Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Leveraging the Entity Matching Performance through Adaptive Indexing and Efficient Parallelization

Demetrio Gomes Mestre

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sistemas de Informação e Banco de Dados

Prof. Dr. Carlos Eduardo Santos Pires
(Orientador)

Campina Grande, Paraíba, Brasil

M586l Mestre, Demetrio Gomes.
Leveraging the entity matching performance through adaptive indexing and efficient parallelization / Demetrio Gomes Mestre. ï Campina Grande, 2018.
155 f. : il. color.

Tese (Doutorado em Ciência da Computação) ï Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.
"Orientação: Prof. Dr. Carlos Eduardo Santos Pires".
Referências.

1. Entity Matching. 2. Métodos de Indexação de EM. 3. EM em Tempo Real. 4. Computação Paralela. 5. Balanceamento de Carga. 6. MapReduce. 7. Spark. I. Pires, Carlos Eduardo Santos. II. Título.

CDU 004(043)

FICHA CATALOGRÁFICA ELABORADA PELO BIBLIOTECÁRIO GUSTAVO DINIZ DO NASCIMENTO CRB - 15/515

"LEVERAGING THE ENTITY MATCHING PERFORMANCE THROUGH ADAPTIVE INDEXING AND EFFICIENT PARALLELIZATION"

DEMETRIO GOMES MESTRE

TESE APROVADA EM 27/03/2018

CARLOS EDUARDO SANTOS PIRES, Dr., UFCG
Orientador(a)

NAZARENO FERREIRA DE ANDRADE, Dr., UFCG
Examinador(a)

CLÁUDIO DE SOUZA BAPTISTA, PhD., UFCG
Examinador(a)

DANIEL CARDOSO MORAES DE OLIVEIRA, Dr., UFF
Examinador(a)

ANA CAROLINA BRANDAO SALGADO, Dra., UFPE
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Entity Matching (EM), ou seja, a tarefa de identificar entidades que se referem a um mesmo objeto do mundo real, é uma tarefa importante e difícil para a integração e limpeza de fontes de dados. Uma das maiores dificuldades para a realização desta tarefa, na era de Big Data, é o tempo de execução elevado gerado pela natureza quadrática da execução da tarefa. Para minimizar a carga de trabalho preservando a qualidade na detecção de entidades similares, tanto para uma ou mais fontes de dados, foram propostos os chamados métodos de indexação ou blocagem. Estes métodos particionam o conjunto de dados em subconjuntos (blocos) de entidades potencialmente similares, rotulando-as com chaves de bloco, e restringem a execução da tarefa de EM entre entidades pertencentes ao mesmo bloco. Apesar de promover uma diminuição considerável no número de comparações realizadas, os métodos de indexação ainda podem gerar grandes quantidades de comparações, dependendo do tamanho dos conjuntos de dados envolvidos e/ou do número de entidades por índice (ou bloco). Assim, para reduzir ainda mais o tempo de execução, a tarefa de EM pode ser realizada em paralelo com o uso de modelos de programação tais como MapReduce e Spark. Contudo, a eficácia e a escalabilidade de abordagens baseadas nestes modelos depende fortemente da designação de dados feita da fase de *map* para a fase de *reduce*, para o caso de MapReduce, e da designação de dados entre as operações de transformação, para o caso de Spark. A robustez da estratégia de designação de dados é crucial para se alcançar alta **eficiência**, ou seja, otimização na manipulação de dados enviesados (conjuntos de dados grandes que podem causar gargalos de memória) e no balanceamento da distribuição da carga de trabalho entre os nós da infraestrutura distribuída. Assim, considerando que a investigação de abordagens que promovam a execução eficiente, em modo batch ou tempo real, de métodos de indexação adaptativa de EM no contexto da computação distribuída ainda não foi contemplada na literatura, este trabalho consiste em propor um conjunto de abordagens capaz de executar a indexação adaptativas de EM de forma **eficiente**, em modo batch ou tempo real, utilizando os modelos programáticos MapReduce e Spark. O desempenho das abordagens propostas é analisado em relação ao estado da arte utilizando infraestruturas de *cluster* e fontes de dados reais. Os resultados mostram que as abordagens propostas neste trabalho apresentam padrões que evidenciam o aumento significativo de

desempenho da tarefa de EM distribuída promovendo, assim, uma redução no tempo de execução total e a preservação da qualidade da detecção de pares de entidades similares.

