Distributed Data Deduplication using Spark framework

Haseeb Jadoon[1], Hammad Malik[2], Waleed Shabbir[3]
[1] Universität Bonn, Germany
S6hajado@uni-bonn.de
[2] Universität Bonn, Germany
S6hamali@uni-bonn.de
[3] Universität Bonn, Germany
S6washab@uni-bonn.de

Abstract

This report is done as part of the Distributed Big Data Analytics Lab during the Winter Semester, 2018-2019 at the University of Bonn. Data deduplication refers to a technique for eliminating redundant data in a data set. The main aim of this project is Data Deduplication from Data represented in RDF(s) implemented in Scala using the Spark framework.

1. Problem Definition

In the process of deduplication, extra copies of the same data are deleted, leaving only one copy to be stored - [Wikipedia].

Data deduplication is a complex problem. There are many traditional Algorithms for Data Deduplication. However, the XML Data is different from normal Data sets. Here we are dealing with records that can be of varying lengths. The complexity of the problem is increased because of the structure of the Data set.

In order to find duplicates, the whole tuple has to be compared. To avoid increasing the complexity, we used the approach mentioned in the research paper "Distributed Data Deduplication", to distribute similar tuples in our Dataset into Blocks and performing comparison inside the blocks. The block can be achieved by using Hashing on a key. The same key will be used to compare the records inside a block. However, distribution into blocks in still a time costly operation on large datasets.

In this paper, we will discuss our approach to solve the Data duplication problem on our Data using the Spark framework.

2. Approach

To solve the de-duplication problem, we researched a number of ways to handle this problem. After discussions and carefully analysing them we have implemented two of the approaches in our code. Here we will mention some which we rejected with its argument and then state the ones we implemented in our code.

2.1. Shared Database

We considered using an additional (key, value) Database i.e. Hbase, Aerospike and modifying Spark distribution algorithm to distribute the data evenly between all the nodes.

Let us take an example here.

We have 4 worker nodes, including the master. And we have 1000 records. The spark master will distribute data evenly between all of the workers, meaning 250 records each. Every node will convert their part of the data into (key, value) pair and query the database to see if the record is there, pseudo code below

```
For all the keys x in the dataset
    getRecordFromDatabase using key
    if key doesn't exist
        Insert key, value into DB.
    Else
        Do nothing
```

At the end, the master will export the Database records to our output location.

Using this approach, the data has to be traversed only once which means the complexity of our algorithm is O(n) for n= number of records.

In order to improve the time complexity, we are compromising the space complexity and the additional cost for the Database.

Looking at our helping research paper we found that they are using a shared-nothing architecture to achieve there desired solution so we switched to the other approaches.

2.2. Naïve-Data deduplication

The first approach is Naïve Data-deduplication. In this approach we leave everything on spark. Spark distributes the data itself. We used a spark RDD “distinct” function to get the job done and remove similarities from the Rdd. We will discuss more in the Implementation.

2.3. Attribute Based Partitioning deduplication

This approach is similar to the Naïve-Data deduplication. In this approach, we are using an attribute from our record to generate a Hash code for partitioning instead of using the whole record. We partition on the base of Hash code by the attribute. In each partition, we match the hash code of the attribute, if they are similar we compare the whole record otherwise we consider it to be unique.

2.4. Dis-deduplication (Blocking & Custom Partitioning)

This approach is similar to the once discussed in the research paper. We divide this approach into three parts; Pre-processing, Dis-partitioning, dis-dedup.

In the pre-processing, we use the word Count Algorithm to find count for our selected attribute. Calculate the threshold of every worker and create a mapping of data to be distributed.

In the partitioning phase, we used the map from pre-processing to distribute the data amongst workers.

In the Main Algorithm, we perform the data deduplication on every worker and combined the result.

2.5. Data Structures

RDD.

Hash Maps

2.6. Data Set

Data Pre-processing is the initial step in the deduplication process. Dataset have attributes i.e. names, tags, summary etc. It is important to make sure that the dataset is properly standardized, which means that the attributes which are going to be used for comparison of the data records are in one standard format.

The steps involved in data standardization are the removal of unwanted characters, meta-data, empty lines, or some unwanted spaces, special characters etc. Each attribute needs to be cleared.

The Datasets used in the reference research paper were publication records from CITESEER, called CSX and a private Dataset OA. We tried to get the same data but they weren’t available. So we generated our own standardized Dataset.

3. Implementation

3.1. Naïve-Data deduplication

The first approach is Naïve Data-deduplication. In this approach we leave everything on spark.

Spark distributes the data itself. We used a spark RDD “distinct” function to get the job done and remove similarities from the Rdd.

We created an RDD[String] and used the spark practitioner to distribute the data amongst the workers. The spark distribution mechanism uses hash codes to distribute the data. It calculates the hash code of every record and assign the same hash codes to same worker for performance optimization. After distribution we used the spark RDD. Distinct function to get the unique records.

```
val deDuplicatedRecords = textFile.flatMap(
        record =>{
                record.split("\n")
        }
).distinct()
```

The spark partitions algorithm works optimally on Dataset with all unique records and evenly redundant records.

Example

> We have 100 records, w-25 times, x-25times, y-25 times and z-25 times. We have 4 workers including the master. Every node processes and compares 25 records. This is the ideal case which reduces our time complexity by n/4.
>
> Consider, the case where the data is skewed. Once worker will get most of the data and the total complexity of our algorithm will be the time complexity of that worker.

Example

> We have 100 records, x-95, and y-5. We have two workers including master. In this 95 records will be given to worker-1 and the total complexity will n. Take this example as a big data case.

