'll need to install the GraphFrames package. It's not included in PySpark by default, so you can install it with:

```bash
pip install graphframes
```

Step 2: Using GraphFrames for Graph Processing

Here's an example of using GraphFrames to create a graph, add vertices and edges, and run common graph algorithms such as **PageRank** and **Connected Components**.

Example Code

```python
# Import necessary libraries
from pyspark.sql import SparkSession
from graphframes import GraphFrame

# Initialize SparkSession with GraphFrames package
spark = SparkSession.builder \
    .appName("GraphFrames Example") \
    .getOrCreate()

# Define vertices (nodes of the graph)
vertices = spark.createDataFrame([
    ("1", "Alice", 34),
    ("2", "Bob", 36),
    ("3", "Charlie", 30),
    ("4", "David", 29),
    ("5", "Esther", 32),
    ("6", "Fanny", 36),
    ("7", "Gabby", 27)
], ["id", "name", "age"])

# Define edges (relationships between nodes)
edges = spark.createDataFrame([
    ("1", "2", "friend"),
    ("2", "3", "follow"),
    ("3", "4", "follow"),
    ("4", "5", "friend"),
    ("5", "6", "follow"),
    ("6", "7", "friend"),
    ("7", "1", "follow"),
    ("3", "1", "friend")
], ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(vertices, edges)

# Show vertices and edges
print("Vertices:")
g.vertices.show()

print("Edges:")
g.edges.show()

# Run PageRank algorithm on the graph
print("Running PageRank...")
page_rank = g.pageRank(resetProbability=0.15, maxIter=10)
print("PageRank Results:")
page_rank.vertices.select("id", "pagerank").show()
```

```
# Run Connected Components algorithm
print("Running Connected Components...")
connected_components = g.connectedComponents()
print("Connected Components Results:")
connected_components.select("id", "component").show()

# Run Breadth-First Search (BFS) from a specific source to destination
print("Running BFS...")
bfs_result = g.bfs(fromExpr="id = '1'", toExpr="id = '4'")
print("BFS Result from id=1 to id=4:")
bfs_result.show()

# Stop the Spark session
spark.stop()
```

Explanation of the Code

1. **Creating the Graph**:
    - **Vertices**: Each vertex represents a person with `` `id` ``, `` `name` ``, and `` `age` `` attributes.
    - **Edges**: Each edge represents a relationship (like `` `"friend"` `` or `` `"follow"` ``) between two vertices (`` `src` `` and `` `dst` ``).
2. **PageRank**:
    - PageRank is run with a reset probability of 0.15 and 10 iterations. This algorithm ranks each vertex based on its connectivity.
3. **Connected Components**:
    - Connected Components finds clusters of connected nodes in the graph. Each component is labeled with a unique component ID.
4. **Breadth-First Search (BFS)**:
    - BFS finds the shortest path from a specified starting node to a target node. Here, we search from the node with `` `id = '1'` `` to the node with `` `id = '4'` ``.

Important Notes

- **GraphFrames** is highly compatible with PySpark, making it a great choice for Python-based graph processing with Spark.
- For larger graphs, it's efficient because it uses Spark's distributed processing capabilities.
- In addition to PageRank, Connected Components, and BFS, GraphFrames supports other algorithms such as **Label Propagation**, **Shortest Paths**, and **Triangle Counting**.

This example covers basic graph processing, but GraphFrames can be applied to more complex use cases in graph analytics.

Teach me pig latin coding with example. Cover all the topics in pig latin

Apache **Pig Latin** is a high-level data flow language used with Apache Pig, a platform for analyzing large datasets in Hadoop. Pig Latin is especially useful for ETL (Extract, Transform, Load) tasks and can manage both structured and unstructured data.

Let's go through the key concepts and constructs of Pig Latin, with examples that cover data loading, transformation, filtering, grouping, joining, and more.

1. Setting up the Environment

First, launch the Pig interactive shell using the command:

```bash
pig
```

Alternatively, you can run Pig scripts saved in a file using:

```bash
pig -x local script_name.pig
```

Note: The `-x local` flag runs Pig in local mode (not on Hadoop), which is good for testing.

