Computer Science
Big Data Computing
Homework

Dorjan Hitaj 1740478
Rexhina Blloshmi 1740477

September 2017

Preface

"Hiding within those mounds of data is knowledge that could change the life of a patient, or change the world."

– *Atul Butte, Stanford*

Contents

1 Abstract

Big Data computing is becoming more and more important nowadays. The amount of data produced each day is of tremendous amount, and this data has to be processed fast in order to get and understand the information it provides in a compact and useful way. Through years, as volume of data increased many ideas arose. One of them which gained a huge interest and use was Hadoop and its Map Reduce implementation. Map and Reduce programming paradigm and their implementation have been among us for many many years and their functionality can be implemented simply in a vast amount of programming languages. But this way of implementation did not bring many breakthroughs in the big data processing field. Hadoop changed this tremendously. The real key to success of the Hadoop MapReduce operation is in its implementation. The ability to divide every task into pieces and distribute them to multiple servers in a cluster for parallel computation is what made Map Reduce stand out in big data computing.

2 Introduction

In this homework we will try to exploit and see for ourselves the impact that Hadoop and its Map Reduce implementation give to Big Data computing. We are required to perform some analysis over a graph from the **Bilion Triples Dataset**. The dataset containing this graph is a Resource Description Framework dataset, which contains triples from the Semantic Web. The RDF data model is similar to classical conceptual modeling approaches (such as entity–relationship or class diagrams). It is based upon the idea of making statements about resources (in particular web resources) in the form of **subject–predicate–object** expressions, known as triples. The **subject** denotes the resource, and the **predicate** denotes traits or aspects of the resource, and expresses a relationship between the subject and the object.[1]

In this report we are going to describe how we implemented the tasks following the Map-Reduce paradigm, the results and statistics about them. Firstly we explain each of the tasks separately and their result when using 3 chunks from Billion Triples Dataset. More experiments will be shown in Run time analysis section and Scalability section where we experiment with more chunks of the dataset and in different machines.

3 Parsing the RDF dataset

Before starting with the analysis we had to parse the dataset. The parsing is done according to the format explained in Wikipedia, taking care of each possible case for the subject, object, predicate and context. Each line of the file has either the form of a comment or of a statement: A statement consists of three parts, separated by white-space:

- **Subjects** may take the form of a URI or a Blank node;
- **Predicates** must be a URI;
- **Objects** may be a URI, blank node or a literal.

where,

- The line is terminated with a full stop.
- URIs are delimited with less-than (<) and greater-than (>) signs used as angle brackets.
- Blank nodes are represented by an alphanumeric string, prefixed with an underscore and colon (_:).
- Literals are represented as printable ASCII strings (with backslash escapes), delimited with double-quote characters, and optionally suffixed with a language or datatype indicator. Language indicators are an @ sign followed by an RFC 3066 language tag; datatype indicators are a double-caret followed by a URI.
- Comments consist of a line beginning with a hash sign (#).[2]

The code we implemented to parse the file is done by taking into consideration each possible combination and following the rules stated above. This parsing is used only in the tasks where the input was the initial RDF dataset. In other tasks, whose input were files produced by other tasks run before, we have used other parsing methods.

4 Tasks

The tasks required to fulfill this homework can be divided into subsets of related tasks. By noticing this relation between them we were able to write MapReduce code that would solve many tasks in one run by chaining related jobs in such a way that, the output of one job would be the input of the other. In this way even on single node clusters we were able to save precious computation time and avoid repetition of mutual pre and postprocessing actions that were needed to obtain the desired results.

First four tasks are related to each other and thus we have implemented an intial preprocessing mapreduce job to obtain information that will be used as input to the four first tasks in the second round of mapreduce job. In the same round we perform several tasks that will be useful for Tasks 1, 2,3 and 4. For example on the same round we count distinct nodes, distinct edges and we also group the parts of the line into subject, object or predicate by concatenating the string **_subject**, **_object** and **_predicate** respectively, information which is needed to complete Tasks 1,2,3 and 4. It will be detailed in the following subsections.

