



ITNPBD7 – Cluster Computing

Course Assignment

David Haveron – Student Number 2527317

Hadoop Distributed File System (HDFS)

Hadoop is an open source Big Data platform used for parallel, distributed computing. Hadoop consists of the Hadoop Distributed File System (HDFS), the MapReduce programming platform and the Hadoop Ecosystem. HDFS is the place in a Hadoop cluster which stores the very large data files (some files the order of petabytes) with streaming data access and run on clusters of commodity hardware. This streaming access quality implies the file system has been designed and optimized for read intensive operations (that is, data is written to HDFS once and read many times and the system only allows data to be appended, not amended). Much of the popularity for distributed computing was primarily driven by the significant advances in storage capacity which lead to the growth in assemblies of inexpensive commodity hardware (now called clusters). These commodity hardware services are analogous to the electricity we buy from energy companies today, where the user of the commodity hardware services (or the electricity) need not pay much attention for how the service is run, managed or maintained, just that the service is efficiently delivered – allowing the ‘customer’ to focus on their own objectives.

Traditionally, data has been stored and read from single servers however the exponential growth of data generated has invoked the need to store data in a distributed, systematic way. And, as these commodity clusters consist of many inexpensive nodes, these clusters occasionally experience node failures. Hadoop has been designed to work around this, with the use of block replication to effectively anticipate and recover from these failures through data replication on the HDFS. *Figure 1 on page 2* illustrates an abstract representation of the HDFS architecture and how key components of HDFS inter-communicate to copy data to HDFS and run Hadoop jobs.

This approach to replicate blocks is outlined as follows: HDFS primarily splits these large files into block-sized chunks of a fixed size (usually 64MB or larger). Each block is replicated a few times (usually three) and these blocks (and their replications) are stored as independent units and distributed across the nodes in the server.

The HDFS cluster consists of a single NameNode (master in combination with a secondary NameNode which keeps a current namespace image) and many DataNodes (worker). The NameNode manages the filesystem namespace and is analogous to the manager of a distribution warehouse, who is expected to understand the distribution and details of products in the warehouse. The NameNode has a similar responsibility in managing/storing the filesystem tree, the metadata for all the files/directories in the tree and data node content (that is, which file blocks are stored on a given DataNode). The DataNodes store (and retrieve) the blocks of data in addition to local information about block location/file identity. These DataNodes report periodically with the NameNode to communicate information about blocks they are storing (this keeps the NameNode ‘well informed’).

A client is used to access this filesystem by communicating with the NameNode and DataNodes. A client typically requests information from the NameNode about which DataNodes store a given files’ blocks. Once the client understands which DataNodes to talk to retrieve/query a file, the client talks directly to the appropriate DataNodes.

For demonstrative purposes, consider the movie review file provided for this Hadoop job. Assuming this file is large (in the order of gigabytes, terabytes or petabytes), an instruction by the Putty client is given to copy this file from the local filestore to the Hadoop Distributed File Store (HDFS). The NameNode communicates back to the client, the details of how to break up the large file into (replicated) blocks (usually 64MB or bigger) and where these blocks are to be distributed within HDFS (to comply with replication standards/architecture, after taking into consideration the current distribution of blocks in HDFS). The client, now equipped with the appropriate instructions, executes these instructions to break the file into block size chunks and distributes these as per instruction on HDFS. Similarly, to run a Hadoop job, the client communicates with the NameNode to find out the metadata (location/s) of the blocks which collectively represent a given file. These (DataNode) details are sent back to the client to execute. The client, then sends copies of the MapReduce job to be completed to the appropriate DataNodes, which run the job and return the results to the client.