2. Loading Data

In Pig, data is loaded using the `LOAD` statement.

```pig
-- Load data from a file
students = LOAD 'students.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int,
grade:chararray);

-- Example data in students.txt:
-- 1,John,20,A
-- 2,Alice,22,B
```

3. Viewing Data

To see a sample of the data:

```pig
DUMP students;
```

4. Basic Data Transformation

a) `FOREACH ... GENERATE`

Use `FOREACH` to iterate over each row and `GENERATE` to create new data transformations.

```pig
-- Select only 'name' and 'age' columns
student_names = FOREACH students GENERATE name, age;
DUMP student_names;
```

b) `FILTER`

`FILTER` removes rows based on a condition.

```pig
-- Filter students older than 21
older_students = FILTER students BY age > 21;
DUMP older_students;
```

c) `ORDER`

`ORDER` sorts the data based on specified columns.

```pig
-- Sort students by age in descending order
sorted_students = ORDER students BY age DESC;
DUMP sorted_students;
```

d) `DISTINCT`

`DISTINCT` removes duplicate rows.

```pig
-- Get unique grades
unique_grades = DISTINCT students.grade;
DUMP unique_grades;
```

e) `LIMIT`

`LIMIT` restricts the number of rows in the output.

```pig
-- Limit output to first 3 records
limited_students = LIMIT students 3;
```

```
  DUMP limited_students;
```

5. Grouping and Aggregation

a) `GROUP`

`GROUP` collects data by a specified field, similar to SQL `GROUP BY`.

```
pig


  -- Group students by grade
  grouped_by_grade = GROUP students BY grade;
  DUMP grouped_by_grade;
```

b) Aggregation Functions

Common aggregation functions include `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.

```
pig


  -- Count students in each grade
  grade_counts = FOREACH grouped_by_grade GENERATE group AS grade, COUNT(students) AS count;
  DUMP grade_counts;
```

6. Joins

Joining allows you to combine data from two or more datasets.

```
pig


  -- Assume 'courses.txt' contains (student_id:int, course:chararray)
  courses = LOAD 'courses.txt' USING PigStorage(',') AS (student_id:int, course:chararray);

  -- Join students with courses on student ID
  joined_data = JOIN students BY id, courses BY student_id;
  DUMP joined_data;
```

7. Advanced Data Transformation

a) `CROSS`

`CROSS` creates a Cartesian product of two datasets.

```
pig


  cross_data = CROSS students, courses;
  DUMP cross_data;
```

b) `FLATTEN`

`FLATTEN` is used to unpack a nested field.

```pig
-- Use GROUP followed by FLATTEN to get a more readable structure
flattened_data = FOREACH grouped_by_grade GENERATE FLATTEN(grouped_by_grade);
DUMP flattened_data;
```

8. Using Built-in Functions

Pig provides many built-in functions for mathematical, string, and date operations.

```pig
-- String functions
upper_names = FOREACH students GENERATE UPPER(name);

-- Mathematical functions
increased_age = FOREACH students GENERATE age + 1;

-- Date functions (assuming date format is supported)
-- formatted_date = FOREACH data GENERATE ToDate(birth_date, 'yyyy-MM-dd');
```

9. Writing Data

Use the `STORE` command to save data back to HDFS or a local file.

```pig
STORE sorted_students INTO 'output/sorted_students' USING PigStorage(',');
```

10. User-Defined Functions (UDFs)

For complex logic, you can write your own UDFs in Java, Python, or other supported languages.

```python
-- In Python, save this file as "my_udf.py"
@outputSchema("name:chararray")
def capitalize_name(name):
    return name.capitalize() if name else name
```

In Pig, register the UDF and use it:

```pig
REGISTER 'my_udf.py' USING jython AS my_udf;
```

```
capitalized_names = FOREACH students GENERATE my_udf.capitalize_name(name);
DUMP capitalized_names;
```

