Master’s Thesis

Performance Comparison of Apache Spark and Tez for Entity Resolution

Author:
Ikram Ul Haq

September 27, 2017

Advisors:
Dr.-Ing. Eike Schallehn
Institute of Technical and Business Information Systems

M.Sc. Xiao Chen
Institute of Technical and Business Information Systems
Ul Haq, Ikram:
Performance Comparison of Apache Spark and Tez for Entity Resolution
Abstract

Entity Resolution is among the hottest topics in the field of Big data. It finds duplicates in datasets, which actually belong to same entity in the real world. Algorithms that perform Entity Resolution are computation intensive and consume a lot of time especially for large datasets. A lot of research has been conducted for improving Entity Resolution solutions. A number of algorithms are developed, in attempt to reduce the time required to execute Entity Resolution algorithms on a given dataset. Efficiency of Entity Resolution algorithms has significantly improved but is still not adequate for large datasets in the Big data field. We are contributing to enhance its performance in terms of time, not by improving the algorithm but finding the most suitable platform on which it runs. This would, in turn, increase its efficiency and indirectly elevate the accuracy of Entity Resolution by empowering it to run more computation intensive algorithm. We have shortlisted Apache Spark (RDD, DataFrame and Dataset) and Apache Tez (Hive) as the set of platforms. In this research work we have chosen the Blocking technique for implementing Entity Resolution in the four above mentioned different applications. We have performed a number of experiments with different configurations to find the most efficient platform by analyzing, comparing and evaluating the results in great detail.
Acknowledgements

First and foremost I would like to thank my university advisor Xiaoy Chen, for continuously guiding me through the practical implementation guidelines as well a vast amount of suggestions while writing this thesis. She was the one who helped me choose this topic in the first place. I thank her for patiently handling the problems and doubts that I had. I would also like to thank my professors Eike Schallehn and Myra Spiliopoulou for agreeing to judge my semester’s work.

Secondly I’d like to thank all the technical teams behind the softwares, websites and frameworks I used (Hortonworks, Google, Stack Overflow and some more), for making them open-source and providing a good documentation. Needless to say, without them, this thesis would not have existed the way it does today.

I would also take this opportunity to thank the institutions Sofwer House(Pakistan), Brokergenius(Spain) and of course my university Otto von Guericke University (Germany) for introducing me to big and beautiful world of Big Data and letting me work in it.

Last but not the least, I express my gratitude to my family and close friends, who supported me emotionally and listened to me attentively while I shared my thesis’ status and anxiety with them. They not only were good listeners, but also helped me manage my time better and provided support in several different ways.

This thesis was made possible by the support, suggestions, recommendations and advice of all those mentioned above and once again I thank them all for being a part of my thesis and also my life.
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Magdeburg, August 27th 2017

Ikram Ul Haq
Contents

List of Figures ................................. xiv
List of Tables ................................ xv
List of Code Listings ......................... xvii

1 Introduction ................................. 1
  1.1 Motivation ................................. 1
  1.2 Goals .................................... 3
    1.2.1 Research Question ...................... 3
    1.2.2 Hypothesis ............................ 3
  1.3 Structure of Thesis ....................... 4

2 Background ................................. 5
  2.1 Entity Resolution ......................... 5
    2.1.1 Applications of Entity Resolution .... 6
    2.1.2 Process of Entity Resolution .......... 7
      2.1.2.1 Pre-processing ..................... 8
      2.1.2.2 Blocking .......................... 8
      2.1.2.3 Pair-wise Comparison ............... 10
      2.1.2.4 Classification ...................... 11
      2.1.2.5 Evaluation ........................ 12
  2.2 Computation Engines ...................... 14
    2.2.1 MapReduce ............................ 14
    2.2.2 Tez ................................ 15
      2.2.2.1 Comparison of Tez and MapReduce .. 16
      2.2.2.2 Hive ............................... 16
    2.2.3 Apache Spark ......................... 17
      2.2.3.1 Capabilities of Spark .............. 19
      2.2.3.2 Spark Core ........................ 20
      2.2.3.3 Spark SQL .......................... 22
      2.2.3.4 Spark Streaming .................... 25
      2.2.3.5 Spark MLlib ........................ 25
      2.2.3.6 Spark GraphX ....................... 25
3 Related Work

3.1 Improving ER Algorithms ............................................ 27
  3.1.1 Query-time Entity Resolution ................................. 27
  3.1.2 Feature Selection for ER .................................. 28
  3.1.3 Hybrid Human-Machine ER ................................. 28

3.2 Improving Parallel Computation of ER ............................ 28
  3.2.1 Set-Similarity Joins .................................... 29
  3.2.2 Dedoop – Deduplication with Hadoop ..................... 29

3.3 Evaluating Hive and Spark SQL .................................. 29

4 Methodology and Implementation .................................. 31

4.1 Theoretical Comparative Analysis of execution engines .......... 31
  4.1.1 Apache Spark vs. Apache Tez .............................. 31
  4.1.2 Apache Spark SQL vs. Hive QL .............................. 32
  4.1.3 Theoretical conclusion .................................. 32

4.2 Design for Experimental Comparative Analysis .................. 33
  4.2.1 Target of Experiments ................................ 33
  4.2.2 Steps to achieve the goal ................................ 34

4.3 Implementation .................................................. 35
  4.3.1 Architecture ............................................... 35
  4.3.2 Algorithms ............................................... 36
    4.3.2.1 RDD ............................................. 36
    4.3.2.2 DataFrame and Dataset ............................. 37
    4.3.2.3 Hive ........................................... 38
  4.3.3 Key Generation process .................................. 38
  4.3.4 Scoring function ........................................ 39
  4.3.5 Thresholding ............................................ 41

5 Experiments ....................................................... 43

5.1 Experimental Setup .............................................. 43
  5.1.1 Environment ............................................. 43
  5.1.2 Datasets ................................................ 44
  5.1.3 Eligibility criteria .................................... 45
  5.1.4 Procedure ............................................. 46

5.2 Experiment Results .............................................. 47
  5.2.1 Performance stimuli independent of platform ............. 47
    5.2.1.1 Direct impact of number of executors on performance . 47
    5.2.1.2 Direct impact of input data on performance ........ 49
  5.2.2 Comparison among Apache Spark APIs ..................... 50
    5.2.2.1 Influence of dataset size .......................... 51
    5.2.2.2 Influence of number of executors .................. 54
    5.2.2.3 Evaluation ....................................... 56
    5.2.2.4 Result of comparisons .............................. 58
  5.2.3 Spark vs. Hive ........................................... 58
5.2.3.1 Speed comparison .............................. 59
5.2.3.2 Throughput comparison ....................... 62

6 Conclusion ............................................. 65

7 Future work ........................................... 67

Bibliography ............................................ 69
# List of Figures

1.1 Google Trends - Apache Hive, Apache Tez and Apache Spark ........ 2

2.1 Process of Entity Resolution ........................................... 7

2.2 MapReduce Processing Model ........................................... 14

2.3 Hadoop’s Components ................................................... 15

2.4 Tez Processing Model ................................................... 16

2.5 Hive’s Architecture ..................................................... 18

2.6 Apache spark Ecosystem ................................................. 19

2.7 Evaluating runtime of aggregation query via DataFrame API and RDD with different programming languages ............................... 23

2.8 Apache Spark components with DataFrame API ...................... 24

4.1 A comparison of Apache Spark, Apache Hive and Hive on Tez on Google Trends ............................................................... 32

4.2 Architecture of our application ......................................... 35

4.3 Algorithm for ER processing using RDD ............................ 36

4.4 ER processing algorithm for DataFrame and Dataset .......... 37

4.5 ER processing algorithm for Hive ...................................... 38

4.6 Process of key generation ................................................ 39

4.7 Algorithm for calculating the final score ........................... 40

4.8 Filtering results to identify matches .................................... 41

5.1 Structure of HDP [1] ....................................................... 44

5.2 Case1: No shuffling ...................................................... 48
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Case 2: Shuffling required</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Case 1: No shuffling</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Case 1: Shuffling required</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance of RDD on different datasets</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>Experiments ran for confirming no biases in cluster</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>Performance of DataFrame API on different datasets</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Performance of Dataset API on different datasets</td>
<td>54</td>
</tr>
<tr>
<td>5.10</td>
<td>Performance gain by increasing a number of executors for processing the small dataset</td>
<td>55</td>
</tr>
<tr>
<td>5.11</td>
<td>Performance gain by increasing number of executors for processing the medium dataset</td>
<td>55</td>
</tr>
<tr>
<td>5.12</td>
<td>Performance gain by increasing number of executors for processing the large dataset</td>
<td>56</td>
</tr>
<tr>
<td>5.13</td>
<td>Apache Spark APIs</td>
<td>57</td>
</tr>
<tr>
<td>5.14</td>
<td>Error recognition in Spark SQL, DataFrame API and Dataset API[2]</td>
<td>58</td>
</tr>
<tr>
<td>5.15</td>
<td>Time consumed for processing input datasets</td>
<td>59</td>
</tr>
<tr>
<td>5.16</td>
<td>Comparison of algorithms with(prev) and without writes outside the memory</td>
<td>60</td>
</tr>
<tr>
<td>5.17</td>
<td>Throughput of applications</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Table A ................................................................. 6
2.2 Table B ................................................................. 6
2.3 Difference between Apache Tez and MapReduce .................... 16

5.1 Specification of each node in YARN cluster. ......................... 44
5.2 Statistics of datasets ................................................. 44
5.3 Eligibility test for small dataset .................................... 45
5.4 Eligibility test for medium dataset .................................. 45
5.5 Eligibility test for large dataset .................................... 46
5.6 Configurations when Apache Spark APIs gave their best ........... 58
List of Code Listings

2.1 RDD code for reading text lines and breaking them into words . . . . . . 24
2.2 Dataset code for reading text lines and breaking them into words . . . 25
1. Introduction

Generally, distinct descriptions of a particular real-world entity may exist. Data formatting, acronyms, typographical errors, and so on are considered as the main reasons of distinct descriptions. Quality of the data reduces due to such kinds of descriptions plus they also bring about false interpretation to the user. Hence, Entity Resolution (ER) is carried out to solve this issue.

For both cleaning as well as integration of data, ER is considered as a key task in order to enhance data quality [3]. It has been analyzed that in the past few years, research analysts from diverse disciplines such as, machine learning, bio-informatics etc., have carried out different researches related to ER [4]. Its applications are for varied areas, such as public health data, web pages, shopping comparison, etc. [4].

For big data, ER process normally requires more than a few hours. With the intention of managing big data, emphasis of research is given towards the evolution of ER process in distributed environments. In such environments, application’s efficiency is enhanced and runtime is reduced. It has been observed that up to now, by novel approaches and algorithms, several parallel ER applications have been progressed. However, in contrast to parallelism approach based on MapReduce and other general parallel computing techniques, not much research is carried out in the domain of Apache Spark and Tez parallelism techniques.

1.1 Motivation

Real world entities (such as people, plants, nutrients in products etc.) need to be digitally recognized in order to gain the most from the data we have. Analysis is easier, faster and less error prone when done digitally, since manual analysis is more time consuming, prone to human errors and can get very confusing very fast (especially for a fast growing entity). We can perform various analyses that can benefit the world or make a process easier with this digital data. Its importance is thus very clear. However,
while creating these digital entities, the quality of data is compromised due to data formatting inconsistencies, acronyms, typographical errors and so on. The process that tries to solve this issue is called Entity Resolution (ER). ER enhances the data quality by cleaning, identifying and handling duplicates.

Real world entities, like population of a country alone, can easily be classified as Big Data. Naturally ER is a Big Data problem, which needs to be solved using Big Data tools and suites. Talking about Big Data tools, the question is which one of them should be used. We no longer live in a world where we only have one option to start a Big Data project. We have an abundance to choose from. The first natural instinct is to check the top n (usually 2-5) tools and then research them further to find which one of them suits you the most.

Hive, Spark and Tez are some examples of the most efficient ones today. While researching them we came across bold speed statements by Spark and Tez where both of these execution engines claim to be a 100 times faster than MapReduce\[5\] [6]. We then checked if these statements affect the popularity they gain and so we compared them on Google Trends.

![Figure 1.1: Google Trends - Apache Hive, Apache Tez and Apache Spark](image)

As seen in the Figure 1.1, Spark is more popular. But according to their claims, the performance of Apache spark and Apache Tez should have been the same. There could be three possibilities for popularity of Apache spark:

1. Data engineers trust the claim of Apache spark more than that Apache Tez for some reason.
2. Data engineers found that apache Tez is 100x faster only for that particular usecase on which they have performed benchmarking but not for their usecases.
3. Data engineers like the ease of the interface of Spark more than Tez. Also spark is a unified stack which falls in its favour.

As we are concerned about execution performance, point 2 is interesting in our research as opposed to the other two. Our usecase is ER and it is ambiguous which engine
will perform better. Because no comparison in this context had been done before i.e. between Apache spark and Tez for ER particularly, in this thesis we are comparing the performance of these two engines in great detail. Our intention is to improve efficiency of ER by choosing most suitable engine, but the best engine for ER can only be decided when each of them is practically tested.