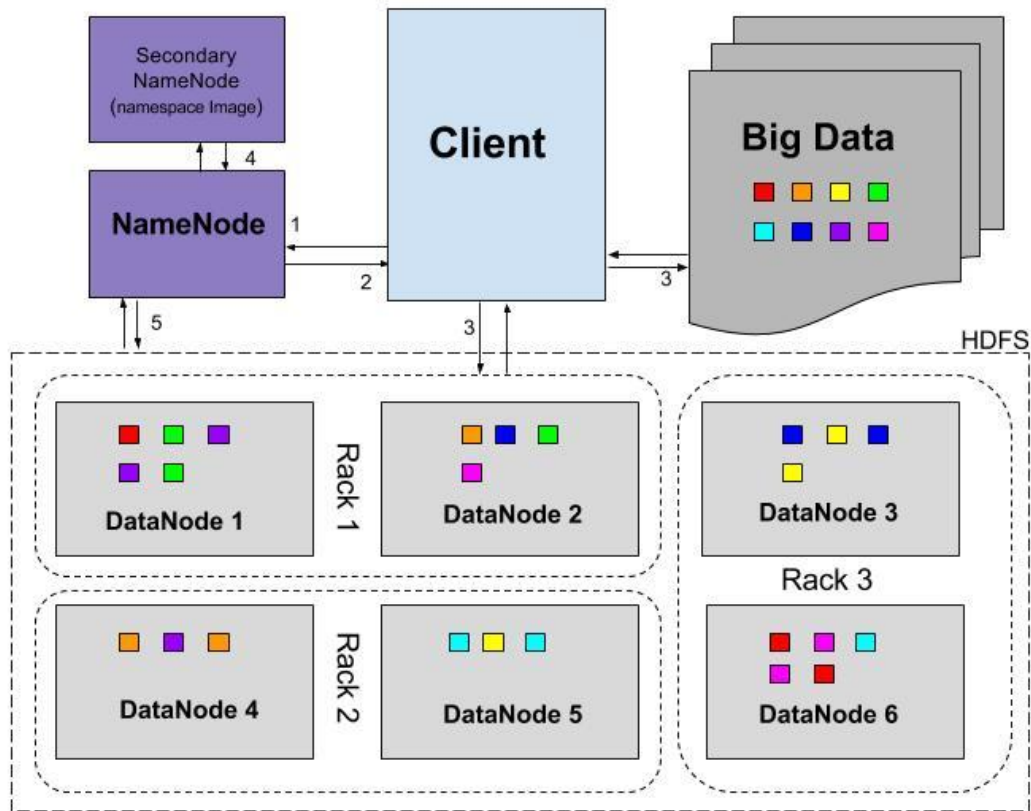


Figure 1 - Architecture of HDFS

1. Client sends a request to the NameNode to add large data file to HDFS
2. NameNode tells the client how and where to distribute the file
3. The Client breaks the data file into blocks and distributes the blocks to the DataNodes as per NameNode instruction (block operations)
4. Secondary NameNode communicates with NameNode to regularly backup the namespace image (note: not a replacement NameNode, just a restore point)
5. NameNode communicates with all the DataNodes in the cluster to store up-to-date metadata of the file and block distribution across DataNodes in the HDFS space.

Commands used to execute a Hadoop job

At a high level, MapReduce jobs are completed through a series of steps as outlined below (Note: a client 'PuTTY' is used to talk to the HDFS system):

1. A file directory to which the data will be copied, is created:
`hdfs dfs -mkdir /home/deh/Data`
2. The datafile/s are copied from the local filestore on the network/hard drive, to the HDFS directory created above:
`hdfs dfs -copyFromLocal RatedReviews.txt /home/deh/Data`
3. The file copy can be confirmed by listing the files in the newly created directory:
`hdfs dfs -ls /home/deh/data`
4. The MapReduce program written to achieve the job is compiled into the java .class file:
`hadoop com.sun.tools.javac.Main movieReview.java`
5. The jar file is then created:
`jar cf movieReview.jar movieReview *.class`
6. The job is then submitted to Hadoop for action, with the dataset file location, the results output location and the file containing the words to be excluded, specified:
`Hadoop jar movieReview.jar movieReview /home/deh/Data/RatedReviews.txt /home/deh/results /home/deh/Data/wordsToBeExcluded.txt`
7. Assuming the MapReduce job executed without problems, the result file can be seen in the results file directory specified in the command above:
`hdfs dfs -ls /home/deh/results`
8. The file results can be read by opening the file:
`hdfs dfs -cat /home/deh/results/part-r-00000`

Example Snippet of the Data File


"in my opinion , a movie reviewer's most important task is to offer an explanation of an uncontrollable rant , obligations damned . however , protocol forces me to do so to provoke the audience into feeling any emotion . when the movie's camerawork is as good as anticipation . we already know what to look for , so why should we be surprised ? the movie relies on random twists and turns with a minimum of logic and loads of laziness and the lamest screenwriting devices around . and guess what ? this movie has both a plot and is worthwhile . since there's no captivating dialogue , no character chemistry and no scenes right , which are becoming more prevalent in pg-13 movies like bring it on , together at a club , he rarely shows them in a full shot and he never keeps them from being undercut by its stupidity . why wouldn't they wash the clothes in the sink or be sorry for affleck , who i've liked in other movies , and bentley , who was great in the last , compelled to knock on doors and warn people about it ." 1