3.2. Attribute Based Partitioning deduplication

This approach is similar to the Naïve-Data deduplication. In this approach, we analyse our data and choose an interesting attribute which doesn't show the characteristics of data being skewed. After this we are pass this attribute to the spark to partition on the base of this attribute.

We override the equal() and getHashcode() methods. On every worker we performed the comparison first between the hash codes of the attribute instead of comparing the whole record. If the attribute hash code matches, we compare the whole records afterwards.

By following this approach, we were able to minimize the comparison time by a value x and avoid the data being skewed case to some extent.

3.3. Dis-deduplication (Blocking & Custom Partitioning)

This approach is similar to the once discussed in the research paper. We divide this approach into three parts; Pre-processing, Dis-partitioning, dis-dedup.

In Pre-processing, we get the total number of records 'n' and workers 'w' and calculate a threshold '$\boldsymbol{\theta}$'.

$$\boldsymbol{\theta = n/w}$$

We then use a simple word Count like blocking function to get the total number of occurrences of our selected attribute, sort them, and create a hash map of our workers and the selected attribute by comparing the attribute count with threshold.

In Dis-partitioning, we are using the hasp map from the pre-processing to create the partitions for the worker. We assign the records in a round robin fashion.

In the dis-duplication part, we are using the same approach as attribute based partitioning deduplication.

4. Evaluation

Below is the performance on different approaches with Different datasets.

Approach	**Time (milliseconds)**
Dis-duplication using custom practitioner	6375
Naïve-deduplication	5127
Attribute based partition deduplication	25016

Table 1 Dataset with 0.4 million records

Approach	**Time (milliseconds)**
Dis-duplication using custom practitioner	11787
Naïve-deduplication	9792
Attribute based partition deduplication	197403

Table 2 Dataset with 1.4 million records

Approach	**Time (milliseconds)**
Naïve deduplication	3903
Dis-duplication using custom practitioner	6119

Table 3 Dataset with 0.4 million records, 0.32 million skewed data

5. Installation and Usage Instructions

- Setup the environment (Install Java, Maven, Spark, Hadoop) using the Instructions provided in the Distributed Big Data Lab Tutorial 1.
- Download and Install Scala IDE for Eclipse.
- Import the code from git hub on local disk using the instructions mentioned at Atlassian Support.
- Import the project into Eclipse.
- Download the dataset.
- Configure the Environment.
- Run as Scala Application.

6. Future Work

In this Lab, we studied the implemented the problem of performing data de-duplication using Spark framework. We tried to minimize both communication cost by shared-nothing environment and computational cost using blocking and custom partitioning. Our future work includes research on data deduplication models in shared nothing environments, and Spark custom partitioning to improve the techniques we used during our Algorithm Implementation. We are going to research and implement different blocking functions.

We are also going to analyse and research on the triangle-deduplication technique using self-joining mentioned in the reference research paper and implement it to improve the computational cost even more, but it is currently dependent on better understanding of Spark and Custom partitioning.

7. Project Time Line

WEEK/ DATE	DESCRIPTION
1 (8TH NOVEMBER)	Project Assignment. *Reading & Understanding the Research paper.*
2 (15TH NOVEMBER)	Project Proposal *Understanding the Research paper.*
3 (21ST NOVEMBER)	First Meeting/Presentation of the Project *Project Presentation, Feedback Analysis.*
4 (29TH NOVEMBER)	R&D on Proposal Feedback *Discussion on Possible Approaches to the problem.*
5-6 (6TH DECEMBER)	Planning and Research *Understanding Spark.* *Research on Spark Architecture.*
7 (20TH DECEMBER)	Research on Dedup Techniques in paper using Spark *Blocking, Blocking functions.* *- Custom Partitioning.*
8 (2TH DECEMBER)	Naïve Deduplication Implementation *Sample Data Generation.* *Naive de-dup using spark.*
9-10 (3RD JANUARY)	Attribute Based Partitioning Deduplication *-Attribute based partitioning* *-R&D on Custom Partitioning in Spark.*
11 (17TH JANUARY)	Data Set Request and Generation *Request CITESEERX for data* *Request Professor for data* *Data Generation*
12 (24TH JANUARY)	2nd Project Meeting *Discussion, Code review, Feedback* *Working on Feedback*
13 (31ST JANUARY)	Dis-dup Pre-processing *R&D* *Pre-processing, Blocking function, Hash map*
13 (7TH FEBRUARY)	Main Algorithm *R&D* *Implementation*
14 (14TH FEBRUARY)	Evaluation & Code Improvement *Evaluation + Comparison* *Code Review, Bug fixing, Code Commenting*
14-15 (16TH FEBRUARY)	Project Report
16 (27TH FEBRUARY)	Final Presentation

Table 4 Weekly Project Timeline

8. References

[1] Spark 2.4.0 ScalaDoc [Internet]. Spark.apache.org2019
https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.Partitioner

[2] Distributed Data Deduplication [Xu Chu, Ihab F. Ilyas and Paraschos Koutris]
https://cs.uwaterloo.ca/~x4chu/CS-2016-02.pdf

[3] Partioning in Spark -Writing a custom partitioner
http://sparkdatasourceapi.blogspot.com/2016/10/patitioning-in-spark-writing-custom.html

[4] Spark Custom partioner [Pravin Boddu]
https://labs.criteo.com/2018/06/spark-custom-partitioner

[5] Data Duplication
https://en.wikipedia.org/wiki/Data_deduplication