1.2 Goals

The key goal of this thesis is to compare the computational performance of an ER application. For this, we design and implement a parallel ER application using Apache Spark (RDD, DataFrame and Dataset) and Tez (Hive) as the set of platforms; and also to assess an application’s performance via different methods by considering time as the main factor.

In order to attain our main goal, we carry out the following tasks:

- An ER application, comprising 5 chief steps of ER: pre-processing, blocking, pairwise comparison, classification and evaluation, is developed. By means of Apache Spark and Hive, all these steps are carried out.

- We develop user defined functions in order to categorize the records into matches and non-matches. Details of the algorithm are given in Chapter 4. Further, most of the computations are carried out in-memory that facilitates to run the application faster.

- We consider three different sizes of datasets: small dataset (105000 records with 5% duplicates), medium dataset (52500 records with 5% duplicates), and large dataset (1100000 records with 10% duplicates). Precision, recall and F-measure are considered with the aim of evaluating the worth of the results. In addition, runtime, efficiency and scale-out experiments are carried out to assess the application’s parallelism in the cluster.

1.2.1 Research Question

This research study addresses the following research question:

- Which application is faster i.e. Hive running on Tez or RDD, DataFrame or Dataset running on Apache Spark for computing Entity Resolution?

1.2.2 Hypothesis

Our proposed hypotheses is as follows:

- It is hypothesized that Hive running on Tez is faster than Apache Spark DataFrame, Apache Spark Dataset and Apache Spark RDD for computing Entity Resolution.
1.3 Structure of Thesis

This thesis has been organized in the following manner:

- **Chapter 1** provides an introduction of the entire thesis. It encompasses motivation behind this research, main goals of the research along with the research questions and hypothesis.

- In **Chapter 2**, we enlighten the background of this research. It holds an in-depth explanation of all the important topics that are vital to understand this thesis.

- **Chapter 3** provides a comprehensive view of the work related to our research domain. This chapter expresses the standpoints discussed in different papers, articles, journals and books published by different researchers and analysts specific to the analogous research domain.

- In **Chapter 4**, we present the implementation details of our application. It also includes methodology, application’s architecture, and algorithms.

- **Chapter 5** describes all the experiments along with its interpretation.

- **Chapter 6** concludes the thesis by mentioning our findings and implications derived from the experimental analysis.

- Our future intention in the similar area is shown in **Chapter 7**.
2. Background

In this chapter, an elucidation of all the significant topics which are imperative to understand this thesis is provided. Initially, the fundamentals of Entity Resolution (ER) are enlightened in Section 2.1 and within that, an in-depth knowledge of ER including its entire process is presented in Section 2.1.2. In Section 2.2.1, we give emphasis to MapReduce. The concepts of Apache Tez and Apache Hive are explained in Section 2.2.2 and Section 2.2.2.2 respectively in an unambiguous way. Then, we discuss Apache Spark comprehensively in Section 2.2 and draw attention to RDD, DataFrame, and Dataset.

2.1 Entity Resolution

Records from one or more sources that may characterize similar or identical real-world entities are categorized in the process of Entity Resolution (ER). Duplicate entities are those, which are similar or identical in the real-world [4]. In the usual courses of events, duplicates prevail in a number of areas of data records, for example, physical condition records of patients, ecological or geographic data, census data, market research data, organizational records, and so forth. Web-pages [7] and databases [8] are the two chief input sources of ER.

By means of other terminologies like de-duplication [9], detection of duplicate [10], fuzzy duplicates [8], record linkage [11], and merge problem [12], ER can be also described in the research society. An elucidation of ER specifically in the community of database is to categorize analogous or similar records from databases that signify identical entities of the real-world [13]. By means of “COUNT” aggregate function with “GROUP BY” keyword, it is easier to identify duplicates from a database if similar records exist. Nonetheless, chief causes of duplicates include acronyms, misspelled words, missing fields, and so on; this makes it quite difficult to detect them by means of a single query [14]. Structured, unstructured, and semi-structured data are the types of data and for ER, the input data possibly will exist in any of these types of data. Data format is
the basis on which the ER process relies. For that reason, for distinct data formats, a number of entity resolution techniques have been put forward [13] [14].

As an example, with the aim of understanding ER, two tables i.e. Table A and Table B are presented below. These tables have customers’ personal information associated with an organization. The considered fields are Customer ID, Name, City, and Birth Date; and it can be observed that both tables contain these five fields. However, it can be analyzed that Dates’ representation is different in both tables. Besides, in personal details, null values and errors are contained by some entries. On the other hand, it can be observed that X1 and Y1 talk about the same individual named Sarah and X4 and Y3 talk about the same individual named George. Understanding such diverse formats, inaccuracies, and so forth as well as finding that the record is associated with the same individual refer to the process of ER. The considered example is quite simple so as to clarify the idea behind ER and as the following tables contain only a few records, it is not difficult to analyze duplicates manually. Nevertheless, the scenario is very different in the real-world; practically one will observe that the data that involves duplicates not just of huge size but intricate as well. And in such data, manual analysis of duplicates is not possible.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>City</th>
<th>Date of Birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Srah</td>
<td>Muzaffarabad</td>
<td>08-12-1986</td>
</tr>
<tr>
<td>X2</td>
<td>Ahmed</td>
<td>Riyadh</td>
<td></td>
</tr>
<tr>
<td>X3</td>
<td>Ghazanfar</td>
<td>Berlin</td>
<td>12-06-1974</td>
</tr>
<tr>
<td>X4</td>
<td>Geore</td>
<td></td>
<td>03-03-1992</td>
</tr>
</tbody>
</table>

Table 2.1: Table A

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>City</th>
<th>Date of Birth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y1</td>
<td>Sarah</td>
<td>Mz</td>
<td>11/08/1993</td>
</tr>
<tr>
<td>Y2</td>
<td>Ayesha</td>
<td></td>
<td>20/05/1986</td>
</tr>
<tr>
<td>Y3</td>
<td>George</td>
<td>Istanbul</td>
<td>03/03/1992</td>
</tr>
</tbody>
</table>

Table 2.2: Table B

2.1.1 Applications of Entity Resolution

In this section, main applications where ER is used are discussed. The general applications of ER are as follows:

- Health data: ER software is employed by different hospitals to enhance the records’ quality in health care information system. The quality of health data can be enhanced by eliminating the duplicates in records, confining the gathering of duplicate information of the patients, incorporating previously gathered patient’s health data from different records, etc. [15].
2.1. Entity Resolution

- Census: In census records, usually ER approaches are used to merge the data gathered from various sources, to match new data with the formerly acquired information to form new records of the data [16].

- In addition, other applications where ER is applied include e-commerce, web information search, bibliographic databases, genome databases, etc. [4].

2.1.2 Process of Entity Resolution

A common process of ER is illustrated in Figure 2.1.

![Figure 2.1: Process of Entity Resolution](image)

Basically, ER consists of 5 main steps. Data pre-processing is the foremost step. After that, blocking, pair-wise comparison and classification are carried out; and finally, evaluation takes place [4]. Understanding all these steps from top to bottom is important in order to comprehend the entire process of ER. As illustrated in Figure 2.1, the input of a step is essentially the output of a step prior to it. In the pre-processing phase, data...
obtained from the data source is pre-processed. This foremost step of ER process is discussed in detail in Section 2.1.2.1. Prior to pair-wise comparison phase, blocking techniques are employed so as to prevent matching every single record with each other in the pair-wise comparison phase. In Section 2.1.2.2, the blocking phase is discussed comprehensively. In the pair-wise comparison phase, records in the dataset are contrasted with one another in order to determine the duplicates in the data. In Section 2.1.2.3, pairwise comparison phase is elucidated in detail. By means of classification techniques which are clarified in Section 2.1.2.4, the compared records are categorized as matches and non-matches. In the evaluation phase which is enlightened in Section 2.1.2.5, by means of particular strategies, the outcome of the classification step is evaluated.

2.1.2.1 Pre-processing

Data pre-processing, also referred to data preparation, is the preliminary phase of ER [17]. In this step, data is prepared in a particular way with the aim of progressing to the other steps of ER.

Moreover, for the ER process, redundant data is encompassed by the data sources from where the duplicates are found out. It is important to remove this redundant data otherwise it can possibly bring about computational operating cost and imprecise outcomes. Hence, so as to acquire better outcomes, this step is essential for enhancing the quality of data. The first step of ER process includes Methods of Extraction and Transformation and Loading (ETL) [18] and can be divided into the following three stages [17]:

- Parsing: In the data, if a field encompasses various words then it is required to locate the as separate words and process. This process is referred to as parsing.

- Stop words removal: Words like ‘is’, ‘the’ etc. are stop words. They are commonly occurred in a text and have no significance. As they contain no information, they are removed in this step.

- Data standardization: In this step, we focus on the standardization of schema as well as data format of the field.

2.1.2.2 Blocking

Blocking is also referred to as indexing [19] [20]. Several fields are contained by the data records and more or less, comparisons are carried out on the entire fields; and in ER, comparison is considered as a quite costly step. So, the chief aim of a blocking step is to cut down the comparisons’ number in between the records. For n records, the comparison step requires \(O(n)^2\) without blocking. For instance, if one has 50 records for ER process, 2500 comparisons come about with no blocking. In brief, cutting the runtime of the ER process and increasing the speed is the chief intention of the blocking step.
There are three blocking techniques which are generally used, namely; standard blocking, sorted neighborhood blocking, and q-gram based blocking [21]. These techniques are discussed below:

**a. Standard Blocking:** Standard blocking is considered as a conventional blocking approach. In this approach, records having identical blocking key values (BKV) are added into the similar block [4]. The records amid the identical block are contrasted to one another after adding into the similar block. If a single BKV is identified, every single record is added simply into one block. Uniform distribution of BKVs can be considered as a chief aspect on which the size of the generated block relies. Block’s size is determined by the frequency distribution of the BKVs. Now, let’s consider that the size of the block is uniformed as the distribution of BKV is uniformed. X and Y are the two data sources which are required to be matched. If |X| records are contained by X, |Y| records are contained by Y, and the total count of BKVs employed is represented by ‘z’ then the generated $U_{XY}$ for the record pairs for X and Y will be:

$$U_{XY} = \frac{|X||Y|}{z}$$

(2.1)

And the generated $U_X$ for matching within a particular data source will be:

$$U_X = \frac{|X|}{2} \times \left(\frac{|X|}{2} - 1\right)$$

(2.2)

The following paragraph elucidates the way of selecting blocking key values.

**Blocking Key Value:** By considering the quality, data completeness in attribute and the attribute’s frequency distribution, Blocking Key Value (BKV) is selected [4]. To get a BKV, two different functions which are applied to attributes in records are phonetic encoding and character-based encoding. The concatenation of these functions on attributes in the data can be selected as blocking keys. In addition, to get better records’ blocks, multiple blocking keys are employed. Examples having multiple blocking keys are given below:

- Soundex(FamilyName) + InitialFourDigits(CellNumber)
- DoubleMetaphone(InitialName) + Soundex(FamilyName)

**b. Sorted Neighborhood Blocking:** Duplicate records tend to come near to one another in the sorting data step and in the phase of merging data, they are contrasted with one another – this supposition forms the basis of Sorted Neighborhood. Prior to sorting, the data from various sources is merged in this approach. Additionally, sorting
key, sorting data, and merging are the three chief steps on the basis of which records in the data are assigned into different blocks [22].

c. **Q-gram based blocking:** In q-gram based indexing technique, the records with similar BKV will be matched just like the standard blocking technique [4]. Nonetheless, one dissimilarity exists which comes about when the BKV is generated. By taking several variants of strings by means of q-grams, BKV will be generated. Basically, the sub-strings of a string having length q represent the q-gram. Every single blocking key can be transformed to a q-grams’ list where an individual can state the maximum length of q in the string. The duplicated records hold greater chance to be contrasted with one another because there are multiple BKVs and this will generate better precision of outcomes. As compared to the formerly considered blocking techniques, time consumed will be more for huge datasets as a great deal of records have to be matched with one another. This aspect is considered as its limitation [23].