Below in this section we are going to explain each task and the respective results for each of them. All the tasks in this section are performed over 3 chunks of the dataset on a single machine with the following parameters:

Machine	ASUS NV52
Type	Laptop
Processor	i7 6700HQ
Processor Count	4
Threads	8
Base Frequency	2.60Ghz
Turbo Frequency	3.50Ghz
Memory	16Gb
Memory Type	DDR4 2133MHz
Nodes	1

Task 1: Number of distinct nodes and edges

The first task is counting the number of distinct nodes and number of distinct edges. Distinct nodes sum up both the subjects and objects nodes since they are the nodes of the graph, while the edges sum up the number of predicates which are the edges in the graph. This task is completed in 1 round, thus using one map and one reduce function. In the map function, while parsing the data files, we *emit (node, one)* if the current part of line is subject or

object and we *emit (edge,one)* if the current part of line is predicate. Note here that **node** and **edge** are of type Text and **one** is a constant *IntWritable()* that is used to compute the number of occurrences of a node in the input. In the reduce function all the values associated to keys *node* or *egde* are summed up separately and written in the output file.

Experiments and Results

Here we show the results of the experiment with the first 3 chunks of data from the Bilion Triples Dataset.

- **Number of distinct nodes: 4617542**
- **Number of distinct edges: 10083**

So there are a lot more distinct nodes than distinct edges in the graph, about 460 times more, which means that a lot of the edges are common among different nodes. In the network there are about 10 000 distinct relations connecting the subject with the object.

Basically, let's suppose we have the subject node "Hamlet" and the object node "William Shakespeare", and the edge (predicate) connecting these two nodes "author". The edge "author" will be also the same for any book connected to a certain author. So the number of distinct nodes increases while the number of distinct edges would still be 1.

Task 2: Outdegree distribution

The outdegree of a vertex is the number of outward directed graph edges from that graph vertex. To compute the outdegree distribution we have to first group by subject, which as mentioned above, is done in the first MapReduce job. The outdegree of a node is how many times this node has been a subject in the triples of the dataset, and this is what the first reduce function does. So in the end of the first round we have a list of subject nodes with their outdegree.

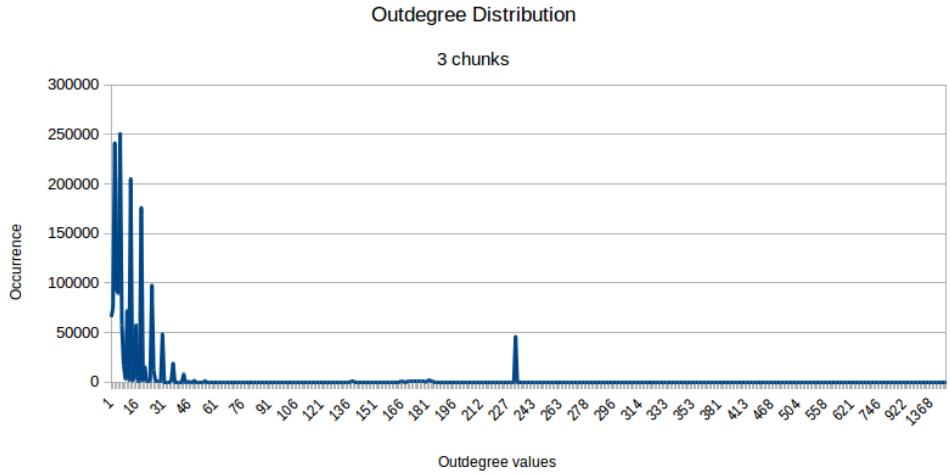
In the second round, in map function, to compute the outdegree distribution we consider only the lines in the output file of first round that contain **_subject** keyword which we intentionally set during the first phase to easily distinguish and count how many times a node has been a subject, thus obtaining its outdegree.

In this function they are grouped according to the outdegree. We *emit (outdegree, one)* in the map function while in the reduce function the values for each outdegree as a key are summed up and written in the output file. For example, we have a pair (32,107) which means that there are 107 nodes that

have a outdegree 32. As the outdegree distribution is written in the output file we plot the data in a graph using excel tools to see if the network is scale free, thus follows a power law distribution. As it will be shown plotted below, the outdegree distribution of our dataset indeed follows a power law distribution, where to the left there are the outdegree values that dominate.

Experiments and Results

The outdegree distribution of the graph follows a power law distribution as also shown in the plot below.