Palavras-chave: Entity Matching, Métodos de indexação de EM, EM em tempo real, Computação Paralela, Balanceamento de Carga, MapReduce, Spark.

Abstract

Entity Matching (EM), i.e., the task of identifying all entities referring to the same real-world object, is an important and difficult task for data sources integration and cleansing. A major difficulty for this task performance, in the Big Data era, is the quadratic nature of the task execution. To minimize the workload and still maintain high levels of matching quality, for both single or multiple data sources, the indexing (blocking) methods were proposed. Such methods work by partitioning the input data into blocks of similar entities, according to an entity attribute, or a combination of them, commonly called "blocking key", and restricting the EM process to entities that share the same blocking key (i.e., belong to the same block). In spite to promote a considerable decrease in the number of comparisons executed, indexing methods can still generate large amounts of comparisons, depending on the size of the data sources involved and/or the number of entities per index (or block). Thus, to further minimize the execution time, the EM task can be performed in parallel using programming models such as MapReduce and Spark. However, the effectiveness and scalability of MapReduce and Spark-based implementations for data-intensive tasks depend on the data assignment made from map to reduce tasks, in the case of MapReduce, and the data assignment between the transformation operations, in the case of Spark. The robustness of this assignment strategy is crucial to achieve skewed data handling (large sets of data can cause memory bottlenecks) and balanced workload distribution among all nodes of the distributed infrastructure. Thus, considering that studies about approaches that perform the efficient execution of adaptive indexing EM methods, in batch or real-time modes, in the context of parallel computing are an open gap according to the literature, this work proposes a set of parallel approaches capable of performing efficient adaptive indexing EM approaches using MapReduce and Spark in batch or real-time modes. The proposed approaches are compared to state-of-the-art ones in terms of performance using real cluster infrastructures and data sources. The results carried so far show evidences that the performance of the proposed approaches is significantly increased, enabling a decrease in the overall runtime while preserving the quality of similar entities detection.

Keywords: Entity Matching, EM Indexing, real-time EM, Parallel Computing, Load Balancing, MapReduce, Spark.

Acknowledgment

To the almighty God, for His infinite mercy and to be the ever-present support in the difficult moments.

To my advisor (Carlos Eduardo Pires), for all the patience during this long journey and for the dedication of the countless meetings and revisions necessary to improve the work developed.

To my parents Heleno Mestre and Maria Helena Mestre, my sisters Débora e Daniele, my wife Palloma and my daugther Ester, for the support and encouragement.

To my friends at the Data Quality Laboratory (Andreza, Veruska, Dimas, Brasileiro and Nobrega), for the countless contributions and discussions about my work, and for the company in various adventures during the Ph.D. journey.

To the BigSEA project and State University of Paraíba, for the financial incentive (essential to my research) and support.

To the teachers (who were indescribably competent in their teachings), employees of COPIN and the Federal University of Campina Grande for the support.

Everyday life is like programming, I guess. If you love something you can put beauty into it. (Donald Knuth)