Apart from the aforementioned blocking techniques which are broadly used, more than a few other blocking techniques exist. Some of them include [23]:

- Suffix array based blocking
- Map based blocking
- Canopy clustering and so on

### 2.1.2.3 Pair-wise Comparison

Pair-wise comparison is the third step of ER process. It is also referred to as the record pair comparison step. In ER, it is required to compare the fields or strings and after that, we assess the correspondence/similarity between them. Techniques like similarity measures, also known as field matching techniques, are used to compare strings. Generally, because of typographical discrepancies, variations in the strings come about. Besides, there are various reasons of typographical discrepancies including missing and misplaced characters, typographical mistakes, missing fields and so forth. Sound-based similarity measures and character based similarity measures are the two major categories of similarity measures [17]; with the aim of calculating the similarity between records, any of these similarity measures can be considered in pair-wise comparison step. Mostly, a pair-wise comparison step employs character-based similarity measures whereas blocking techniques usually employ sound-based similarity measures. Following are the different types of sound and character-based similarity metrics:

- Sound-based similarity metrics (commonly used techniques consist of soundex, metaphone, and double metaphone)
- Character-based similarity metrics (commonly used techniques include Levenshtein edit distance, Hamming distance, Jaro and Winkler distance metric)
2.1. Entity Resolution

- Exact Similarity Metric

It should be noted that based on the algorithm that is used to contrast the strings, the values of the pair-wise comparison will be formed. The value of the outcome will show a discrepancy with a number of algorithms. The comparable fields have to be examined and appropriate algorithm must be selected on the basis of the data type.

2.1.2.4 Classification

Classification technique classifies the compared records in the pair-wise comparison step as matches and non-matches [24]. In order to classify the compared records, different classification techniques including threshold-based, rule-based, and learning-based classification are elucidated below.

a. Threshold-based Classification: A classification technique in which the summation of similarities between the entire attributes in pair-wise comparison is computed as TotalSimilarity [4] on which a similarity threshold is employed so that the class of both match and non-match is determined in accordance with the threshold provided. For example, TotalSimilarity is computed for \((hx; hy)\) – a record-pair which are contrasted. And let the similarity threshold employed to \((hx; hy)\) be \(z\). Following is the classification of record-pair:

\[
\text{Total Similarity}(hx; hy) \geq z \implies (hx; hy) \text{ is a match} \quad (2.3)
\]

\[
\text{Total Similarity}(hx; hy) < z \implies (hx; hy) \text{ is a non-match} \quad (2.4)
\]

The attributes are considered equivalent in threshold-based classification devoid of giving significance to the specific and distinct attributes. In order to prevail over such restriction, we can compute a weighted threshold via passing particular weight to every single attribute relying on their significance [4]. In this technique, complete information of the particular similarities of record pairs is mislaid. Moreover, this problem can be overcome by applying different classification techniques.

b. Rule-based Classification: Certain rules are required to be identified in rule-based classification technique. On the basis of the rule, the records will be categorized into matches and non-matches. Similarity values form the base of rule and these values are acquired by the record fields plus they are merged by means of conjunctions, disjunctions, and negations. Let \((hx; hy)\) be a record pair, \(D\) be the predicate on record pair, and \(R\) be the classification’s outcome. Afterward, \(D \implies R\) will be the form of the rules applied. In addition, learning from the training data or applying rules manually is possible [4].

c. Learning-based Classification: To classify the records, learning-based classification techniques from machine learning and data mining areas are put into effect [25].
Training data is provided and the algorithm learns from it. For the records, true match and non-match labels are contained by the training data. The identical values acquired from field comparison techniques are referred to as parameters or feature vectors in the domain of machine learning. From the training data, parameter/feature vectors are selected so that the classifier categorizes the record pairs as matches and non-matches in an accurate way.

The classifier’s accuracy is tested and set to get great accuracy prior to classify the other record pairs. Learning-based classification approaches which are usually applied are elucidated below:

1. Decision Tree:
   For supervised learning classification, decision tree classification is considered as one of the most regularly applied algorithm [25]. Decision tree holds top-down approach. It has three types of nodes i.e. root node, decision node and leaf node. At the root level, the training data is acquired. In decision nodes, by making specific decisions and rules, the attributes in the training data are split; plus in this same node, tests are carried out on attributes. With the exception of the root node, each node in the tree is the outcome of the prior node. Till there is no more decision to make and the entire attributes are split, splitting will keep happening by the decision nodes. If all attributes in it belong to the same class, splitting will be discontinued. When decision node carries out all the steps, leaf node along with the class label (match or non-match) is allocated to it [25].

2. Random Forest:
   An assortment of decision tree classifiers is referred as random forest [26]. If the grouping of classifiers increases, the classifier’s accuracy to forecast the classes surges. In random forest, creation of random subsets takes place by choosing arbitrary attributes from the training data. Multiple decision trees are created via these random subsets. These multiple decision trees are referred as random forests. Seeing that there are lots of decision trees to select expected labels in this assortment of decision tree classifiers, voting is carried out for every single decision tree and on the basis of the results of voting, the class of the record pair is anticipated.

   As there are many decision trees to choose predicted labels in the random forest, voting is conducted for each decision tree and the class of the record pair is predicted based on the results of voting.

2.1.2.5 Evaluation

In ER, the fifth as well as the last step is Evaluation. In this last step, we calculate the accuracy of the records which are categorized as matches and non-matches. So as to calculate the quality of acquired outcomes, evaluation measures are indispensable. The gold standard data of the records is required for the entire evaluation measures [4] and
by such type of data, the correctness of the classified records can be known. In addition
to this, such kind of data holds the info of the accurate matches and non-matches of
the records. On the basis of records’ classification, they can be categorized into the
following groups [27]:

- True Positives (TP): A true positive test outcome is the one that finds out the
  state when the state is present
- True Negatives (TN): A true negative test outcome is the one that does not find
  out the state when the state is absent
- False Positives (FP): A false positive test outcome is the one that finds out the
  state when the state is absent
- False Negatives (FN): A false negative test outcome is the one that does not find
  out the state when the state is present

A matrix which is referred to as confusion matrix contains values which are associated
with the aforementioned categories. Recall, precision and F-measure, are computed in
order to evaluate the records’ classification. Additionally, in information retrieval, these
metrics are extensively used [27].

a. **Precision:** the ratio of the number of relevant records retrieved to the total number
    of irrelevant and relevant records retrieved is called as precision. It is also referred to as
    positive predictive value [27]. It is computed as:

\[
\text{Precision} = \frac{TP}{TP + FP}
\]  

(2.5)

b. **Recall:** The ratio of the number of relevant records retrieved to the total number of
    relevant records in the database is called as Recall. It is also referred to as true positive
    rate [27].

\[
\text{Recall} = \frac{TP}{TP + FN}
\]  

(2.6)

c. **F-measure:** The harmonic mean (H.M) of precision and recall is known as F-
    measure, also referred to as F-score. There is a direct relation, thus if the values of
    precision and recall decrease, the value of F-measure will also be decreased and vice
    versa. Both precision as well as recall are required to be stable and high so as to get a
    better F-measure [27].

\[
\text{F-Score} = 2 \times \left( \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \right)
\]  

(2.7)
2.2 Computation Engines

In this section, we discuss the concepts and approaches of different computation engines. In Section 2.2.1, we discuss MapReduce and in Section 2.2.2 Apache Tez is discussed. In Section 2.2.3, a detailed explanation of Apache Spark is presented plus an elucidation of RDD, DataFrame, and Dataset is also presented.

2.2.1 MapReduce

With the intention of processing the prodigious amount of data, an algorithm was presented by Google in which divide and conquer rule was followed. It is referred to as MapReduce [28]. Map and reduce are the two major functions offered by the programming model of MapReduce. A key-value pair, that creates the intermediary set of key-value pairs, is encompassed by the Map function. The entire intermediate values which hold the identical key is blended by the reduce function [28]. For applications, both great tolerance as well as scalability is offered by it. The processing model of MapReduce is shown in Figure 2.2 [29]:

![MapReduce Processing Model](image)

Figure 2.2: MapReduce Processing Model

On the basis of Google’s MapReduce algorithm, for data processing, Apache Hadoop was developed - an open-source framework. In this open source framework, by means of map and reduce tasks, Java applications can be developed. Figure 2.3 illustrates the chief components of Hadoop and elucidation of these components is given below [30]:

a. Hadoop Common: The entire Java libraries, operating system level abstraction, vital Java files and scripts which are essential to carry out Hadoop are offered by Hadoop Common.
2.2. Computation Engines

2.2.1 Hadoop

b. **Hadoop YARN**: The framework which is used for job scheduling accompanied by cluster resource management is YARN which stands for Yet Another Resource Negotiator [31].

c. **Hadoop MapReduce**: For parallel processing of huge datasets, programming model based on MapReduce is used.

d. **Hadoop Distributed File System (HDFS)**: Distributed storage to huge datasets in the cluster is offered by HDFS. It not just gives excellent throughput access to application data but also suitable for applications that hold huge data sets [30].

2.2.2 Tez

Apache Tez, an open-source framework, was designed to build data-flow driven processing runtimes [32]. It offers a framework as well as library components that can be employed to develop effectual and scalable data-flow centric engines in a quick manner. The processing model of Tez is shown in Figure 2.4 [29].

Tez offers a reusable, flexible and extensible platform in which random data-flow oriented frameworks are supported, whereas replicated functionalities are avoided. Tez APIs let frameworks model both logical as well as physical semantics of the data flow graphs in a clear manner, with nominal code.

Following significant contributions are made by Apache Tez [33]:

- Permits model computation as a Directed Acyclic Graph (DAG)
- Uncovers APIs to progress the DAG definition in a dynamic way
- Offers an efficient and scalable execution of advanced features

Figure 2.3: Hadoop’s Components
2. Background

2.2.2.1 Comparison of Tez and MapReduce

On the basis of different parameters, a number of differences can be observed between Tez and MapReduce as shown in the following table [29]:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Apache Tez</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Model</td>
<td>specific Map phase and we possibly will have several reduce phases</td>
<td>Prior to the reduce phase, a map phase is always needed by MapReduce</td>
</tr>
<tr>
<td>Response time</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Temporary data storage</td>
<td>More efficient.</td>
<td>Not efficient. After every map and reduce phase, it keeps temporary data into HDFS [34].</td>
</tr>
<tr>
<td>Usage of Hadoop containers</td>
<td>Employ existing containers as well.</td>
<td>Extra containers are needed for further jobs.</td>
</tr>
</tbody>
</table>

Table 2.3: Difference between Apache Tez and MapReduce

2.2.2.2 Hive

Built on top of Hadoop, an open-source data warehousing solution that is referred as Apache Hive upholds queries specified in a declarative language known as HiveQL which is like SQL [35]. By means of Hadoop, the queries are compiled into MapReduce tasks.
Moreover, to link up custom MapReduce scripts into queries, HiveQL can be used. It also provides support to tables which encompass primitive types, collections like arrays and maps, etc. A system catalogue which is referred to as *Metastore*, is also included by Hive; both statistics and schemas are contained by Metastore that play a great role for data exploration, query optimization and query compilation.

Facebook uses Hive extensively. 800 thousand tables encompass 300 petabytes of data [36]. On a daily basis, 4 new Petabytes of data is generated by Facebook plus it executes 600 thousand queries in addition to 1 million jobs related to MapReduce [36].

Components that are considered as the key building blocks in Hive are *Metastore, Driver, Query Compiler, Execution Engine, HiveServer, Clients Components, and Extensibility Interfaces*. In addition, Hive’s architecture is illustrated in Figure 2.5 [35].

The fundamental architecture of Hadoop Hive represents Command Line Interface, Web Interface, and JDBC/ODBC. It illustrates that when an individual comes with Command Line Interface, it connected to Hive Drivers in a direct manner; and when an individual comes with JDBC or ODBC at that instant by means of Thrift Server (an API), it connected to Hive driver; and when an individual comes with Web Interface, it directly connected to Hive Driver.

**Hive Running on Top of Tez**

Apache Hive, one of the prominent Hadoop ecosystem projects, can be configured to use Apache Tez or MapReduce as a execution engine. Apache Hive that is ideal for batch as well as interactive queries at petabyte scale embeds Tez with the intention of translating difficult SQL statements into greatly improved, purpose-built data processing graphs. In this way, an appropriate equilibrium between performance, scalability, and throughput, is achieved. Moreover, Tez facilitates make Hive interactive.

### 2.2.3 Apache Spark

Apache Spark, an open source engine for big data processing, was initially developed in the year 2009 at UC Berkeley’s AMPLab [37]. With well-designed and significant development APIs, it was developed around speed, accessibility, and cutting-edge analytics. Spark holds flexible in-memory data processing across several use cases; this aspect of Spark allows batch processing, prompt streaming, and leading-edge modeling and analytics on the Apache Hadoop platform [38].

In carrying out extensive applications which are data-intensive on commodity clusters, MapReduce along with its variants have been very effective and thriving [6]. Nonetheless, it has also been scrutinized that an acyclic data flow model is quite inappropriate for other prevalent applications and a good deal of such systems are developed around this model. Zaharia, Chowdhury, Franklin, Shenker, and Stoica (2010) mainly focused on a particular aspect of such applications [6]. Across several parallel operations, these researchers concentrated on the class that re-uses an operational set of data. This consists of loads of interactive data analysis tools which are iterative in nature on top of
machine learning algorithms. Considering all this, these researchers anticipated a novel framework which is referred to as Spark [6]. The chief purpose of this framework is to back such applications whilst keeping both the MapReduce’s fault tolerance as well as scalability. With the intention of attaining these objectives, an abstraction which is referred to as resilient distributed datasets (RDDS) was introduced by Spark [39]. RDD is discussed in detail within Section 2.2.3.2.