Task 3: Indegree distribution

The indegree of a vertex is the number of arcs leading to that vertex.

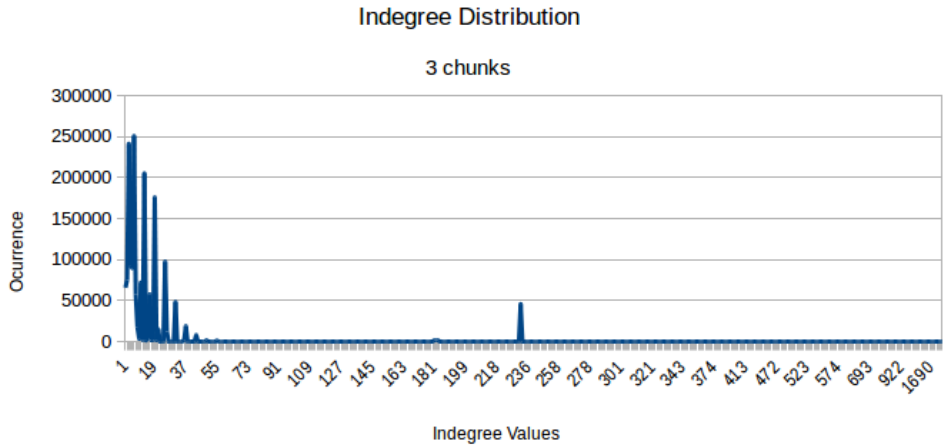
Similar to task 2, to compute the indegree distribution we have to first group by object, which is also done in the first mapreduce job. The indegree of a node is how many times this node has been an object in the triple and this is also done in the first reduce function. So in the end of the first round we have a list of object nodes with their indegree.

In the second round, in the map function to compute the indegree distribution we consider only the lines in the output file of the first round that contain **_object** keyword that we intentionally have appended during the first round. In this function they are grouped according to the indegree. We *emit(indegree,one)* in the map function while in the reduce function the values for each indegree as a key are summed up and written in the output file. For example, we have a pair (200,35) which means that there are 35 nodes that have a indegree 200. As the indegree distribution is written in the output file we plot the data in a graph using excel tools to see if the network is

scale free, thus follows a power law distribution. As it will be shown plotted below, the indegree distribution does follow a power law distribution, where to the left there are the indegree values that dominate.

Experiments and Results

The indegree distribution of the graph follows a power law distribution as also shown in the plot below.



Task 4: Maximum Outdegree nodes

To fetch the top 10 nodes with highest outdegree we have used only a **map** function without any reducer since the data was already computed in the previous job but needed only to be transformed. To obtain the 10 top nodes without going to the end of file we have overriden the ***compare()*** function from the **WritableComparator** class to sort the outdegrees in descending order instead of the **Hadoop** ascending default value used in the comparator class. In the map function which takes as input the file containing grouped nodes by subject, object or predicate as mentioned above, we consider only the lines containing **_subject** keyword and ***emit(subject, outdegree)*** which is then sorted by invoking compare function. This outputs a file containing all the subjects sorted in descending order by their respective outdegree. Note that this task does not need a reduce function since we are not reducing anything, but just sorting pre-computed data.

Experiments and Results

In the table below it is shown a list of the top 10 nodes having the highest outdegree and their correspoding outdegree.

Outdegree value	Node (subject)
12896	$< http://openean.kaufkauf.net/\text{id}/>$
6597	$< http://www.proteinontology.info/po.owl\#A >$
4380	$\_ : genid1xxhttpx3Ax2Fx2Fwwwx2Ecpantestersx2Eorgx2Fauthorx2FAx2FASCOPEx2Erss$
2546	$http://products.semweb.bestbuy.com/company.rdf\#BusinessEntity_BestBuy >$
1988	$\_ : genid1xxhttpx3Ax2Fx2Fblogspacex2Ecomx2Fswhackx2Fchatlogsx2F2002x2D01x2D16x2Erdf$
1690	$< http://purl.org/dc/terms/license >$
1690	$< http://purl.org/dc/elements/1.1/rights >$
1625	$< http://purl.uniprot.org/citations/15685292 >$
1508	$< http://sw.cyc.com/CycAnnotations\_v1\#externalID >$
1368	$< http://xmlns.com/foaf/0.1/Document >$

From this table we see that the node $< http://openean.kaufkauf.net/\text{id}/>$ has been a subject in the graph about 13 thousand times, which is approximately twice more than the node ordered second.