Thus far the Lord has helped us. 1 Samuel 7:12

Contents

List of Symbols

API - *Application Programming Interface*

AVL - *Automatic Vehicle Location*

BAW - *Blocked Adaptive Windowing*

BULMA - *BUS Line MAtching*

BULMA-RT- *BUS Line MAtching Real-Time*

DAG - *Acyclic Directed Graph*

DCS - *Duplicate Count Strategy*

DFS - *Distributed File System*

DNA - *Deoxyribonucleic Acid*

EM - *Entity Matching*

EMaaS - *Entity Matching as a Service*

FP - *False-Positives*

FN - *False-Negatives*

GPS - *Global Positioning System*

GTFS - *General Transit Feed Specification*

HDFS - *Hadoop Distributed File System*

JSF - *Java Server Faces*

MR - *MapReduce*

MR-DCS++- *MapReduce-based Duplicate Count Strategy*

OBAW - *Overlapped Blocked Adaptive Windowing*

RDD - *Resilient Distributed Datasets*

REST - *REpresentational State Transfer*

ROBAW - *Retrenched Overlapped Blocked Adaptive Windowing*

SBM - *Standard Blocking Method*

SNM - *Sorted Neighborhood Method*

SQL - *Structured Query Language*

S-BAW - *Spark-based Blocked Adaptive Windowing*

S-BULMA - *Spark-based BUS Line MAtching*

S-BULMA-RT- *Spark-based BUS Line MAtching Real-Time*

S-DCS++ - *Spark-based Duplicate Count Strategy*

S-OBAW - *Spark-based Overlapped Blocked Adaptive Windowing*

S-ROBAW - *Spark-based Retrenched Overlapped Blocked Adaptive Windowing*

TN - *True-Negatives*

TP - *True-Positives*

XML - *eXtensible Markup Language*

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, both academy and industry have attracted interest in efficient techniques to handle the problems brought by the complex and runtime costly data-intensive tasks when the volume of data is large [14]. Due to the impressive increasing of data generation (Big Data era), one of the main interests is the search for innovative techniques that allow for efficient processing, analyzing and mining of data sources. One of the most intriguing tasks in this scenario, which has been recognized to be of increasing importance in many application domains, is the Entity Matching [67].

Entity Matching (EM), also known as Entity Resolution, Deduplication, Record Linkage, or Reference Reconciliation [52], is the task of identifying entities referring to the same real-world object. Matching records that refer to the same entity from several data sources is an important mechanism to integrate/combine information and improve data quality, as well as to enrich data to enable a more effective data analysis. Thus, the task is a fundamental problem in every information integration and data cleansing application [49], *e.g.*, to find duplicate publication titles or to match authors in Digital Libraries data sources. It is also essential for other types of applications, such as web pages deduplication [14], plagiarism detection [22], and click fraud detection [65]. However, together with its importance comes its difficulties. In practice, the EM execution is high costly when the data sources involved are too large. For example, the matching of two data sources, each one containing one million entities, would result in about one trillion of possible entity pair comparisons.

There are two typical situations when we need to resolve entities in data sources. The first refers to the task of finding similar entities in a single data source, and the second one

refers to the special case of finding similar entities available in two or more data sources [52]. Both situations share two major challenges that makes the EM task execution computational costly, especially in the Big Data era.

The first challenge relies on the fact that each entity (from the same or different data sources) needs to be compared with all others, *i.e.*, to calculate the entity similarity based on the Cartesian product (naïve[1]). The application of the Cartesian product implies the task execution time to reach an asymptotic complexity $O(n^2)$, which makes the EM task inefficient when dealing with large data sources (with sizes in the order of millions of entities). This generation condition of data-intensive processing, characterized as a batch processing, is indicated when the EM must be applied over data sources whose data does not have a dynamic nature[2]. When the data source has a dynamic nature, such as sensor streaming storage, the second challenge, related to the high frequency of modifications in large data sources, emerges. Regarding data sources whose periodicity to process new data is small, perform the EM task within a short time interval, and still maintain a high EM quality (matching rate), is rather challenging [75].

To address the first challenge, which means to minimize the workload caused by the Cartesian product execution and still maintain high levels of matching quality, a commonly used solution is to reduce the search space by applying indexing (blocking) techniques [14, 61, 62, 93]. Such methods work by partitioning the input data into blocks of similar entities, according to an entity attribute, or a combination of them, commonly called "blocking key", and restricting the EM process to entities that share the same blocking key (*i.e.*, belong to the same block). For instance, it is sufficient to compare entities of the same publication year when matching published articles. Among the existing indexing methods in the state of the art, the methods that have adaptive characteristics are the most promising in terms of performance. Such methods have the ability to adapt (resize) the entity search space based on the detection rate of similar entities. Thus, the adaptive indexing methods promote the decrease of the amount of comparisons made unnecessarily and the increase of the quality of the EM. Adaptive indexing methods are part of the study object of this thesis and will be

[1] Naïve approaches are those that compute the Cartesian product in a data source

[2] A data source is considered dynamic when modifications, *i.e.*, insertions, deletes or updates, occur periodically in the data

detailed in Chapter 2.