"you can watch this movie , that is based on a sci-fi work by robert heinlein , and it will stimulate your brain cells , because you will either find this film entertaining and a waste of time or something . but i have come to the conclusion that this insipid b- movie , is a mediocre and a sardonic one liner , such as this one , that makes light of the film's heavy subject matter . it can be seen as a war movie , because its spoof of the military , grew thin very fast when it came to the first place , that by spending so much time with spoofing the military , it lost its focus , as it makes use of its astronomical budget to become solely a computerized action movie . i will be doing my duty as a movie reviewer , if i should fail to point out that the movie is a waste of time . loves carmen (denise richards) , the school beauty and math brain , who is a waste of time unless you are really into special effects or found this sophomoric satire fun . a waste of time conflagration film . " 2

MapReduce Design

The previous section of this report mentioned that Hadoop consists of the Hadoop Distributed File System (HDFS), the MapReduce program and the Hadoop Ecosystem. The MapReduce program is effectively a query or set of instructions (often written in Hadoop's native language, Java) which is sent to DataNodes (to execute on the appropriate DataNode blocks) across the cluster. The MapReduce program is defined by a Map and a Reduce phase with each phase receiving an input and emitting an output. As the Map function runs first in the MapReduce program, the mapper loads data from blocks on the HDFS, parses, transforms or filters the data and finally emits a key-value pair to the context. The reducer program receives these key-value pairs from the context and iterates through the data it receives from all the mappers to yield an output (where the specifics of the Map and Reduce program are defined by the programmer).

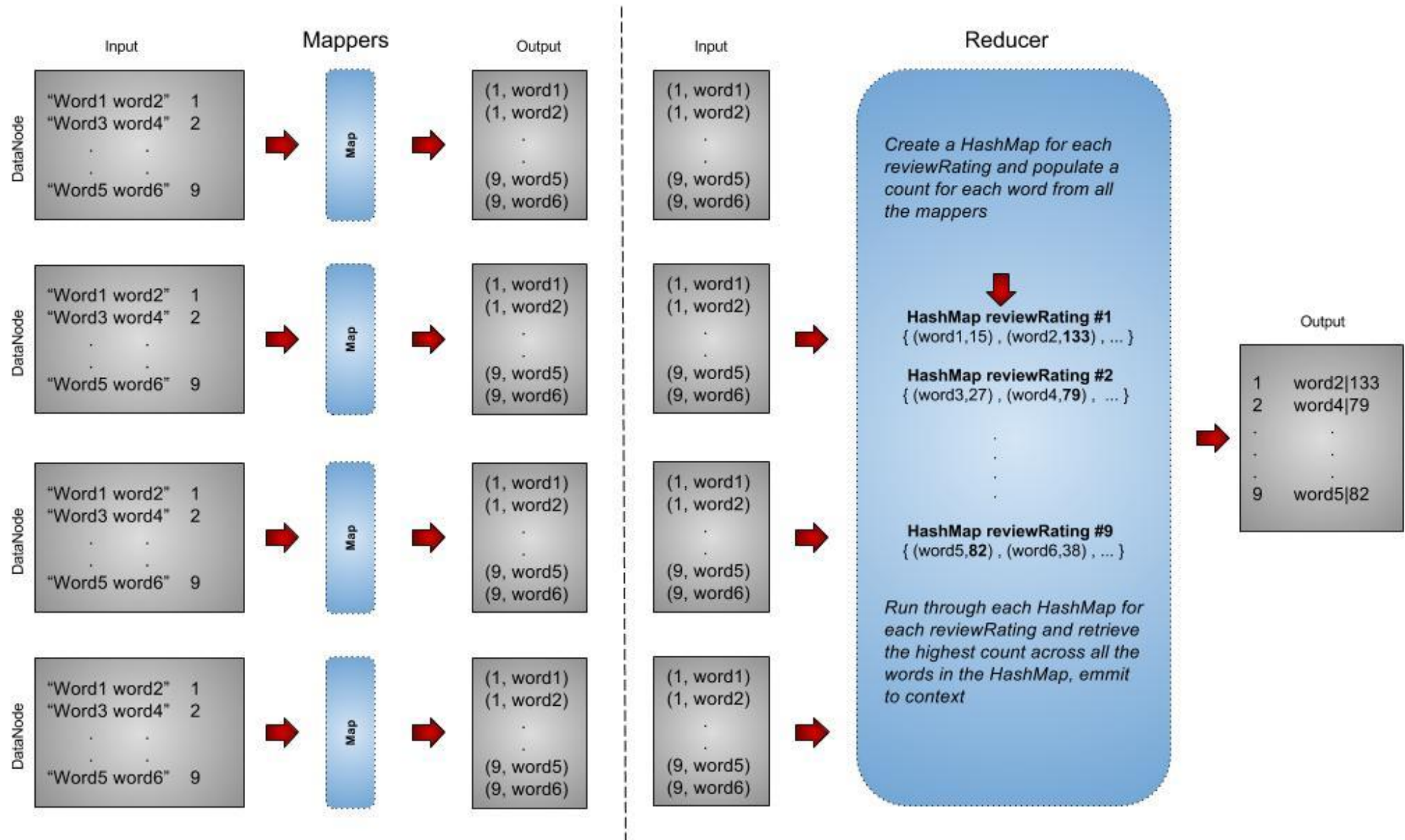
Two MapReduce designs were considered to process the movie review data (the file of which was loaded onto the HDFS) to produce a file with the desired output – a list of movie review ratings, with the most common word found for that rating (excluding a file of stop words). The objective of considering two designs was to develop a design which minimises the flow of data as far as reasonably practicable. *Figure 2* below, illustrates the format expected after the MapReduce job is done – the format of the results Hadoop returns will be in the format below where the first column is *Review Rating*, then *the Most Frequent Word|Frequency*:



| | |
|---|---------------|
| 1 | Poor 42 |
| 2 | Bad 15 |
| 3 | Boring 53 |
| 4 | Dull 72 |
| 5 | Average 73 |
| 6 | Okay 85 |
| 7 | Great 37 |
| 8 | Brilliant 26 |
| 9 | Phenomenal 43 |

Figure 2 - Expected Hadoop Results (format only, not values)

Design 1



Design 1 discussion:

This initial design (seen in the schematic above), primarily reads the data into the mappers, where the data is split into two parts: a string with the review words and the reviewRating (number). The string is processed to remove any punctuation and bring the data into a lower case.

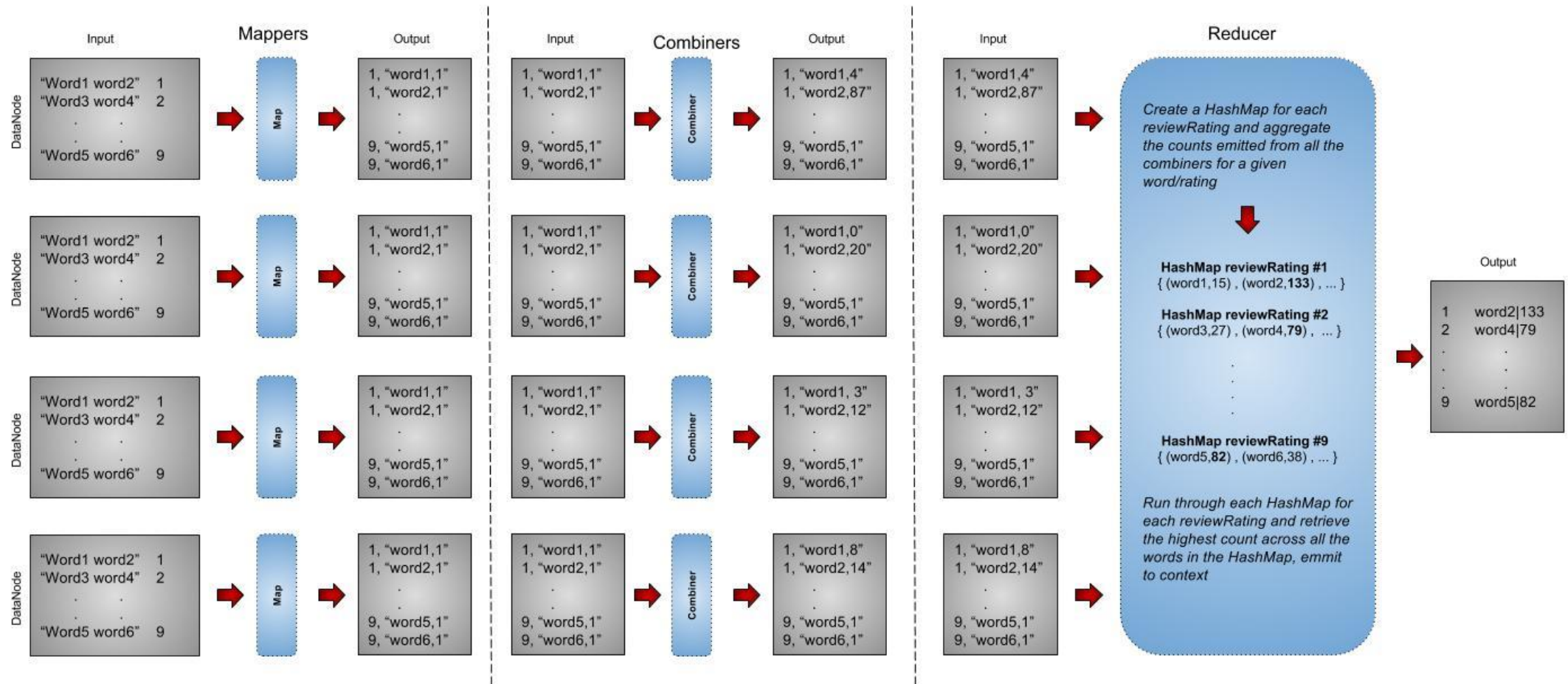
The program iterates over the tokens in the string and checks each word against the words which are to be excluded from the count.