Task 5: Triples with empty context, blank subject and blank object

In the task five we have to compute the percentage of the triples that have empty context, blank subject or blank object. To obtain these results during the map phase we emit an IntWritable() one each time we encounter one of the above cases, where blank nodes are characterized by _: starting characters, while blank context is were the line does not have a fourth part other than subject, predicate and object. These are emmited related to a specific key, blank_context, blank_subject and blank_object respectively, and in the reduce phase all is left is to sum up those emmited counts related to their specific keys. In the end we obtain the number of triples that have an empty context, blank subject or blank object. Moreover to calculate the percentage we need the total number of triples in the dataset, value which is calculated by emmiting **one** for every line during the map phase associated to a single predefined key.

Experiments and Results

As for the percentage of the triples with empty context, blank subject or blank object it was interesting to see that in these chunks there was no triple with an empty context. While as for blank object or subject the percentages are as following. The total number of triples in first three chunks is **30000000**.

	Triples	**Percentage**
Total	30'000'000	100 %
Empty context	0	0 %
Blank subject	1'927'756	6.43%
Blank object	409'647	1.36 %

Task 6: Number of distinct contexts

Here we have to count how many times a triple occurs but the context of occurrence are different from one another. To find the number of distinct contexts we have considered the full triple as a key and the context that it occurs as a value. So in the map function we *emit(triple, context)*, where triple and context are both of type **Text**. In the reduce function for each triple there is an array of contexts in which the triple occurs. Note that in this array coming from map phase there might exist duplicated contexts which we don't count during the reduce as following. We populate another array for each triple with only the contexts appearing only once, and in the end the triple and the size of this new arraylist is written in the output file, which indicates the number of distinct contexts in which a certain triple appears. Since we want the top 10 triples that have the highest number of different contexts in the dataset we have implemented a mapping phase in which we override the default ***Comparator*** class method ***compare()*** to sort the triple, number of contexts according to the second parameter, thus number of contexts.

Experiments and Results

In the table below it is shown a list of the top 10 triples having the highest number of distinct contexts in which they occur and the corresponding number of contexts.

Distinct Contexts	Node	**Triple**
2199	Subject: Predicate: Object:	$< http://www.proteinontology.info/po.owl\#A >$ $< http://www.w3.org/1999/02/22-rdf-syntax-ns\#type >$ $< http://www.proteinontology.info/po.owl\#Chains >$
2199	Subject: Predicate: Object:	$< http://www.proteinontology.info/po.owl\#A >$ $< http://www.proteinontology.info/po.owl\#ChainName >$ $"ChainA"\|http://www.w3.org/2001/XMLSchema\#string >$
2199	Subject: Predicate: Object:	$< http://www.proteinontology.info/po.owl\#A >$ $< http://www.proteinontology.info/po.owl\#Chain >$ $"A"\|http://www.w3.org/2001/XMLSchema\#string >$
1690	Subject: Predicate: Object:	$< http://purl.org/dc/elements/1.1/rights >$ $< http://www.w3.org/1999/02/22-rdf-syntax-ns\#type >$ $< http://www.w3.org/2002/07/owl\#AnnotationProperty >$
1689	Subject: Predicate: Object:	$< http://purl.org/dc/terms/license >$ $< http://www.w3.org/1999/02/22-rdf-syntax-ns\#type >$ $< http://www.w3.org/2002/07/owl\#AnnotationProperty >$