Essentially, Apache Spark has a sophisticated execution engine which is referred to as directed acyclic graph (DAG) that supports in-memory processing [40].

Spark’s ecosystem is pictorially represented in Figure 2.6. On top of Spark Core, four other prominent libraries which are also considered as the chief components of the Spark ecosystem and offer extra potentials not only in the domain of Big Data analytics but
also in the area of Machine Learning are: Spark Streaming, SQL, MLlib, and GraphX [37]. They are enhanced to deal with the musts of four distinctive use cases.

### 2.2.3.1 Capabilities of Spark

In-memory cluster computing is considered as the key feature of Spark that upsurges the processing swiftness of an application [41]. Other imperative aspects of Spark are discussed below:

- **Speed**: Spark is 10 times swifter on disk and in iterative machine learning tasks [6]. And as compared to Hadoop MapReduce, by utilizing in-memory computing and other optimizations for huge scale data processing, it carries out programs almost 100 times quicker. When talking about storing data on disk, Spark is also swift and at present, it owns the record for huge scale on-disk categorization on world level. With a response period of sub-second, it can be exploited to query an approximately 40 giga-byte dataset in an interactive way [6].

- **Accessibility**: To operate on huge datasets, Spark has APIs. For transforming data, an assortment of more than hundred operators is included and to handle semi-structured data, data frame APIs are also incorporated [6]. Now, applications can be written in Java, Scala, Python, etc. in a quick and great way and this adds one more feather in the cap of Spark. Thus, it can be used from the Scala, Python etc. in an interactive manner. In addition, Spark has made the development of parallel apps easier as it gives more than eighty high-level operators. Furthermore, APIs of Java, Scala, and Python provide a platform for distributed Extract, Transform and Load (ETL) application development.

- **Cutting-edge Analytics**: A package of advanced libraries comes with Apache Spark. And a support for SQL queries, machine learning, streaming data and
graph processing is provided by these standard libraries. Besides, they not only enhance the efficiency of the developer but can be impeccably combined to build intricate workflows [37].

- **Run Everywhere:** Spark can get into different data sources comprising HDFS, HBase, Cassandra, Tachyon and so forth. By means of its individual cluster mode, Spark can be run on Apache Mesos, Hadoop, EC2 etc. [37]

### 2.2.3.2 Spark Core

Apache Spark encompasses Spark Core, which is considered as its heart, and a set of libraries. The core is considered as the distributed execution engine. It offers in-memory computing abilities to give speed, a widespread execution model to back a broad range of applications, and APIs for Scale, Java, and Python APIs to make development easy. Also, management functions such as task, is the responsibility of Spark Core.

At least any one of the libraries (involves in Spark’s ecosystem) accompanied by Spark Core is usually needed by distinct applications. Undoubtedly, those applications in which blend of two or more than two libraries (in command of Spark Core) are required, enlighten the flexible yet robust aspect of Spark [37]. In addition, the components of Spark’s ecosystem are elucidated in Section 2.2.3.3, Section 2.2.3.4, Section 2.2.3.5, and Section 2.2.3.6.

Moreover, Spark’s core abstraction for working with data is the RDD - resilient distributed dataset. It is explained in detail in the following section.

**Resilient Distributed Datasets**

As discussed above, Spark core implements and relies on a programming abstraction known as RDDs. Zaharia et al. (2012) put forward a distributed memory abstraction which is referred to as resilient distributed datasets (RDDs) [39]. In Spark Core, RDD is an immutable chief abstraction of data [6]. For programming clusters, three simple data abstractions are offered by Spark namely, RDDs, broadcast variables and accumulators. Moreover, the latter two restricted types of shared variables [6]. A read-only objects’ assortment apportioned across a set of machines is represented by RDDs. And if a partition is missed or lost, they can be re-developed. An RDD can be unambiguously cached in memory across machines and one can use it again in several MapReduce-like parallel processes. By means of an idea of lineage, this distributed memory abstraction attains fault tolerance – as discussed earlier, if RDD’s partition is lost, it can be rebuilt because it possesses sufficient information regarding how it was stemmed from other RDDs [6]. RDDs represent an optimum point or combination of factors between lucidity or clarity on the one side and scalability along with reliability on the other side; however they are specific shared memory abstraction. Furthermore, it has been analyzed that they are suitable for a range of applications.

Following are the two ways of creating RDDs in Spark [6] [42]:
2.2. Computation Engines

1. By means of dataset referencing in an external storage system, for example, a shared files system, HDFS, and so forth, or parallelizing a current assortment from driver program.

2. By altering, caching or continuing a current RDD to a different RDD.

As compared to MapReduce, more operations can be carried out on RDDs. The two chief types of RDD operations are transformations and actions [39].

Transformations: A new RDD is created from an existing one by this type of RDD operation [6]. On the whole, transformations are carried out slowly, that is RDD is simply transformed when an action is called on it.

Some of the common examples of transformations are filter(), distinct(), map(), union(), and so on.

- filter(): Using the filter operation on a function i, a new dataset will be returned via choosing the elements identified in i.
- distinct(): Distinct elements as a new dataset from the earlier dataset will be returned.
- map(): By passing the entire elements in the source by means of the specified function i in the map operation, a different dataset will be returned.
- union(): A different dataset with the union of the elements will be returned by considering dataset and the argument as input.

Actions: By carrying out the transformations, actions return a value to the driver program [43]. Some of the common examples of actions are: count(), collect(), reduce(), first() and saveAsTextFile(path).

- count(): The total number of elements contained in the dataset will be returned.
- collect(): Elements of dataset in the form of an array will be returned to the driver program.
- reduce(): By means of a specified function h, it will merge the elements in the dataset.
- first(): First element in the dataset will be returned.
- saveAsTextFile(): If the path is stated, it will save the dataset as the text file.
2.2.3.3 Spark SQL

On top of Spark Core, Spark SQL initiates another data abstraction which is referred to as SchemaRDD [37]. For both structured as well as semi-structured data, this data abstraction offers support. Moreover, for structured and semi-structured data, a new data structure, alluded to as DataFrames, is introduced by Spark SQL. DataFrames is discussed in detail in Section 2.2.1.

Basically, it gives the opportunity of initiating SQL queries in the Spark programs. Alongside command-line interfaces as well as JDBC/ODBC regulators, DataFrames gives SQL language support. Over JDBC API, the potential to reveal the Spark datasets is provided by this data structure. Plus, by means of conventional BI and visualization tools, it also permits to carry out the SQL like queries on Spark. Now, because of Spark SQL [41], users are able to ETL the data from diverse format (such as JSON, Parquet etc.); plus they can not only transform the data but also reveal it for ad-hoc querying [40].

For structured data processing, Spark SQL is one of the prominent modules of Spark. It offers interfaces that give Spark further information regarding the structure of the data along with the computation being carried out. This aspect of Spark SQL makes it different from the fundamental Spark RDD API. Internally, to carry out additional optimizations, Spark SQL makes use of this additional information. In addition, a number of ways are offered to interact with Spark SQL comprising SQL, DataFrames API and Datasets API. In the following section, DataFrame and Dataset, two important aspects of Spark SQL engine, are discussed.

DataFrame

A significant API offered by Spark SQL is DataFrame [44]. On both external data sources and incorporated distributed collections of Spark, it can carry out relational operations [38]. The widely used notion of data frame used in R and DataFrame API are analogous; however, DataFrame API assesses operations slowly with the intention of carrying out relational optimizations. In the initial progress, it was referred to as SchemaRDD since it is alike Spark RDD, however with a schema.

Within Spark programs, powerful relational integration is offered by the DataFrame API. Assortments of structured records form the DataFrames and they can be manipulated by means of procedural API of Spark, or by latest relational APIs that permit great optimizations. Besides, from incorporated distributed assortments of Java or Python objects, they can be formed in a direct way, allowing relational processing in current Spark programs [45].

Machine learning library, another Spark component, takes and produces DataFrames too. In several common circumstances, DataFrames are more appropriate plus more effective as compared to the procedural API of Spark [42]. For instance, using a SQL statement, they make it simple to compute numerous aggregates in a single pass; however it is something which is not easy to express in old functional APIs. In an automated manner,
they also store data in a columnar format that is considerably more compressed as compared to Java or Python objects. In addition to this, not like current data frame APIs in Python and R, DataFrame operations in Spark SQL make use of a relational optimizer which is referred to as Catalyst [46].

In contrast to RDD, regardless of the programming language used, DataFrame enhances the application’s performance and this notion is illustrated in Figure 2.7 [47].

In addition, following figure gives a clear view of Apache Spark components with DataFrame API:

By employing DataFrame API, a number of operations can be carried out on a DataFrame. Some of the common examples of DataFrame are count(), drop(), cache(), withColumn() etc [42]. Their description is given below:

- **count():** It returns the total number of rows in a by putting in count operation on a DataFrame a.

- **drop():** A new DataFrame, excluding column x, is returned by using drop operation of a column x on a DataFrame.

- **cache():** Using cache operation on a DataFrame a caches a in-memory.

- **withColumn():** By means of adding a new column x1 or via substituting the current column x, the withColumn operation of a column x on a DataFrame returns a new DataFrame.

**Dataset**

Spark Dataset, an extension of the DataFrame API, is a contemporary interface incorporated in Spark 1.6 which was released in 2016 [48]. A type-safe and object-oriented
24 2. Background

Figure 2.8: Apache Spark components with DataFrame API

programming interface is offered by this API. Besides, it also offers the advantages of enhanced execution engine of Spark SQL along with the pluses of RDDs; this is its great typing accompanied by the capability to employ influential lambda functions.

An API preview of Datasets is included in Spark 1.6 and it has been stated that for the following number of versions of Spark, it will be a development emphasis. By revealing expressions as well as data fields to a query planner, Datasets take benefit of Spark’s Catalyst optimizer just like DataFrames. Speedy in-memory encoding is also influenced by the Datasets. Through compile-time type safety, Datasets broaden these advantages; it means that before executing production applications, they can be scrutinized for errors. Moreover, over user-defined classes, direct operations are also allowed by them.

At the focal point, the Dataset API is a novel notion which is referred to as encoder [49]. The main accountability of encoder is to make conversion between Java Virtual Machine (JVM) objects and tabular representation. Via Spark’s internal Tungsten binary format, this tabular representation is kept. It not only permits operations on data in serial form but also enhances the usage of memory [49]. Furthermore, there are a lot of similarities between RDDs and Dataset API as both offer similar functional transformations (for instance, map, flatMap, etc.). Source code in Listing 2.1 and Listing 2.2 for RDD and Dataset API respectively, reads lines of a text file and then breaks them into words:

Listing 2.1: RDD code for reading text lines and breaking them into words

```scala
val linesOfText = sc.textFile("/infoworld")
val breakWords = linesOfText
```
2.2. Computation Engines

```scala
Listing 2.2: Dataset code for reading text lines and breaking them into words

