Documentation for the "CountRecs" Hadoop MapReduce job:

This Hadoop MapReduce job is designed to count the number of lines in a given dataset, excluding the header line. The job is implemented using three classes: CountRecs (the driver class), CountRecsMapper (the mapper class), and CountRecsReducer (the reducer class). The CountRecs class is responsible for setting up the job and configuring the mapper and reducer classes, while the mapper and reducer classes handle the actual counting of lines.

The input to the job is provided via the command line arguments, which are passed to the main() method of the CountRecs class. Specifically, the input path and output path are expected to be provided as the first and second arguments, respectively. If these arguments are not provided or if an incorrect number of arguments are provided, an error message is displayed and the program exits.

Once the input and output paths are validated, a new Hadoop job is created using the Job class. The job is then configured with the appropriate jar file (using job.setJarByClass()) and a human-readable job name (using job.setJobName()). In addition, the job is configured to use only one reducer (using job.setNumReduceTasks(1)), to ensure that the output is written to a single file.

The input and output directories are set using the FileInputFormat.addInputPath() and FileOutputFormat.setOutputPath() methods, respectively. These methods accept Path objects as arguments, which represent the file or directory location on Hadoop Distributed File System (HDFS). The mapper and reducer classes are specified using job.setMapperClass() and job.setReducerClass() methods.

The mapper class CountRecsMapper extends the Hadoop Mapper class and overrides its map() method. The map() method takes a LongWritable key (representing the byte offset of the current line), a Text value (representing the content of the current line), and a Context object (used to emit output from the mapper). If the first character of the current line is not 'U' (indicating that it is not the header line), the mapper writes a new key-value pair to the output context with the key "Lines in original dataset, before cleaning" and the value 1.

The reducer class CountRecsReducer extends the Hadoop Reducer class and overrides its reduce() method. The reduce() method takes a Text key (representing the key emitted by the mapper), an Iterable of IntWritable values (representing the values emitted by the mapper for the given key), and a Context object (used to emit output from the reducer). The reducer simply sums up the values for each key and emits the result as a key-value pair to the output context.

Finally, the driver class calls job.waitForCompletion(true) to submit the job to Hadoop and wait for its completion. If the job completes successfully, the program exits with exit code 0; otherwise, it exits with exit code 1.

In summary, this Hadoop MapReduce job counts the number of lines in a dataset, excluding the header line, using a single reducer. The job is designed to be run on a Hadoop cluster and accepts input and output paths via command line arguments.

Documentation for the "Clean" Hadoop MapReduce job:

This Hadoop MapReduce job is designed to clean a dataset by removing unnecessary columns and rows with missing data. The job is implemented using three classes: Clean (the driver class), CleanMapper (the mapper class), and CleanReducer (the reducer class). The Clean class is responsible for setting up the job and configuring the mapper and reducer classes, while the mapper and reducer classes handle the actual cleaning of the dataset.

The input to the job is provided via the command line arguments, which are passed to the main() method of the Clean class. Specifically, the input path and output path are expected to be provided as the first and second arguments, respectively. If these arguments are not provided or if an incorrect number of arguments are provided, an error message is displayed and the program exits.

Once the input and output paths are validated, a new Hadoop job is created using the Job class. The job is then configured with the appropriate jar file (using job.setJarByClass()) and a human-readable job name (using job.setJobName()). In addition, the job is configured to use only one reducer (using job.setNumReduceTasks(1)), to ensure that the output is written to a single file.

The input and output directories are set using the FileInputFormat.addInputPath() and FileOutputFormat.setOutputPath() methods, respectively. These methods accept Path objects as arguments, which represent the file or directory location on Hadoop Distributed File System (HDFS). The mapper and reducer classes are specified using job.setMapperClass() and job.setReducerClass() methods.

The mapper class CleanMapper extends the Hadoop Mapper class and overrides its map() method. The map() method takes a LongWritable key (representing the byte offset of the current line), a Text value (representing the content of the current line), and a Context object (used to emit output from the mapper). The mapper performs the following cleaning tasks on each row:

1. It splits the row into an array of strings using the comma delimiter.
2. It checks that the row is not the header line (identified by the first character not being 'U') and has the correct number of values (39).
3. If the row is valid, it creates a new list of strings containing only the values from specific columns (identified by their index).
4. It joins the cleaned columns into a new string using the comma delimiter.
5. It emits the cleaned row as a key-value pair to the output context, where the key is the cleaned row and the value is 1.

The reducer class CleanReducer extends the Hadoop Reducer class and overrides its reduce() method. The reduce() method takes a Text key (representing the key emitted by the mapper), an Iterable of IntWritable values (representing the values emitted by the mapper for the given key), and a Context object (used to emit output from the reducer). The reducer simply sums up the values for each key and emits the result as a key-value pair to the output context.

Finally, the driver class calls job.waitForCompletion(true) to submit the job to Hadoop and wait for its completion. If the job completes successfully, the program writes a new header line to the output file (identified by the path "output/part-r-00000"), and the program exits with exit code 0; otherwise, it exits with exit code 1.

In summary, this Hadoop MapReduce job cleans a dataset by removing unnecessary columns and rows with missing data, using a single reducer. The job is designed to be run on a Hadoop cluster and accepts input and output paths via command line arguments. The cleaned output is stored in a single file with a new header line, and the original file is left

Scala Code Documentation

This is a Scala code that performs data analysis on two datasets related to restaurant inspections and food poisoning cases in New York City.

The code recieves its CSV from the MapReduce, which produces a cleaned version.

The code starts by importing the necessary libraries, including Spark SQL, Spark MLlib, and Apache Commons Math. It then reads in a CSV file containing food poisoning case data, cleans the data, and converts it to a DataFrame. The DataFrame is then manipulated to extract the relevant fields and convert them to the correct data types.

Next, the code reads in Ryan’s CSV file containing restaurant inspection data, cleans it, and computes various statistics based on the data, including the average inspection score and critical violation percentage for each zip code. It then merges this data with the food poisoning case data using the zip code as the key.

The code then calculates the Pearson correlation coefficient between the number of food poisoning cases and various other statistics, including the critical violation percentage, average inspection score, and average grade for each zip code. It also trains a logistic regression model to predict the grade of a zip code based on the number of food poisoning cases and the average inspection score.

Finally, the code writes the results of the analysis to a text file and saves it to a specified directory. The file contains the Pearson correlation coefficients and the accuracy of the logistic regression model.