1612	Subject: Predicate: Object:	$< http://openean.kaufkauf.net/id/ >$ $< http://www.w3.org/2002/07/owl\#imports >$ $< http://openean.kaufkauf.net/id/businessentities/ >$
1612	Subject: Predicate: Object:	$< http://openean.kaufkauf.net/id/ >$ $< http://www.w3.org/2002/07/owl\#imports >$ $< http://purl.org/goodrelations/v1 >$
1612	Subject: Predicate: Object: -	$< http://openean.kaufkauf.net/id/ >$ $< http://purl.org/dc/elements/1.1/rights >$ "This data set is available under the GNU Free Documentation License; see $http://www.gnu.org/copyleft/fdl.html$."@en
1612	Subject: Predicate: Object:	$< http://openean.kaufkauf.net/id/ >$ $< http://purl.org/dc/terms/license >$ $< http://www.gnu.org/copyleft/fdl.html >$
1612	Subject: Predicate: Object:	$< http://openean.kaufkauf.net/id/ >$ $< http://www.w3.org/1999/02/22 - rdf - syntax - ns\#type >$ $< http://www.w3.org/2002/07/owl\#Ontology >$

Task 7: Remove duplicate triples

Triples in the dataset are found in different contexts. As computed in the Task 6, there are triples that occur up 2000 times. This means that there are a lot of duplicate triples and removing those would shrink the dataset a lot. This is what Task 7 requires. We have to remove these duplicate triples and see the change in dataset size.

To complete this task we consider each triple as a key during the Map phase. This means that in the Reduce phase that key which is going to occur only once has solved our task which requires the same thing. Thus in the output file we write only the key and not the value, resulting in the desired output. In this step we can compare the sizes of the initial dataset and the dataset obtained by removing the duplicate triples.

Experiments and Results

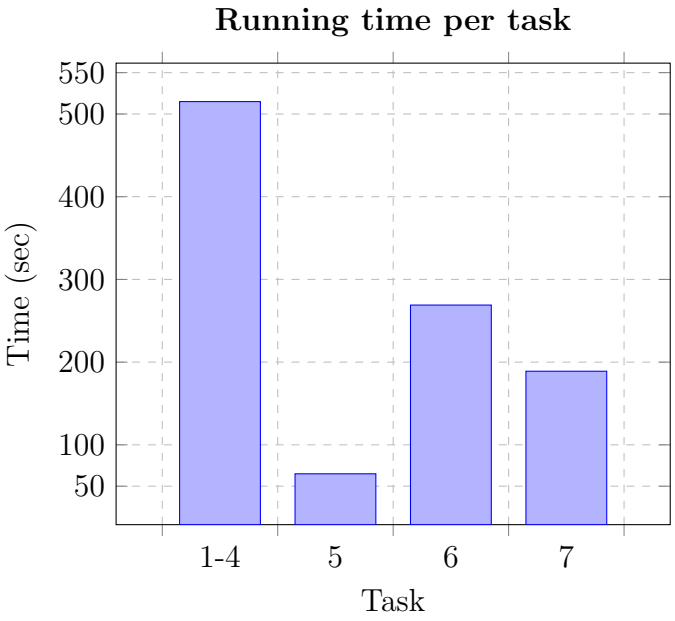
After removing the duplicate triples the size of the file reduced sharply. First 3 chunks of the dataset all together have a size of 6.6GB. The output file size is much smaller, just 1.6GB, thus more that 4 times smaller.