```val``` linesOfText = sqlContext.read.text("/infoworld").as[String]
```val``` breakWords = linesOfText
  .flatMap(_.split(" "))
  .filter(_ != "")

Sooner or later, it is projected that Datasets will turn out to be a great way to write more competent Spark applications. They have been designed to work together with the existing RDD API, nonetheless enhance productivity when data can be characterized in a structured format. In a nutshell, an initial glimpse at Datasets is offered by Spark 1.6 and it is expected that in upcoming releases, it will be improved.

2.2.3.4 Spark Streaming

Building scalable fault-tolerant streaming applications and robust interactive applications has become easier because of Spark Streaming [38]. Now, same code can be used again for batch processing, join streams in contrast to past data, or execute ad hoc queries on stream state. To carry out streaming analytics, speedy scheduling ability of Spark Core is influenced by Spark Streaming. By exploiting micro batches of data which are speedily processed, Spark Streaming permits to employ the Spark’s API in streaming settings. Such design allows to use the equivalent set of batch code (RDD transformation formed it) in streaming analytics with more or less no variation. In addition to this, to process the real-time data, Spark Streaming makes use of the DStream which is essentially considered as a series of RDDs. And it can work with a number of data sources including HDFS, Flume etc. [40].

2.2.3.5 Spark MLlib

Both common learning algorithms as well as statistic utilities form Spark’s Machine Learning library (MLlib) [37] [50]. Classification, Clustering, Regression, Collaborative Filtering, Dimensionality Reduction, Extraction and features’ transformation are those chief approaches which are included in MLlib [50]. Simplifying ML pipelines in huge-scale settings was the main intention to design this library. In the current versions of Spark, MLlib and ML are the two packages of MLlib. The initial one was built on top of RDDs whereas the latter one was built on top of DataFrames for creating pipelines. Besides, this machine learning library is 9 times as speedy as the Hadoop disk-based version of Apache Mahout.

2.2.3.6 Spark GraphX

In Apache Spark, the graph processing system is referred to as Spark GraphX. Now, both graphs as well as collections can be viewed, transformed and merged interchangeably
by the users [51]. Via the Pregel abstraction, expressing the graph computation is also permitted by it. Devoid of data replication or movement, the GraphX API makes it possible for the users to observe data both as a graph and as collections (RDDs). The implementation of graph operations can be optimized by GraphX via integrating contemporary developments in graph-parallel systems. Besides, a directed multi-graph with properties affixed to every single vertex and edge is referred to as the Resilient Distributed Property Graph; and at an elevated stage, GraphX expands the Spark RDD by initiating it. A set of elementary operators, for instance subgraph, aggregate messages and so forth is exposed by GraphX with the aim of supporting graph computation [40].
3. Related Work

In this chapter, related work in the domain of Entity Resolution (ER) is discussed. To reduce the processing time of ER, different approaches exist. In this section, we chiefly focus on the aspect of improved algorithm and parallel computation in the area of ER. In Section 3.1, we consider those research studies in which significance is given on improving the algorithms of ER. Above and beyond, in Section 3.2, related work in order to improve parallel computation of ER is presented.

3.1 Improving ER Algorithms

A number of studies have been carried out so as to discuss different aspects of ER like privacy, real-time matching, or matching unstructured data [4], similarity metrics that are generally used to identify similar field entries [17], and trend towards employing training-based (supervised) and hybrid approaches to semi-automatically shape an entity matching strategy for a specified match task [52]. ER is considered as an active research topic [53] and to influence individuals for carrying out ER tasks, several frameworks have been developed in recent times like probabilistic reasoning and crowdsourcing techniques for large-scale entity linking [54] and labor independence - Declarative Crowd-Machine Data Integration [55].

3.1.1 Query-time Entity Resolution

Bhattacharya and Getoor (2007) triggered off the issue of query-time entity resolution. For collective resolution to answer entity resolution queries at query-time, these researchers concentrated on evolving novel algorithms [13]. In order to achieve fast computation to answer queries, they outperform their unrestricted expansion strategy and proposed an adaptive algorithm. The suggested algorithm cut the processing time by decreasing the size of the pertinent references and finding the most enlightening references for any query.
3.1.2 Feature Selection for ER

Yi, Xing-chun, Jian-jun, Xing and Yu-ling (2017) shaped an aggregate learning approach on the basis of feature selection for ER. The chief intention of this research was to enhance the exploitation level of high dimensional data features [56]. To unscramble the model for every single base classifier, the researchers used an adapted multi-objective ant colony optimization algorithm plus organized two pheromone matrices and to combine values of these two matrices, an approach of weighted product was applied. To get the ultimate outcome of ensemble classifiers, the classification results are combined after training all base classifiers by means of max-wins polling technique. Result showed that a better performance is achieved in contrast to the other considered approaches.

3.1.3 Hybrid Human-Machine ER

In the domain of hybrid human-machine ER, a number of research studies have been carried out. Gruenheid, Kossmann, Sukriti and Widmer (2012) revolved around the usage of strong and weak transitivity so as to tolerate errors from contradictory individual votes. The algorithm proposed in this research is quite close to $\pi_{\text{rand}}$, however every single question was asked to several humans and to find out the number of individuals to ask a question, transitivity was used [57]. No assurance of optimality is presented in this research. Contrary to this research, Vesdapunt, Bellare and Dalvi (2014) offered theoretical assurances when the crowd is ideal [58], considering the existing approaches [59] [60] [61]. so as to make crowd consistent. Vesdapunt, Bellare and Dalvi (2014) analyzed an approach based on hybrid human-machine in order to solve the ER issue [58]. By means of crowdsourcing where edges are marked with matching’s likelihood from a machine learned model, the researchers examined the issue of entirely resolving an entity graph. Eventually, this study made evident that the node-wise strategy offered in this research carried out very efficiently in practice on three different real-world data sets.

3.2 Improving Parallel Computation of ER

In order to reduce the processing time of ER using distributed environment, several methods as well as frameworks have been progressed and assessed for ER like a family of algorithms, referred to as D-swoosh, for generic and distributed entity resolution in which generic matching and merged functions were used [62] and fast approach, referred to as MD-Approach, for parallel deduplication on multicore processors in which a proficient blocking technique in conjunction with a robust data parallel programming model was employed [63]. Nevertheless, there are comparatively less techniques that give consideration to parallel entity resolution. One of the research studies demonstrated the way in which the match computation can be parallelized amid more than a few cores on a particular node [64]. Kim and Lee (2007) elucidated the parallel appraisal of the Cartesian product of two sources [65]. Kirsten, Kolb, Hartung, Gross, Kopcke, and Rahm (2010) put forward a broad model for parallel entity matching on the basis
of a balanced partitioning of the input data with the aim of creating match tasks that can be assessed in parallel [66]. This research study concentrated on parallel matching whilst blocking was not carried out in parallel and the essentials of MR was not focused.

3.2.1 Set-Similarity Joins

Vernica, Carey and Li (2010) worked on set-similarity joins, a parallel ER approach, on the basis of MapReduce programming model in Hadoop [67]. Like ER, set-similarity join is used to determine the duplicate or matching records. It is an approach where a number of records from different files are combined when a specific similarity threshold is met. Through several string similarity metric operations, similarity threshold is computed. Besides, in order to carry out set-similarity joins in an efficient way, different techniques are suggested. Gravano, Ipeirotis, Koudas and Srivastava (2003) recommended techniques to join the text for RDBMS data by means of SQL queries [68]. On the other hand, for set-similarity joins along with functional dependencies which are carried out on the subset of data, Chaudhuri (2006) employed overlap operators in SQL [69]. Filtered set-similarity join was implemented by these researchers where they combined the records when the subsets’ prefix is identical. In addition to this, based on MapReduce framework, Sarma (2014) also implemented parallel set-similarity joins in an effective way [70].

3.2.2 Dedoop – Deduplication with Hadoop

Kolb and Rahm (2012) offered a general idea of a novel tool for parallel ER on cloud infrastructures on the basis of MapReduce [53]. This new tool, referred to as Dedoop (Deduplication with Hadoop), supported a web-based specification of intricate ER tactics plus offered a huge library of blocking along with matching techniques. In this research study, it was highlighted that machine learning approaches based on training can be used with Dedoop so as to make the alignment of ER strategies with numerous similarity metrics simpler. For parallel implementation on distinct Hadoop clusters, particular ER strategies are transformed into MapReduce jobs in an automated way. In order to enhance performance, Dedoop supported improved load balancing approaches to handle data skew and kept away from redundant evaluations for multi-pass blocking. Along with this approach, there are not many advance proposals to utilize MR for ER [71] [72]. Advanced features are not supported by these approaches, for example, redundancy-free or load balancing multi-pass blocking. Also, learning-based ER is not supported by them.

3.3 Evaluating Hive and Spark SQL

Ivanov and Beer (2016) evaluated and contrasted two different processing engines: Apache Spark and MapReduce [73]. As a Big Data benchmark, these researchers employed BigBench that is considered as the fundamental end-to-end analytics Big Data benchmark. Their initial objective was to assess the benchmark by carrying out a range
of scalability tests and authenticate that it is competent to stress test the processing engines. After analyzing the steps that are essential to perform the available BigBench’s MapReduce implementation on Spark, they analyzed the changes in performance. With the aim of analyzing the changes in performance with the increase of the data size, thirty BigBench queries were carried out on Hive/MapReduce with dissimilar scale factors. Then, they executed HiveQL queries on Spark SQL and evaluated against the respective Hive runtimes. Considering the resource utilization, it was determined that Spark SQL (a) employed less CPU, although it displayed higher I/O wait as compared to Hive, (b) used less memory as compared to Hive, and (c) over the network, it sent less data as compared to Hive.
4. Methodology and Implementation

We compared the performance of two data processing engines i.e. Apache spark and Tez, to find out which one executes ER better. In this chapter we discuss how we achieved our goal. As a starting point we compared these computation engines on the basis of theories, trends and non-experimental (especially for ER) information in Section 4.1. For evaluating how helpful these theoretical concepts are in selecting the best big data tool, we designed an experiment. Details of how these experiments were designed are available in Section 4.2. Next, we needed to develop ER applications. We developed applications for Apache Spark RDD, Apache Spark DataFrame API, Apache Spark Dataset API and Hive. Implementation details for each application is provided in Section 4.3.

4.1 Theoretical Comparative Analysis of execution engines

In this section, we discuss only the theoretical aspect of Apache spark and Apache Tez. we provide some of their very generic comparison.

4.1.1 Apache Spark vs. Apache Tez

The claim made by Apache spark, brings it under the umbrella of the best performers since it states that it is a 100 times faster than the popular execution engine i.e. MapReduce[6]. They proved it by benchmarking their performances. Tez is considered a successor of MapReduce but with better efficiency. Tez also claims that it is a 100 times faster than MapReduce[5].

Considering platform dependencies for execution engines. Apache spark can run anywhere e.g. it runs as a standalone cluster, on YARN cluster and on MESOS cluster. Apache
Tez is bound to only Hadoop V2 (YARN cluster). So theoretically, this means that Apache Tez is highly optimized for running on YARN.

Apache spark provides its own abstraction, for writing applications i.e. RDD, DataFrame and Dataset API. This means these APIs are highly optimized for running on Apache spark’s core. On the other hand, to benefit from Apache Tez, we need to write applications in other tools like HIVE or PIG. So Apache spark has a unified stack unlike Apache Tez.

### 4.1.2 Apache Spark SQL vs. Hive QL

For processing data on Tez, we developed application in HIVE QL. In this section, comparison of Apache spark SQL and Hive QL is discussed. To work with structured or semistructured data in Apache Spark, we use Spark SQL. It enables a user to query structured(JSON, tables) or semistructured(XML) data in his/her code or in external tools that connect to Spark with their drivers(JDBC/ODBC). SqlContext serves as the entrance for Spark applications to run Spark SQL. We can also execute queries interactively using Spark-shell.

Hive is particularly designed for execution of SQL queries, which in Hive’s context is HQL. In Hive we can execute queries interactively or as a batch process. Although Hive QL and Spark SQL are relatively similar, they differ in some ways as follows:

**Partitioning method**  
Hive supports sharding while Spark SQL can perform native partitioning using the spark core.

**Indexing**  
In terms of maturity, Hive QL is a little behind Spark SQL when it comes to indexing. It does not provide secondary indices, as opposed to Spark SQL.

### 4.1.3 Theoretical conclusion

Apart from the technical differences between Hive and Spark as well as Hive QL and Spark SQL, it is also important to consider trends like how widely a tool is used.

![Figure 4.1: A comparison of Apache Spark, Apache Hive and Hive on Tez on Google Trends.](image)
Theoretically, a developer would definitely use the best option. Google trends indicate that barely anyone uses Apache Tez as compared to Spark. This can be observed in Figure 4.1. It also shows that despite the bold statements that “Tez” makes, it is still not as popular as Spark. Even though Apache Hive is almost as popular as Spark it still fails in comparison, and “Tez on Hive” fails miserably in the popularity section. This analysis creates an illusion that “Spark” is probably better than “Hive on Tez”. But still it is not a fact, just an illusion due to popularity of Apache spark. This thesis aims to find out whether this theoretical conclusion holds.

4.2 Design for Experimental Comparative Analysis

Section 4.1.1 describes the theoretical comparisons of Apache spark and Apache Tez. No one can judge (with a 100% guarantee) which one will perform better than the other by simply reading their documentation. To find out the best performer for a certain problem (or algorithm), we need to see the experimental performance of each execution engine. That is why, We want to see experimental performance of Apache spark and Hive running on Tez. For that purpose, we need to set a target of our experiment.