In spite to promote a considerable decrease in the number of comparisons executed, indexing methods can still generate large amounts of comparisons, depending on the size of the data sources involved and/or the number of entities per index (or block). This condition of data-intensive processing (batch) is not indicated when the EM needs to be applied over dynamic data sources. Due to the high processing frequency of new entities in large data sources, it is impractical to process the EM task in batch whenever modifications are applied in the data sources. Thus, the second challenge is a demand for EM real-time strategies capable of promoting the rapid updating of the EM results as new data arrives. These strategies are known in the literature as strategies of real-time EM or real-time record linkage [75]. Their purpose is to promote a considerable reduction in the EM execution time in relation to the execution time required to process the EM in batch and also to achieve EM results similar to those achieved when the EM processing is applied in batch.

Nevertheless, even using indexing methods or real-time EM strategies, EM remains a heavy task to be processed for large volumes of data [50]. In particular, in the case of real-time EM, if the data streaming is large, the processing of the EM task may still be computationally costly [96]. Thus, to mitigate these difficulties, distributed computing has become an important resource for efficiently processing data and computationally intensive application tasks in the era of Big Data [50]. Extensive powerful distributed and parallel hardware and service infrastructures capable of processing millions of these tasks are available around the world. Aiming at making efficient use of such cluster environments, many programming models have been created to deal with a vast amount of data. In this context, MapReduce (MR) [23] and Spark [104], programming models for parallel processing on cluster infrastructures, have given to the data management community a powerful "chainsaw" to tackle Big Data problems. Its simplicity, flexibility, fault tolerance and capability for being a scalable parallel data-processing makes MR and Spark excellent resources for the efficient workload distribution of data-intensive tasks.

Therefore, the EM task in batch and real-time are interesting problems to be treated with distributed solutions. With the usage of several map and reduce tasks through the MR and Spark models, the EM task execution in batch or real-time can be executed in parallel efficiently. More details about the usage of the MR and Spark models to address the EM

problems will also be presented in Chapter 2.

1.1 Relevance

It is a fact that data is among the most important assets of an organization. With the increase of the data volume and improvement of the information systems capabilities to collect data from various sources, organizations are gathering large amounts of data. However, these commonly distributed and/or heterogeneous data sources often receive dirty data (*i.e.*, inaccurate/incomplete/erroneous). Thus, data quality problems related to redundant data are increasingly plentiful [14]. As mentioned previously, EM is a task of high computational complexity, especially when handling large volumes of data. Excessive redundancy decreases data reliability and usability, causes unnecessary spending, customer dissatisfaction, produces poor performance indicators, and inhibits understanding of data and its values [67]. That is why EM is a problem studied over 40 years and still remains a very active research topic [3, 14, 30, 31, 34, 35, 56, 67, 95, 97].

Lately, huge EM research advances in the field of Computer Science have been seen, especially in areas such as Data Mining, Machine Learning, Information Retrieval, Database, and Data Warehousing [31, 35]. As the volume of data sources maintained by the organizations grows and the data understanding and its values are recognized as one of the greatest difficulties in the processing of such data sources, the task of identifying similar entities in these data sources, heterogeneous or not, has become more pervasive than ever before. Several EM solutions have been proposed powered by sophisticated approaches involving machine learning, natural language, and graphs processing [14, 36, 68, 74, 107]. These techniques have improved both the quality and execution time of detecting similarities in large data sources containing millions of entities.

Despite the advances, due to the enormous proportions that the data sources have achieved recently, the techniques (especially indexing techniques) elaborated during these 40 years (without taking into account parallelization mechanisms) have become inefficient in terms of execution time [30, 47]. In the current context, it is no longer interesting to have an EM process that runs for several days [16]. To overcome this inefficiency, the scientific community has emphasized the use of distributed computing to process EM tasks by propos-

ing distributed approaches that reflect the same behavior of already consolidated serialized approaches [50, 61]).