- **Original size: 6.6GB**
- **Reduced size: 1.6GB**

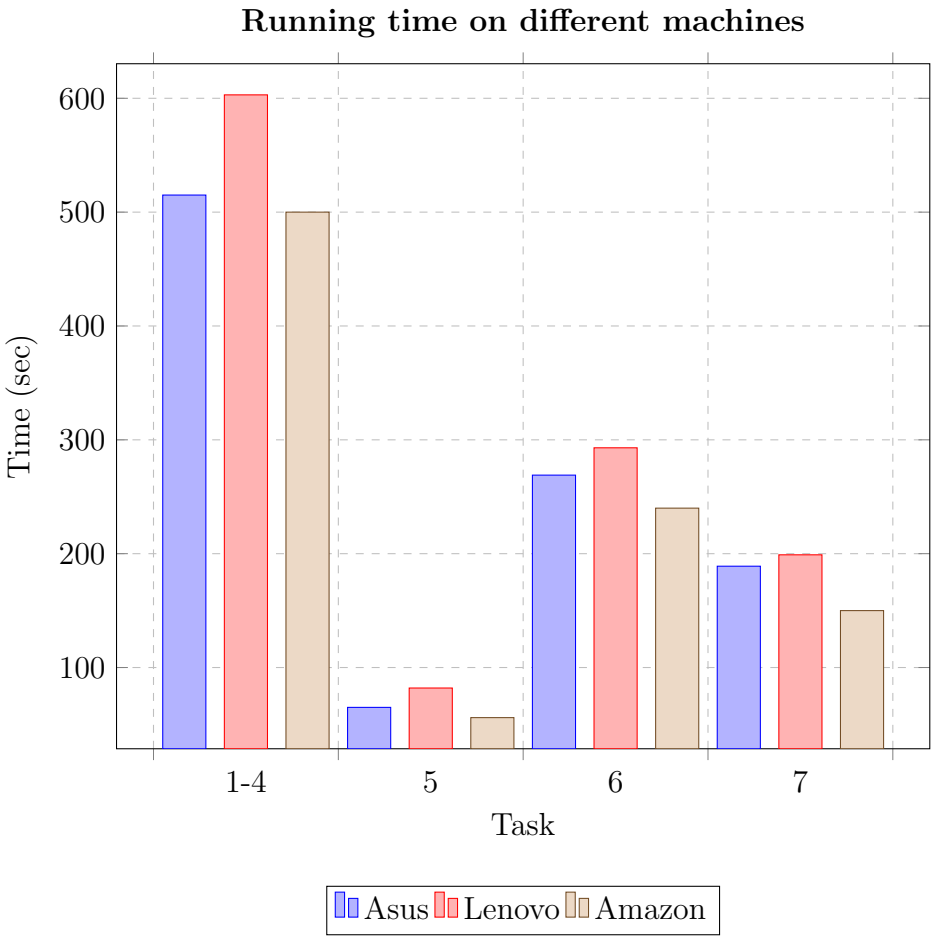
5 Running time analysis

In the tasks section we have not mentioned anything about time required by each task to be performed. Here in this section we are going to show running time, compare them among tasks and among machines used to run these tasks. First of all, as noted before, first four tasks are related to each other and they run in the same computation. So their running time is measured as one. Other tasks are run separately so the time reported are for each of them. As shown in the table below, first four tasks take more time than the others, about 8 minutes and 35 seconds. This is normal because these tasks include counting and sorting and 2 rounds each. Other tasks require less time, since they perform only one task each, where the task requiring more time is task 6, thus counting distinct contexts and sorting them to get top 10 triples with maximum number of distinct contexts which also requires two rounds. The task requiring less time is the task 5, which is performed in only 1 round counting the blank object, subject and empty context in triples of dataset.

Task	**Rounds per task**	**Time**
Task 1-4	2	8min 35sec
Task 5	1	1min 5sec
Task 6	2	4min 29sec
Task 7	1	3min 9sec



We also run the tasks in different machines, 2 laptops and 1 AWS cluster. They have different parameters as they are defined in the **Hardware Specification** Section below. The resulting time for each of the machines is as shown below for each of the tasks separately. The tests displayed here are on the first 3 chunks of the dataset.