4.2.1 Target of Experiments

Our intent was to reduce the execution time of running ER algorithms for a given dataset. We did not improve the algorithm to achieve our goal, but specifically aimed at finding the best tool to process data by comparing computation and optimization powers of execution engines. Our target was to run the complete ER algorithm, which must fulfill all requirements of ER. Requirements of ER algorithm are explained in Chapter 2.

Data preprocessing We handled missing values in the input dataset by inserting “empty string” and “0” for string type values and integer/double type values respectively. We removed tags e.g. “-dup” from values of “rec-id” column. We also changed names of columns which were not acceptable for Apache Spark APIs e.g. replacing “-” with “_”.

Blocking The standard blocking technique is the heart of the algorithm, which creates blocks/clusters from the given dataset by generating BKV (Blocking-Key Value). We used first_name and surname columns for generating BKV for each record, see details in Section 4.3.3.

Pair-wise comparison After forming blocks, all the attributes of all candidates (generated by Blocking) are compared with one another. We performed self join operation to get candidates in pairs. A score was calculated for each attribute using Jarowinkler and Absolute difference functions for string and double data types respectively. A sum of individual scores formed the total score. Details of calculating the score are in Section 4.3.4.
Classification We used threshold based classification. We set a threshold value equal to 8.25, see details in Section 4.3.5. The total score of each candidate is passed through the threshold to classify it as match or non match.

Evaluation For evaluating the algorithm we have calculated Precision, Recall and F-Measure. Details of finding values for these measures are given in Chapter 2.

4.2.2 Steps to achieve the goal

In order to achieve the goal, the ER implementation had to be run on the four applications. To run these experiments, we had to perform the following steps:

1. Data Generation:
   The first step was to have input data. We generated our own datasets of different sizes by GeCo\cite{74} data-generator. In order to make our results more general and reliable, we generated several datasets with different data sizes.

2. Cluster creation:
   We chose to create a cluster using Hortonworks. Hortonworks was selected on the basis of its maturity, acceptance, open source license and hardware compatibility.

3. Installing nodes:
   A Big Data application is always incomplete without it running on multiple data nodes. In our setup, we created 7 nodes in the Hortonworks cluster.

4. Configuring applications:
   In the next step, we configured Apache Spark: RDDs, DataFrames and Datasets along with Hive on Tez using Hortonworks’ management suite: Ambari.

5. Implementing ER:
   Once the setup was completed, the code needed to be implemented. We rewrote the ER algorithm for each of the four applications.

6. Equalizing applications:
   We needed to confirm that an unbiased environment was provided for each of the applications, and that they were fair competitors. For this purpose, we created a test as the eligibility criteria as explained in Section 5.1.3. An application was only considered as a competitor, if it passed this test. In our case, every application was eligible so no further action was required.

7. Run experiments:
   In the last step the experiments were run. The results of these experiments are discussed in Chapter 5.
4.3 Implementation

This section talks about designing and development of four different applications, later on, which will be used for performing experiments. Before understanding the implemented algorithms, it is important to understand architecture of big data applications, see Section 4.3.1. Details of algorithms for each application are discussed in Section 4.3.2.

4.3.1 Architecture

This section describes the architecture of the whole system used to run our applications. Most big data applications have an architecture that comprises of three layers[75]:

1. Storage layer
2. Processing layer
3. Management layer

The storage layer is at the bottom of the stack. This naturally includes HDFS; HDFS being the primary storage is referred as a heart of Hadoop. Apart from that, interactions with database are also considered as a part of this layer. When one talks about databases in a big data context, one naturally thinks about NoSQL databases. In this thesis, however, we opt to use HDFS without any need of databases.

![Figure 4.2: Architecture of our application.](image)

The processing layer operates directly on top of the storage layer. All the processing and analysis happens in this layer. YARN is the governor of this layer and provides resources
to the corresponding execution engines. Along with YARN, this layer contains the processing suites like Hive, Pig, RDD, DataFrame, Dataset etc that run on the underlying distributed-processing-technology like MapReduce, Apache Tez, Hydra or Apache Spark. This is the major research area in our thesis. Finally the management layer consists of applications that are responsible for high-level job scheduling, configuration interfaces, coordinators and workflow managers like Oozie, Zookeeper etc.

The architecture (as in Figure 4.2) of our whole application, deals with the first two layers. The storage layer consists of HDFS. YARN is a part of the processing layer above it, along with Apache Tez and Apache Spark as execution engines. We compare the computation power of Tez and Spark, given the optimization techniques of Hive for Tez with RDD, DF and DS for Spark.

4.3.2 Algorithms

We now explain the ER processing algorithm in all contexts required for each application i.e. Spark RDD, Spark DataFrame, Spark Dataset and Hive.

4.3.2.1 RDD

![Diagram](image_url)

Figure 4.3: Algorithm for ER processing using RDD

Apache spark’s API is explained in detail in Chapter 2. The diagram, in Figure 4.3 explains the process of reading our input, processing it, and then writing it to an HDFS.

Our input data is a CSV file and is available in HDFS. RDDs cannot read CSV files directly. Therefore we need to load it as a text file and then parse it accordingly. For
parsing we need to scan the entire RDD in which the text file is loaded. Parsing is followed by preprocessing of the file. This step deals with reserved words, encoding symbols and handling such inconsistencies in the text files to avoid any problems while processing them. Shortly after, a key is generated for the “entities”. Details about how this key is generated, are explained in Section 4.3.3. The purpose of this step is to enable clustering or blocking of entities, in order to reduce the processing time.

An equi-join[76] is performed on the data within the blocks. An equijoin means that a column from both tables is tested for equality of the values in it, and only the matched ones form the result set. The join is required to group the records together, that have a probability of representing the same entity. Once we have a huge set of possibilities, we minimize it by applying a scoring function on the candidates. This function is explained in section Section 4.3.4. Then this score for each candidate is passed through threshold, explained in Section 4.3.5, to take decision that it is match or non-match. The data, with information of match or non-match is persisted at an early stage to avoid recomputation of the DAG for the previous steps. So it basically optimizes the use of resources. Consequently, the results are written back to HDFS because this is the output for user who wants to do deduplication or data cleaning. But still we need to process data further to analyze efficiency of algorithm. For that purpose we need to calculate output that is F-measure, precision and recall. Explanation of these parameter is available in Chapter 2. Last step is to store that output to HDFS.

### 4.3.2.2 DataFrame and Dataset

This section describes the ER process algorithm for DataFrame API and Dataset API. The algorithm for DataFrame and Dataset APIs is almost the same as for RDDs, with some
minor differences. It directly loads CSV files using the spark-csv connector [77]. It also loads the schema of that CSV file. Due to certain limitations we process the data and add the key in two different steps unlike RDDs. The same applies for the score calculation and generating results.

4.3.2.3 Hive

ER processing of Hive also follows the footsteps of the previously mentioned applications. However, it is more different than the others.

![ER processing algorithm for Hive](image)

In order to run the algorithm on Hive, the first step remains the same as Apache Spark DataFrame and Dataset i.e. to load the CSV from HDFS. However, the next step involves configuring Hive to include the .jar files in User Defined Functions (UDF). Jar files are nothing but archived files for a java project, that clubs multiple classes and dependencies together. The .jar file in this case contains code to handle and process strings. The configuration and code can be found on an open source project [78] on Github, that is built by using Apache Maven [79].

Temporary functions need to be created, following the configuration of Hive. These functions include code for calculating text distance, Jaro-Winkler method, double metaphone algorithm, cleaning records, matching algorithm and so on. In other words, all text processing is incorporated as UDFs or temporary functions. The steps that follow viz. Creating keys, joins, calculating the score and output, are in theory, the same as the others.

4.3.3 Key Generation process

This section describes how a key is generated for entities. This forms the basis of blocking.
For the generation of the key, we consider the first and last name of the entity. We pass it through the double-metaphone algorithm \[80\]. The implementation is published at a c/c++ issue[81]. This algorithm is built to optimize the encoding of similar sounding words. The “double” in the name, refers to the fact that it yields two codes for a given name, one primary and one secondary. This helps to identify names with the same roots worldwide, and in different contexts.

Once this is done, we check the length of the string. If it is greater than 2, we only consider the first 2 characters. Otherwise, the full string is considered. This step ensures that the length of key for each: the first name and the last, is restricted to 2.

### 4.3.4 Scoring function

Once a key exists and a preliminary grouping of entities is achieved, it assigns a score to each of the match-candidates. This section explains how this score is calculated.

Once the join is performed on the candidates, we find the total score by using a combination of two algorithms: JaroWinkler [11] and AbsoluteDifference.

**JaroWinkler** The Jaro-Winkler algorithm measures the distance between strings. Mathematically we can define Jaro-Winkler distance as:

\[
D_{\text{wink}} = D_{\text{jaro}} + (\text{prefLen} \times \text{wt}(1 - D_{\text{jaro}})) \tag{4.1}
\]
Figure 4.7: Algorithm for calculating the final score

Where,

- Dwink is the Jaro-Winkler distance i.e. the output
- Djaro is the Jaro distance between two string candidates
- prefLen is the length of the common prefix (maximum 4 characters)
- wt is the decided weight for adjustment

Which brings us to the Jaro Distance[82], which was created to link records in the census calculation way back, almost 40 years ago.

\[
Djaro = \frac{1}{3} \times ((m/|s1|) + (m/|s2|) + ((m - t)/m))
\]  

(4.2)

Where,

- Djaro is the Jaro distance i.e. the output
- M is the number of matching characters
- T is the transpose of half the matches, required when the matches appear out of order.
- S1 and S2 are strings; and the pipes indicate their lengths
AbsDiff. For the integers, we opt to choose the Absolute Difference. In other words, the difference between two integers disregarding the direction, is taken into account. As the variance of difference between integers can be higher, we alter the function so that output of this function will be “1” or “0”. If the value of both integers are same then the output will be “1” otherwise “0”. This dampens the affect of large valued integers.

4.3.5 Thresholding

This section describes how matches and non matches are decided based on a threshold value on the total score. The total score is passed for a threshold value of 8.25. This

Figure 4.8: Filtering results to identify matches

value has chosen after a series of experiments on training data, and is found to be most effective. Every pair that crosses over the 8.25 value, is considered a match. Every other pair is regarded as a non-match.
5. Experiments

In Section 4.1 we have done the theoretical comparison of execution engines and then we have planed to analyze that conclusion, experimentally. In this chapter we discuss the experimental setup required to perform experiments, considering Apache spark and Tez as execution engines. Experiment results are discussed in Section 5.2, with the explanation of possible reasons involved in unexpected behavior of execution engines. Apart from our planed experiments, we performed some extra experiments to validate logics provided during discussion of results.

5.1 Experimental Setup

In this section we discuss, what are the components involved in the setup or installations. How we performed experiments and got results, on the basis of which, we have compared execution engines.

5.1.1 Environment

As we need to perform experiments on Apache Spark and Tez. Both are parallel computation engines. For parallel computation we need to setup a multi-node cluster, which must be compatible with Apache Spark and Tez. There are some Hadoop distribution vendors, which provide configurable frameworks.

In our experimental environment, Hortonworks Hadoop distribution vendor is a backbone of environment. Structure of HDP (Hortonworks Data Platform) is illustrated in Figure 5.1. We have used Spark as In-Memory computation engine and Hive running on top of Tez for processing SQL queries. YARN is used as a resource manager. HDFS is a component of storage layer of this structure. It holds the input data and results generated by applications after processing.

In our cluster there are 10 nodes installed. One node is reserved for accessing the cluster and two name nodes (one active and one standby). Rest of the 7 nodes are executor
nodes on which, execution engines can process data. As each node contains 15 GB of RAM but only 6GB is available for running applications. Specifications of each executor node is in Table 5.1. More the number of nodes, more the parallelism is possible.

<table>
<thead>
<tr>
<th>Features</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>15GB (6GB is available for running applications)</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
</tr>
<tr>
<td>Storage</td>
<td>150GB</td>
</tr>
</tbody>
</table>

Table 5.1: Specification of each node in YARN cluster.

### 5.1.2 Datasets

We generate 3 data sets of different sizes and different number of records in each dataset (details are in Table 5.2).

<table>
<thead>
<tr>
<th></th>
<th>Small dataset</th>
<th>Medium dataset</th>
<th>Large dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of records</td>
<td>105000</td>
<td>525000</td>
<td>1100000</td>
</tr>
<tr>
<td>No. of records after join</td>
<td>2054074</td>
<td>51411973</td>
<td>222160602</td>
</tr>
<tr>
<td>Size of file</td>
<td>12MB</td>
<td>60.2MB</td>
<td>125.9MB</td>
</tr>
</tbody>
</table>

Table 5.2: Statistics of datasets
5.1. Experimental Setup

5.1.3 Eligibility criteria

Given the same dataset to each application, they must generate the same value for the following metrics:

- Number of records read
- Number of records after join
- Number of actual duplicates
- Number of true positives
- Number of false positives
- Output: Precision: Recall: F-measure

<table>
<thead>
<tr>
<th></th>