11. Example End-to-End Script

Here’s a full Pig script example that combines all of the above steps:

```pig
-- Load students and courses data
students = LOAD 'students.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int,
grade:chararray);
courses = LOAD 'courses.txt' USING PigStorage(',') AS (student_id:int, course:chararray);

-- Filter students over 21 and sort by age
older_students = FILTER students BY age > 21;
sorted_older_students = ORDER older_students BY age DESC;

-- Join with courses data
joined_data = JOIN sorted_older_students BY id, courses BY student_id;

-- Group by grade and count
grouped_by_grade = GROUP students BY grade;
grade_counts = FOREACH grouped_by_grade GENERATE group AS grade, COUNT(students) AS count;

-- Store results
STORE grade_counts INTO 'output/grade_counts' USING PigStorage(',');
```

Summary of Key Pig Latin Topics

- **Data Loading**: `LOAD`
- **Transformations**: `FOREACH`, `FILTER`, `ORDER`, `DISTINCT`, `LIMIT`
- **Grouping and Aggregation**: `GROUP`, aggregation functions like `COUNT`, `SUM`, `AVG`
- **Joins**: `JOIN`, `CROSS`
- **Advanced Operations**: `FLATTEN`, built-in functions
- **User-Defined Functions (UDFs)**: For custom operations

Give Java mapreduce code for searching alone

Here's an example of Java MapReduce code that searches for a specific term in a dataset and outputs the lines containing that term. This is often called "grep" in MapReduce, inspired by the Unix `grep` command.

The code takes a search term (e.g., "Hadoop") and filters lines in a file containing this term.

Code Explanation

1. **Mapper**: Reads each line, checks if the line contains the search term, and emits the line if it does.

2. **Reducer**: Collects and outputs the matching lines (though for this case, it's optional since there is no further aggregation needed).

Example Code for Searching in MapReduce

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class SearchMapReduce {

    public static class SearchMapper extends Mapper<LongWritable, Text, Text, Text> {
        private String searchTerm;

        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            // Retrieve the search term from the configuration
            Configuration conf = context.getConfiguration();
            searchTerm = conf.get("searchTerm").toLowerCase(); // Convert to lowercase for
case-insensitive search
        }

        @Override
        protected void map(LongWritable key, Text value, Context context) throws IOExcepti
on, InterruptedException {
            String line = value.toString().toLowerCase(); // Convert line to lowercase

            // Check if the line contains the search term
            if (line.contains(searchTerm)) {
                context.write(new Text("Match"), value); // Output the matching line
            }
        }
    }

    public static class SearchReducer extends Reducer<Text, Text, Text, Text> {
        @Override
        protected void reduce(Text key, Iterable<Text> values, Context context) throws IOE
xception, InterruptedException {
            for (Text value : values) {
                context.write(key, value); // Output each matching line
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        // Set the search term in the configuration
        conf.set("searchTerm", args[2]); // Pass search term as a command-line argument

        Job job = Job.getInstance(conf, "Search for Term in File");
        job.setJarByClass(SearchMapReduce.class);
        job.setMapperClass(SearchMapper.class);
        job.setReducerClass(SearchReducer.class);

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

        FileInputFormat.addInputPath(job, new Path(args[0])); // Input file path
        FileOutputFormat.setOutputPath(job, new Path(args[1])); // Output directory path

        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
    }
```

Explanation of Command-line Arguments

To run this MapReduce job, provide the following arguments:

1. **Input Path**: Location of the input file.
2. **Output Path**: Location for output.
3. **Search Term**: Term you want to search for.

Example Execution Command

```bash
hadoop jar SearchMapReduce.jar SearchMapReduce /path/to/input /path/to/output Hadoop
```

How It Works

1. **Setup Configuration**: The search term is passed as a command-line argument and stored in the job configuration.
2. **Mapper**:
   - Reads each line of the input.
   - Checks if the line contains the search term.
   - If a match is found, it emits the line with a key (e.g., "Match").
3. **Reducer**:
   - Outputs each matching line without modification.

This approach makes the job efficient by filtering out only the relevant lines during the map phase. The reducer simply outputs the matches (or could be omitted entirely if no further processing is required).