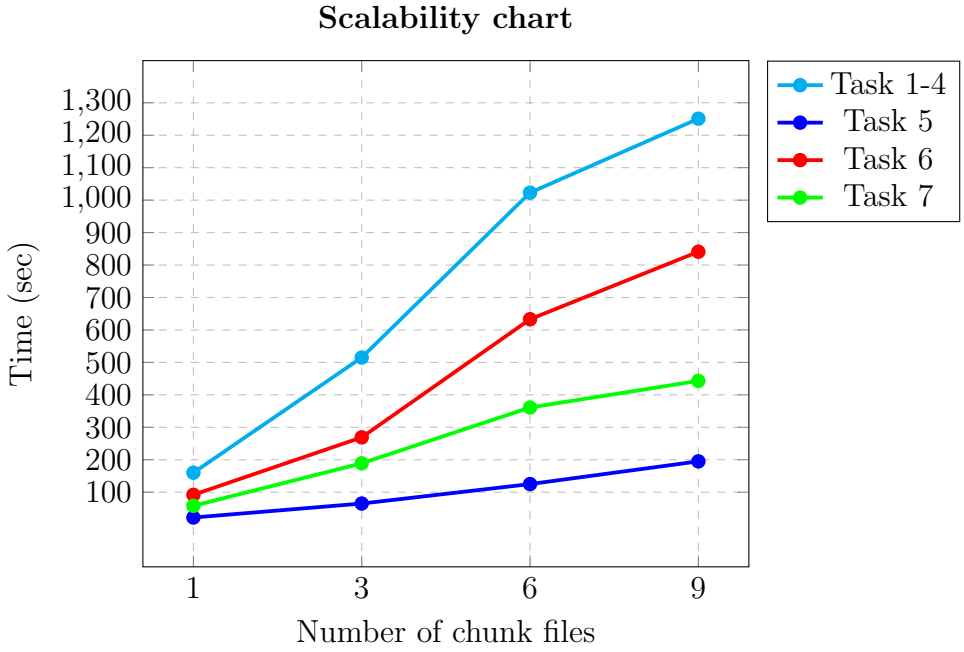


Amazon Cluster

We ran the tasks also on a self built Amazon AWS cluster[3]. The machines composing the cluster were weaker in power compared to our laptops, so the cluster was performing almost like our laptops as can be seen from the graph, but as soon as we started to increase the number of files (k) the parallel computation done by the cluster began to gain advantage over our single node clusters on our machines even though it was composed of weaker machines. To be noted is that in the cluster there is needed one more extra job that will merge the output on the separate datanodes, time which is added to the overall computation time.

6 Scalability analysis

In this section we have performed some analysis on how the running time increases as the the number of input files increases. Moreover we comment some of the results of the tasks when running on different subsets of chunks. Note here that due to memory issues we have performed these analysis on up to 9 chunks (18GB) of dataset.



As expected the time required for running the implementations increases a lot as the number of chunks of dataset increases. Also from the chart above it can be seen that the Tasks from 1-4 are the ones requiring more time even when more chunks are considered. As for the single tasks, Task 6 requires more time required to others since it consists of counting jobs and also sorting jobs.

An interesting observation that comes out while checking the output files is that the results seem to be consistent as the number of chunks increases. For example, in the task 4, where we list the nodes having maximum outdegree, the node ranked first when considering 1 chunk, is again that same node ranked first when considering 2, 3, 4 and so on, as number of chunks increases, just it's outdegree value increases. Also in the task number 6, where sorting the triples by the number of distinct contexts where they appear, we observe a similar behaviour.

7 Instructions on the code

The section below explains how to run the code for each of the required tasks on a single node configuration on a local machine. Single node installation tutorial can be found at.[4]

Depending on your hadoop installation directory and configurations some of the initial commands may vary, please change when needed.

- Open terminal by pressing: **Ctrl+Alt+T**
- Change directory to the directory of Hadoop installation and Start Hadoop: Under **sbin/** directory run ***./start-all.sh***
- Check if **Datanode** and **Namenode** are properly initialized by typing: ***jps*** on the terminal
  - If the **Datanode** is not showing then you should stop Hadoop via command **./stop-all.sh**
  - After stopping run: ***hadoop namenode -format***
  - After format is successful restart hadoop with the command written above.
- Create a directory in the hadoop distributed file system where you will put the ***Billion triples dataset chunks*** that will be used to perform computation:
  - Type in terminal **hadoop dfs -mkdir -p *inputDirectory/***
- Copy the files in the newly created directory by running:
  - **hadoop dfs -copyFromLocal /sourceDirectory /inputDirectory/**
    - **sourceDirectory** – Is the directory where the input files reside on your machine file system
    - **inputDirectory** – Is the distributed file system directory where you want to put the files
- In order to run the code by using the jar provided you need to:
  - To perform the tasks from 1 to 4 run:

* **hadoop jar** ***/path_to_project_files/BDC17-SourceCode-DorjanHitaj-RexhinaBlloshmi/Task_1-4/task1-4.jar /inputDirectory /outputDirectory1 /outputDirectory2 /outputDirectory3***

  Note here that there are 3 output directories, where one of them is the file containing the nodes grouped by subject, object and predicate, the second one is the file containing the indegree and outdegree distributions and the third one is the file containing the outdegree values for each subject node sorted descending from which we extract top 10 nodes having the maximum outdegree.

- To perform the task 5 run:
  * **hadoop jar** ***/path_to_project_files/BDC17-SourceCode-DorjanHitaj-RexhinaBlloshmi/Task_5/task5.jar /inputDirectory /outputDirectory1***

  In this task there is only one output directory which contains the file with the information about blank subjects, object or context.