<th>RDD</th>
<th>DataFrame</th>
<th>Dataset</th>
<th>Hive</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records read</td>
<td>105000</td>
<td>105000</td>
<td>105000</td>
<td>105000</td>
<td>Pass</td>
</tr>
<tr>
<td>Records after join</td>
<td>2054074</td>
<td>2054074</td>
<td>2054074</td>
<td>2054074</td>
<td>Pass</td>
</tr>
<tr>
<td>Duplicates</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>Pass</td>
</tr>
<tr>
<td>True positives</td>
<td>4443</td>
<td>4443</td>
<td>4443</td>
<td>4443</td>
<td>Pass</td>
</tr>
<tr>
<td>False positives</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pass</td>
</tr>
<tr>
<td>Precision</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>Pass</td>
</tr>
<tr>
<td>recall</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>Pass</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Table 5.3: Eligibility test for small dataset

<table>
<thead>
<tr>
<th></th>
<th>RDD</th>
<th>DataFrame</th>
<th>Dataset</th>
<th>Hive</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records read</td>
<td>525000</td>
<td>525000</td>
<td>525000</td>
<td>525000</td>
<td>Pass</td>
</tr>
<tr>
<td>Records after join</td>
<td>51411973</td>
<td>51411973</td>
<td>51411973</td>
<td>51411973</td>
<td>Pass</td>
</tr>
<tr>
<td>Duplicates</td>
<td>25000</td>
<td>25000</td>
<td>25000</td>
<td>25000</td>
<td>Pass</td>
</tr>
<tr>
<td>True positives</td>
<td>21939</td>
<td>21939</td>
<td>21939</td>
<td>21939</td>
<td>Pass</td>
</tr>
<tr>
<td>False positives</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pass</td>
</tr>
<tr>
<td>Precision</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>Pass</td>
</tr>
<tr>
<td>Recall</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>Pass</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Table 5.4: Eligibility test for medium dataset

Table 5.3, Table 5.4 and Table 5.5 display values, produced by each application, for each metric by feeding small, medium and large input datasets respectively. Results of tables depicts that all of the applications have passed all tests to meet the requirements to be eligible for performance comparisons.
5. Experiments

<table>
<thead>
<tr>
<th></th>
<th>RDD</th>
<th>DataFrame</th>
<th>Dataset</th>
<th>Hive</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Records read</td>
<td>1100000</td>
<td>1100000</td>
<td>1100000</td>
<td>1100000</td>
<td>Pass</td>
</tr>
<tr>
<td>Records after join</td>
<td>222160602</td>
<td>222160602</td>
<td>222160602</td>
<td>222160602</td>
<td>Pass</td>
</tr>
<tr>
<td>Duplicates</td>
<td>100000</td>
<td>100000</td>
<td>100000</td>
<td>100000</td>
<td>Pass</td>
</tr>
<tr>
<td>True +ve</td>
<td>88140</td>
<td>88140</td>
<td>88140</td>
<td>88140</td>
<td>Pass</td>
</tr>
<tr>
<td>False +ve</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Pass</td>
</tr>
<tr>
<td>Precision</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>Pass</td>
</tr>
<tr>
<td>Recall</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>Pass</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Table 5.5: Eligibility test for large dataset

5.1.4 Procedure

Before start performing experiment, it is necessary to make sure that all applications are ready. For that purpose we passed all applications from eligibility test and each of them passed the test, see details in Section 5.1.3.

As we have 4 applications, 7 executor nodes in the cluster and 3 different input datasets (as mentioned in Table 5.2).

An important aspect, which influences number of experiments conducted, is that Apache Spark provides a number of configuration flags through which it gives control to developers to allocate resources for processing data e.g. number of executors, memory for each executor and number of cores for each executor. A developer can set these parameters while submitting an application. On the other hand Tez does this by itself. That is why we have run a number of experiments for each application of Apache Spark by varying the number of executors over three different sizes of input datasets, see Table 5.2. While for Hive running on top of Apache Tez, we have only three results one for small size input dataset, second for medium size input dataset and third for large size input dataset.

As discussed above, for Apache Spark RDD, DataFrame and Dataset APIs, it is possible to run these applications on any number of executors(in our case ranging from 1 to 7) by providing particular value to “–num-executors” flag while submitting application. In case of Hive, it is not possible because it does not provide this option or flag. Therefore for each input dataset we run Hive only once but for Apache spark each application ran 7 times , for each dataset.

Number of experiments for Apache spark applications : $3 \times 7 \times 3 = 63$
Number of experiments for Hive = 3
Total number of experiments = 66

For increasing precision of experiment results, procedure of performing experiments repeated two times more. In total, there are three results for each experiment. Average of those results for each experiment is considered as a final result.
We have compared performances of Apache Spark APIs among each other in great details. As in our experiments, Hive was running on top of Apache Tez and Tez claims that it uses resources efficiently and provides optimized execution plan. We selected the best performances of Apache spark’s applications and then compared that with Hive for each input dataset.

5.2 Experiment Results

In this section we discuss the results of experiments performed by running four different applications, implemented in Apache Spark and Hive. Background and functionality of Apache Spark APIs are described in Section 5.2.2 whereas, Hive and Tez is described in Chapter 2. Procedure of performing experiments is described in Section 5.1.4. Here we discuss only the results in great detail. We also talk about the factors involved in deviating behavior of execution engine from expected behavior, see Section 5.2.1

5.2.1 Performance stimuli independent of platform

In this section particularly, the major difference is that we are not considering any execution engine specifically. We discuss factors, which directly impact performance of any execution engine.

5.2.1.1 Direct impact of number of executors on performance

In this section we discuss the effect of increasing number of executors, on performance of execution engine to process same dataset. It can not be guaranteed that increasing number of executors will always achieve better performance. To explain it we have considered two cases.

5.2.1.1.1 Case 1

In this case we are using only two executors to process input data. The task is to perform self join on “Key” column in given dataset.

The splitting of input dataset can be considered as an ideal case, as in Figure 5.2. In this case we do not need to shuffle data to perform a self join. The workload is equally balanced on two executors.

5.2.1.1.2 Case 2

Task to perform in this case is also a self join. Input dataset is also same. The only difference is number of executors. Instead of two we are using three executors. The intention is to enhance performance by increasing resources.

As shown in Figure 5.3, increasing the number executors from 2 to 3 puts forward a challenging situation. One fact is very clear that at any cost shuffling is required. Now
optimalization techniques come into play. One possible way to shuffle is, to move data from executor one and three to executor 2. But the performance will be the worst
because, for processing data it will use only one executor. Best possible shuffling is to move first record from Executor 2 to Executor 1 and second record to Executor 3. Now this is the best possible performance gain in this situation. Still if we compare this case with case 1 (in 5.2.1.1.1). In both cases, applications are actually getting benefit from only two executors. In case 2, it needs to perform shuffling and also launch one extra container. That is why performance in case 2 will be worse than in case 1 and also in case 2 execution engine is wasting resources. So it is not necessary that performance will always be increased by increasing number of executors.

5.2.1.2 Direct impact of input data on performance

The performance of execution engines does not solely rely on its computation power. There are other factors like optimization techniques, which also play very important role in processing particular input dataset efficiently. This section describes how input data affects the performance of execution engines. The time required to process data also depends upon input data itself. In our case most expensive operation is JOIN. And in case of distributed environment it becomes even more critical. On multi node cluster, each machine processes data required to be processed by the task initiated by application master. Data distribution is significantly important.

5.2.1.2.1 Case 1

![Figure 5.4: Case1: No shuffling](image-url)
As shown in figure Figure 5.4, we are performing a self join on original input data file and we have two executors. Data will be distributed equally in ideal case. In total we have 4 records, first two go to executor 1 and and last two go to executor 2. And join will be performed on the “Key” column. In this case, no shuffling is required for self join operation.

5.2.1.2.2 Case 2

As in Figure 5.5, original input data file contains same data as in case 1 but in slightly different order. Rest of conditions and operations are exactly same as in case 1 (in 5.2.1.2.1). We are splitting data on two executors to perform self join.

In this case shuffling is required for performing self join operation. Similar keys are on different executors and they must be on same physical location to be joined into single record.

5.2.2 Comparison among Apache Spark APIs

In this section performance of Apache Spark APIs is discussed, considering results of experiments. The best performances of Apache Spark APIs will be considered and will be compared with Hive running on top of Tez in Section 5.2.3. We have considered, both of the aspects i.e. varying dataset is focused in Section 5.2.2.1 and varying number of executors focused in Section 5.2.2.2
5.2. Experiment Results

5.2.2.1 Influence of dataset size

For each API of Apache spark, we have independently evaluated results. Also confirmation experiment ran for testing biases of cluster in 5.2.2.1.1.

5.2.2.1.1 Apache Spark RDD

This section explains the experiment results of RDD API of Apache Spark and its performance on various datasets over different number of executors ranging from 1 to 7.

![Performance of RDD on different datasets](image)

A log scale[83] is used in Figure 5.6 as there is a huge difference between time consumption for large and small dataset. Figure 5.6 illustrates the time (in seconds) consumed by RDD for each dataset over different number of executors. It depicts that even on increasing number of executors, the time consumed by RDD to process same input dataset is almost same. The behavior is fortuitous as a reduced processing time is expected by increasing number of executors.

For a small dataset, splitting a job over more than one executor may increase the overhead of launching more containers and communicating with them. Therefore the reduction in processing time is probably hidden by the overhead time. But in case of medium and especially large dataset this cannot be the reason. Monitoring these jobs narrates two significant facts, which play very important role in performance.

- RDD was only occupying those resources, which were allocated i-e number of executors but actually using less than the allocated number of executors. In other words it was launching containers as mentioned in configuration but actually it was processing data only on less number of executor. For example in case of assigning 7 executors, it was processing data only on 5 executors.
• Wobbly load balancing was also noticed. Very few tasks and sometimes only one task was taking too long to be finished as compared to other tasks. This happens in RDD because of not having any optimizer, which proposes logical and physical execution plan.

These are the drawbacks of Apache spark RDDs that it uses resources inefficiently. RDDs would be used in those scenarios where there is a need to launch number of jobs to process small dataset. Usually we do not have these use cases in the field of big data.

**Confirmation Experiment**  An unexpected behavior of Apache spark RDD is noticed. It was supposed to be more efficient as we were increasing number of executors to process same amount of data. For confirming that there is no biases or any kind of cluster specific hurdle, particularly for apache spark RDD. We have performed very generic problem that is “word count” problem. We have used code given in Apache Spark’s documentation for RDD. We have run this experiment by varying number of executors ranging from 1 to 7 and noticed time consumed in each case.

![Figure 5.7: Experiments ran for confirming no biases in cluster](image)

Figure 5.7 illustrates very smooth and very expected behavior of RDD’s performance. As the number of executors increases, it consumes less time to process input data. This means, that nothing is wrong with cluster. Reason of bad performance or bad utilization of resources is solely because of RDD’s execution strategy for executing particular algorithm on particular input dataset.

### 5.2.2.1.2 Apache Spark DataFrame

This section focuses on the DataFrame API provided by Apache Spark. DataFrame abstraction can be used when there is a need to execute SQL queries on some dataset.
5.2. Experiment Results

It also supports a number of data formats that can be loaded on single function call, details are in Section 4.3.2.2.

![Figure 5.8: Performance of DataFrame API on different datasets](image)

Behavior of DataFrames for medium and large dataset is very impressive. It reduces time consumption for processing medium dataset upto 50% when it uses two executors as compared to one executor. Then, using more than two executors, it reduces processing time upto 25% percent to that of last processing time and it was expected as it has to tackle a lot of overheads. Only exception for large dataset is performance on 7 executors is worse than the performance on 6 executors. Justification of this behavior is explained in Section 5.2.1.1.

5.2.2.1.3 Apache Spark Dataset

This section focuses on the Dataset API provided by Apache Spark. Details of Apache Spark Dataset API are provided in Chapter 2

Performance of Dataset API is almost similar to that of DataFrame API except for large dataset when it runs on single executor. The difference between Dataset API and DataFrame API is that values in Dataset API are type safe unlike DataFrame API. Dataset API provides all benefits of DataFrame API, on top of that, Dataset API saves time of developer while developing application by detecting analysis erros during compile time, see Figure 5.14. For better performance it does not provide any further optimizations.

It is quite naturalistic that for getting more, we need to do more. Same happens with Dataset API, it provides more functionality i.e. enables developer to write application using functional programming and/or structural programing. So it is the responsibility of developer to use it efficiently otherwise it will be a overhead only.
5.2.2.2 Influence of number of executors

In this section we discuss improvement in the performance of Apache Spark APIs by increasing number of executors over different sizes of input datasets, in 5.2.2.2.1 small dataset, in 5.2.2.2.2 medium dataset and in 5.2.2.2.3 large dataset. Theoretically, a linear improvement in performance is expected by increasing resources linearly\[84\]. But practically, there are some overheads which need to take some extra time for synchronization and communication among nodes. Performance gain also depends upon behavior and optimization plan of execution engine. We have calculated performance by very naive approach:

\[
Performance = \frac{1}{TimeConsumedInSeconds} \tag{5.1}
\]

5.2.2.2.1 Small input dataset

In this section, we discuss how each of the Apache Spark’s applications perform on a small dataset over different number of executors.