The words iterated over, which are not in the exclusion list are passed through to the context as a key-value pair produced as ("reviewRating", "word") for all the reviewRating keys (1-9) across the words associated with each reviewRating.

The reducer then receives these key-value pairs, and begins to populate a HashMap where a count of frequency for a given word is kept (this is done for nine times, once for each reviewRating). The HashMap values are iterated over (again once for each reviewRating, 1 – 9) to pull out the highest frequency found and its associated word.

Although this functionally works to achieve the desired output, there is a relatively large amount of data being passed from the mapper to the reducer. This idea is refined in Design 2, below.

Design 2



Design 2 discussion:

This second design (which can be seen above), similarly reads the data into the mappers, where the data is split into a string with the review words and the reviewRating (number). The string is processed to remove any punctuation and bring the data into a lower case.

The program iterates over the tokens in the string and checks each word against the words which are to be excluded from the count.

The words which are not in the exclusion list are passed on to create a compound "value" consisting of a concatenation of the "word" with "1" or "word,1" as a compound comma separated string. These compound values are written to the context with the reviewRating number (as the key) in which that word was found. These key-value pairs are passed to the combiner on each mapper to locally aggregate.

The combiners create a local HashMap to count the frequency of words for a given reviewRating, and repeats this for all the reviewRatings (1-9). For each HashMap, the word-count pairs are recombined again to form the compound string value "word,27" (for example) with the key defined as the reviewRating (or HashMap) number.

These subtotals from the combiner/s are passed to the reducer for final aggregation. As with the local aggregation in the combiner, the reducer iterates over the values received, separates the compound string and aggregates the sub counts in a HashMap. For each HashMap populated (from 1-9) the highest count/frequency is determined, the word/s associated with that frequency are pulled out and these are combined to a Text value, with the key still defined as the reviewRating number.

Finally, these Text keys (reviewRating number from 1-9, one for each HashMap) and compound values (containing the word with highest frequency and its frequency) are written to a file as the output.

This second design seems more appropriate as instead of all the key-value pairs being passed from the mapper to the reducer/s, a local count is completed with the combiners and this condensed data is passed to the reducer/s, resulting in less data flow between the mappers and reducer/s.

Implement

On successful completion running the bespoke java MapReduce program with Hadoop, the following results were generated. The headings below describe the format of the results Hadoop returned where duplicate counts were returned for a reviewRating of 1 and 9, as per below.

Review_Rating (1-9): **Most_Frequent_Word/s|Frequency:**

```
1      character|movies|sagemiller|get|any|3
2      good|28
3      good|93
4      good|101
5      good|148
6      good|100
7      director|78
8      life|56
9      masterpiece|life|23
```