- To perform the task 6 run:
  * **hadoop jar** ***/path_to_project_files/BDC17-SourceCode-DorjanHitaj-RexhinaBlloshmi/Task_6/task6.jar /inputDirectory /outputDirectory1 /outputDirectory2***

  This task has 2 output directories, where the first one contains the files with triples and the number of distinct contexts in which each triple appears, while the second one contains the file with ordered triples by the number of distinct contexts from which we extract the top 10 triples having the maximum number of distinct contexts.

- To perform the task 7 run:
  * **hadoop jar** ***/path_to_project_files/BDC17-SourceCode-DorjanHitaj-RexhinaBlloshmi/Task_7/task7.jar /inputDirectory /outputDirectory1***

  In this task there is only one output directory which contains the reduced file after removing duplicate triples.

To browse the hadoop distributed file system, Hadoop comes with also a web interface which by default is accessible on the url: **http://localhost:50070**. There you can check system information and download and see all the files that are in the distributed file system.

8 Hardware configuration

To solve this homework we used 3 different types of computational resources. In all these machines we installed Apache Hadoop version 2.7.3[5]. Two of them were our personal laptops, which are very different in both processor and memory frequencies and a self made cluster on Amazon AWS by using 4 t2.micro ubuntu machines in the configuration 1-Namenode and 3-Datanodes as in the tutorial prepared by Chris Dyck[3]. Since this was free we had the possibility to freely experiment with hadoop. The drawback of this was that the machines were weak. We tried also the Amazon EMR for a short period of time, but due to costs we decided to build our own cluster on the free machines that Amazon provides in its 12 month free tier usage. Amazon EMR is really straight forward and user friendly. You basically run everything on the web interface without any concerns and just get the result in the destinated EMR-Bucket. On the other hand setting up the cluster by ourselves we encountered a lot of challenges but we also learned a lot on how a cluster can be done, on a set of servers by taking care of all the network permissions and topolgy set up needed to run Hadoop on a distributed environment as it is destined to. The machines used on the cluster setup on Amazon were weak in performance compared to our machines in the processor count and memory frequency and size, but as the number of input files increased the cluster showed the importance of running jobs in parallel even though on weaker machines.

The code written for solving the above tasks was run on:

Machine	ASUS NV52	Lenovo X1 Carbon	AWS
Type	Laptop	Laptop	Cluster
Processor	i7 6700HQ	i7 5500u	XEON (unknown serial)
Processor Count	4	2	1
Threads	8	4	1
Base Frequency	2.60Ghz	2.40Ghz	2.50Ghz
Turbo Frequency	3.50Ghz	3.00Ghz	2.70Ghz
Memory	16Gb	8Gb	1Gb
Memory Type	DDR4 2133MHz	DDR3 1600MHz	DDR4
Nodes(N-D)*	1-1	1-1	1-3

*N - Number of Namenodes, D - Number of Datanodes

9 Conclusion

In this homework we had to perform statistical computations on the **Billion Triples Dataset**, by means of **Hadoop** and its **Map-Reduce** implementation. Large size of the input files and scarse computational resources made us understand and evaluate the importance of the parallelism that Hadoop makes possible. No matter how powerful can a single machine be, it can not match the combined power of a cluster running in parallel when it comes to Big Data processing. We wrote Map-Reduce code to solve each of the tasks by reducing also the number of jobs needed where possible, by chaining related tasks in such a way that the input of a task was the output of a task before it. In this way we decreased the number of jobs needed if we would have solved each task separately, but also the overall time to complete all of the 7 tasks required by this homework. We performed also scalability analysis on our code by increasing the size of the input dataset. We ran those tests on single node clusters on our personal machines and also on a mini-amazon cluster. As size of the files increased, the more was obvious the need for job parallelism when dealing with **Big Data**, and Hadoop makes this possible in a clean and comfortable way.

References

[1] Wikipedia. Resource description framework.

[2] Wikipedia. N-triples.

[3] Chris Dyck. Our own hadoop installation on amazon aws.

[4] Rajan Kumar. Single node hadoop installation.

[5] Apache Software Foundation. Apache hadoop documentation.