As in Figure 5.10, it is quite obvious that none of the Apache Spark APIs have gained considerable performance improvement. Although Apache Spark DataFrame API and Dataset API do show some improvement, they still do not beat RDD. As there are some overheads for DataFrame API and Dataset API which are not intended to be implemented for processing small input datasets. On the other hand, RDD does not implement optimization techniques as in DataFrame and Dataset API. So RDD performs better in case of small input dataset.
5.2. Experiment Results

Figure 5.10: Performance gain by increasing a number of executors for processing the small dataset

5.2.2.2.2 Medium input dataset

Figure 5.11: Performance gain by increasing number of executors for processing the medium dataset
In case of medium dataset, performance of RDD keeps decreasing while increasing number of executors. It seems, RDD does not utilize resources properly. Figure 5.11 clearly depicts that RDD is wasting resources. On the other hand DataFrame API and Dataset API are utilizing resources efficiently and also making use of optimizations that is catalyst optimizer and tungsten. Both of the APIs are showing potential that we can increase number of executors to gain more performance.

5.2.2.2.3 Large input dataset

In this section, we see how each of the Apache Spark applications perform on a large dataset, as we increase the number of executors.

![Figure 5.12: Performance gain by increasing number of executors for processing the large dataset](image)

As we can see in Figure 5.12, the initial performance gain of DataFrame API and Dataset API is impressive while moving from 1 executor to two and then three. However, from 3 to 6 executors their improvement is very limited and then starts to deteriorate. Reason of deteriorating performance, even on increasing number of executors, is given in Section 5.2.1.1. On the other hand, the performance of RDD is even worse than medium input dataset.

5.2.2.3 Evaluation

Performances of Apache spark APIs by varying dataset sizes and number of executors discussed in Section 5.2.2.1 and Section 5.2.2.2 respectively. Here we evaluate results and conclude it by selecting best performances of Apache Spark APIs in Section 5.2.2.4.
5.2.3.1 Performance differences of Apache Spark APIs

Apache Spark RDD, DataFrame API and Dataset API run on same execution engine that is Apache Spark and yet differ in performance. This is because of optimization techniques used in these APIs.

Figure 5.13 illustrates the difference among three APIs. RDD runs directly on Apache spark core while DataFrame API and Dataset API run on top of SQL engine and pass through optimization phase before start of actual execution on Apache Spark core. The Catalyst Optimizer optimizes SQL queries. It creates logical and physical optimization plan. SQL engine runs on top of Apache Spark core. In most of the cases this optimization is advantageous but still sometimes it becomes only a overhead, especially in case of small input datasets, as seen in 5.2.2.2.1. We need to use it intelligently.

In Figure 5.13, there is a very clear difference between RDD and DataFrame/Dataset API. But difference between DataFrame API and Dataset API is not obvious. This is because we do not have any major difference, which influences performance. But unlike DataFrame API, Dataset API facilitates developers even more.

It can be incurred from Figure 5.14 that there are some advantages for the developer while using Dataset API for developing applications. In case of DataFrame API, analysis errors can be found at run time but in case of Dataset API developers can find analysis errors at compile time. This saves the precious time of developers, although it does not make any remarkable difference in performance of execution engine.

Besides detecting analysis error during compile time, encoders (explained in chapter Chapter 2) are available in Dataset API unlike DataFrame API. Encoders are beneficial
when there is a need of converting JVM objects to internal Spark SQL representation and vice versa. This feature makes it possible to write applications using structural programming as well as functional programming. Again, it does not make any exceptional performance differences. That is why the performance of DataFrame API and Dataset API is almost similar.

### 5.2.2.4 Result of comparisons

Comparisons among Apache Spark APIs are done. The best results for each Spark API is selected and will be used in later comparisons in Section 5.2.3. The configurations, when Apache Spark APIs performed best are shown in Table 5.6.

<table>
<thead>
<tr>
<th>API</th>
<th>Dataset</th>
<th>number of executors</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDD</td>
<td>Small</td>
<td>5</td>
</tr>
<tr>
<td>RDD</td>
<td>Medium</td>
<td>1</td>
</tr>
<tr>
<td>RDD</td>
<td>Large</td>
<td>4</td>
</tr>
<tr>
<td>DataFrame</td>
<td>Small</td>
<td>6</td>
</tr>
<tr>
<td>DataFrame</td>
<td>Medium</td>
<td>7</td>
</tr>
<tr>
<td>DataFrame</td>
<td>Large</td>
<td>6</td>
</tr>
<tr>
<td>Dataset</td>
<td>Small</td>
<td>7</td>
</tr>
<tr>
<td>Dataset</td>
<td>Medium</td>
<td>7</td>
</tr>
<tr>
<td>Dataset</td>
<td>Large</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.6: Configurations when Apache Spark APIs gave their best

### 5.2.3 Spark vs. Hive

Repeating what is constantly mentioned, Hive is not included for earlier comparisons because Hive does not provide configuration to run job on particular number of executors, the reason being that Hive runs on top of Tez and Tez claims that the user does not need to provide any configuration. It allocates and deallocates resources during job
execution in optimized way. That is why we have selected best performance of RDD, DataFrame and Dataset for each data size and included in this comparison.

5.2.3.1 Speed comparison

In this section, we compare speed of each application to process different sizes of datasets. Speed is reciprocal of the time consumed to process input dataset.

![Figure 5.15: Time consumed for processing input datasets](image)

For small dataset, RDD is the winner and Hive’s performance is the worst. However, the tables turn in case of medium and large datasets where Hive outperforms all APIs of Apache spark with a huge margin, especially in case of large dataset. As Hive stores each output in persistent storage, we use HDFS as persistent storage. Storing results back to HDFS is its implicit function but in case of spark it is not implicit. We need to explicitly write command while writing application to store data back to persistent storage for later use. This is unpleasant for spark, because it is highly optimized for in-memory computation. Section 4.3.2.3 contains explanation of steps that are required to implement ER for Hive where you can see, it does not write back to HDFS explicitly. But in case of Apache Spark APIs in Section 4.3.2.1 and Section 4.3.2.2, it clearly writes back results and output to HDFS.

5.2.3.1.1 Demonstration of theory

This section discusses the theory (given in Section 5.2.3) behind the performance lag of Apache Spark APIs, rooting from explicit write operations. Apache Spark APIs are optimized for in-memory computations, albeit do not perform their best when they need to write data, outside the memory.
To verify that logic, we are assuming that there is no need to write data back to HDFS for persistent storage. For that purpose we made changes to Apache Spark applications so that application will not store data back to HDFS. It will just process it. But for Hive we did not make any change because it store data in persistent storage implicitly.

After making changes to Apache Spark applications, we had run experiments again for medium and large datasets. This time we did not perform experiments on small dataset because performance of Apache Spark APIs was already better than Hive. Also we did not include RDD because its performance was far worse than Hive in case of medium and large datasets. Another reason of its bad performance was because of not using resources properly, it was not only about application.

![Comparison of algorithms with(prev) and without writes outside the memory](image)

Figure 5.16: Comparison of algorithms with(prev) and without writes outside the memory

As shown in Figure 5.16, we have previous and new results. For previous results we have used the prefix “Prev” and for new we did not use any prefix e.g. “Prev-DataFrame” is for previous DataFrame results and “DataFrame” is for new DataFrame results. Since we do not need to write the results back to HDFS we only need to compute it, and therefore we can see a significant increase in the performance of Apache Spark APIs, especially DataFrame APIs. There is an improvement of the same, for large datasets as well, however not as significant. A speculated reason is that for larger datasets, the intermediate data (more than 10GB) exceeds the assigned internal memory, and internal swapping is unavoidable.

For small dataset

\[
12MB = 105000\text{rows} \\
1MB = 8750\text{rows}
\]
5.2. Experiment Results

For medium dataset

\[
60.2 MB = 525000 row
\]
\[
1 MB = 8720.9 rows
\]

For large dataset

\[
125.9 MB = 1100000 rows
\]
\[
1 MB = 8737 rows
\]

Average

\[
(8750 + 8721 + 8737)/3 = 8736
\]
\[
1 MB = 8736 rows
\]

Let us consider 1MB approximately equals to 8736 rows. Now we need to find intermediate data size that data size after join.

For small dataset, data size after join

\[
NumberOfRowsAfterJoin = 2054074
\]
\[
DataSize = 2054074/8736
\]
\[
DataSize = 235 MB
\]
\[
AsRowSizeIsDoubled = 235 \times 2
\]
\[
= 470 MB
\]

For medium dataset, data size after join

\[
NumberOfRowsAfterJoin = 51411973
\]
\[
DataSize = 51411973/8736
\]
\[
DataSize = 5885 MB
\]
\[
AsRowSizeIsDoubled = 5885 \times 2
\]
\[
= 11770 MB
\]
\[
= 11.77 GB
\]

For large dataset, data size after join

\[
NumberOfRowsAfterJoin = 222160602
\]
\[
DataSize = 222160602/8736
\]
\[
DataSize = 25430 MB
\]
\[
AsRowSizeIsDoubled = 25430 \times 2
\]
\[
= 50860 MB
\]
\[
= 50.86 GB
\]
As we have estimated size of data after join. For small and medium dataset, it is not a huge intermediate data size, comparatively. But in case of large dataset, after join, size of data is more than 50 GB. In total we have 7 executors each containing 15 GB but we can’t assign whole RAM to process this job. So we have assigned 6 GB on each executor. If we consider ideal case, then workload of \( \frac{50}{7} \) = 7.14 GB will be allocated to each executor. Which can not fit into RAM of each executor. In realistic case, this situation would be even worse. This means a lot of swapping between hard disk and RAM is required. Considering all these scenarios, it seems spark is more efficient when data fits into RAM. As we are performing ER, we need to make data persistent for later use. So data has to be stored in persistent storage. Overall, in terms of efficiency for performing ER, Hive running on Apache Tez is winner.

5.2.3.2 Throughput comparison

In this section throughput of each application for each input dataset is compared. In this case throughput can be referred as number of records processed in one second. Formula for calculating throughput for each dataset is:

\[
ONORR = \text{OriginalNumberOfRecordsRead} \\
NORAJ = \text{NumberOfRecordsAfterJoin} \\
TCTPDIS = \text{TimeConsumedToProcessDataInSeconds} \\
Throughput = \frac{ONORR + NORAJ}{TCTPDIS}
\]

![Figure 5.17: Throughput of applications](image)
Figure 5.17 confirms that the throughput of Hive is the worst in case of small input dataset. But its throughput keeps increasing at a very high rate as the data size increases. Hive is potentially capable of gaining more throughput for even larger input datasets. On the other hand RDD’s throughput for medium dataset decreases considerably and remains constant thereafter. DataFrame API and Dataset API show quite an impressive throughput gain for medium dataset but only a slight improvement is achieved for large dataset. This indicates that DataFrame API and Dataset API have reached almost their maxima.

We have thus experimented with RDD, DataFrame API, Dataset API and Hive under varying conditions and metrics leading to certain conclusions discussed in the next chapter.
6. Conclusion

Our objective was to compare two technologies viz. Apache Spark and Hive running on top of Apache Tez to identify the best suited execution engine for speeding up the the ER process. We began by researching both the data processing engines.

We observed that Apache Spark was way more popular on Google Trends than Apache Tez. Popularity is a very important feature in choosing any given technology for most of the enterprises since this could mean more experts and more support. On the other hand, Apache Hive is only optimized for SQL queries and Tez for YARN. Contrary to Hive, Apache Spark is a general purpose execution engine which provides libraries for graphs, streaming, SQL and machine learning. Contrary to Tez, it can run in a stand-alone mode, on Mesos or YARN.

Considering these points, one would incline towards the idea that Apache Spark could be better suited even for processing ER. To analyze, we conducted a series of experiments. The experiments proved that for most cases especially for larger input datasets, Apache Hive is faster than Apache Spark for processing ER. We then tested them for their throughputs. Again, Hive on Tez has proved to be more efficient. The only place, where Spark beats Tez is for small datasets.

We therefore conclude by stating that for processing ER algorithms, Hive on Tez is a better option than Apache Spark.
7. Future work

So far in this thesis, it has been well established with a variety of experiments and strong theoretical concepts that Apache spark RDD is the best platform for processing small dataset and Hive running on top Apache Tez is strongly recommended for processing medium and large dataset for computing Entity Resolution. Although we have conducted experiments on Apache spark and Hive on Apache Tez that are very common and known for their high efficiencies, there are a lot of other platforms out there claiming that they have a higher computation power than others. It was not possible to accommodate all platforms, all ER algorithms and the wide range of datasets in such a short time span. But after all this work, it has been evident that detailed study should be conducted on all available platforms and scenarios. If we investigate the reason why we need this study then it goes back to a very basic reason that is a huge amount of data and tremendous benefits of computing it.

Problem  A large number of Big Data solutions can be found in the market, given the vast amount of research and uses cases. These big data solutions together make big data ecosystem. Big data ecosystem has expanded so much in size that it is turning into big data itself. It is generating a new problem of decision making for enterprises. Soon enterprises will be at an impasse about which tool/solution should be used for which problem. Binge solution disorder seems to be an appropriate name for this condition.

Solution  Most probably in near future there will be a need of decision-maker sort of software, that would be analyzing a problem/usecase from it’s basic parameters and then decide which platform is most suitable for that particular problem. For development of such software there is a need of detailed study on the footings of study presented in this thesis. The most fitting name for this software could be “BIG-MVC”.

Requirements of BIG-MVC would be
1. Model: It must provide an interface where a user can model properties of the dataset e.g. schema of the input dataset and size of the dataset.

2. Controller: It must provide an interface where a user can implement or select an already available algorithm to process the dataset. There would also be a possibility of manually shortlisting big data tools.

3. View: It must contain a visualization interface where a user can view the performance of active execution engines, in the form of charts.

BIG-MVC would benchmark the input dataset considering all control parameters and keep updating view interface. Finally, it would suggest a best fitting tool.
Bibliography