Thus, in order to enable the already consolidated EM approaches to execute without delay over data sources sized in the order of millions of entities, it is notorious the demands for studies about the EM task performance gain, in terms of execution time. This notoriety is among the reasons why this doctoral thesis is partially included in the EUBra-BIGSEA project. This project is a research and innovation action funded by the European Commission under the co-operative program Horizon 2020 (grant agreement No. 690116). EUBra-BIGSEA is the result of the 3rd BR-EU Coordinated Call for Information and Communication Technologies (ICT), announced by the Brazilian Ministry of Science, Technology and Innovation (MCTI).

In the context of the BIGSEA project, the smart city applications developed during the project faced several problems regarding the integration of the transportation open data sources. Several predictive algorithms embedded in these applications needed high-quality data sources linkage of the historical geospatio-temporal information of public bus transportation to mitigate the built-in margin of the predictive models error and thus provide high-quality bus trip/time predictions. Regardless of recent advances in the availability and formats for sharing transport data between transportation companies and the government or citizens, the formats and inconsistencies commonly prevalent in historical transport data pose a number of EM challenges for performing bus trajectories matching in order to enable a high-quality data sources integration of the historical geospatio-temporal information of public bus transportation.

1.2 Objectives

In this section, the general objective and specific objectives of this work are presented.

1.2.1 General Objective

In view of the existing demands for new EM approaches in parallel, the scope of this work covers improving the performance in terms of execution time by preserving with small losses the quality levels of EM results in the context of structured and dynamic large data sources.

1.2.2 General Hypothesis

Considering that the investigation of efficient EM adaptive indexing methods and real-time EM approaches in the context of distributed computing have not yet been contemplated in the literature, the general objective of this doctoral thesis is to propose a set of approaches capable of performing efficient adaptive indexing and real-time EM approaches using the programmatic models MapReduce and Spark. Thus, the general hypothesis of the work is to evaluate if the proposition of new approaches for adaptive indexing of the EM task and real-time EM in parallel are able to reduce significantly the execution time of the large-scale EM task and maintain with small losses the quality levels of EM results.

1.2.3 Specific Objectives

Considering the proposed general objective, this doctoral thesis has the following specific objectives:

1. To model, prototype and evaluate efficient parallel EM indexing approaches considering the parallelization of the Duplicate Count Strategy (DCS++) [28], an state-of-the-art adaptive indexing method;

2. To model, prototype and evaluate new methods for EM adaptive indexing, aiming at reducing even more the number of unnecessary comparisons, as well as approaches for the efficient parallelization of these new methods;

3. To model, prototype and evaluate a new method of EM indexing, in the context of bus trajectories matching (batch mode), as well as the approach for the efficient parallelization of this method;

4. To model, prototype and evaluate a new approach of EM adaptive indexing, in the context of real-time bus trajectories matching, as well as the approach for the efficient parallelization of this method.

1.3 Methodology

Aiming at proposing solutions to the EM problems and challenges investigated in this work, a methodology has been used that divided doctoral research into micro research projects. For each research project, a preliminary planning has been carried out which specifies a detailed description and a schedule for the following activities: i) theoretical foundation and relevance justification of the problem or the bibliographic survey area considered; ii) reading of books, articles and technical reports related to the problem considered; iii) proposing new approaches and/or methodologies to address the challenges of EM considered; iv) definition of hypothesis and planning of an experimental design aiming to evaluate the research hypothesis considered; v) formatting and discussion of the results obtained in the previous activity; vi) listing the lessons learned and conclusions from conducting the research; and vii) definition of future work.

In this work, new EM adaptive indexing methods and efficient parallel approaches of these new methods were proposed using the well-known programmatic models MapReduce and Spark. More specifically, we accomplished the following objectives: i) Investigate solutions for EM in parallel; ii) Propose an efficient EM approach using MapReduce based on a well-known state-or-the-art adaptive indexing method; iii) Propose an EM adaptive indexing method that combines the traditional blocking method with the adaptive indexing method, as well as the approach for the efficient Spark-based parallelization of this new method; iv) Propose an EM indexing method, in the context of batch bus trajectories matching, as well as the approach for the efficient parallelization of this new method; v) Propose an EM adaptive indexing, in the context of real-time bus trajectories matching, as well as the approach for the efficient parallelization of this method.

1.4 Main Results

The main result of this work is the propositions of MapReduce and Spark-based EM adaptive indexing approaches, as well as the Spark streaming-based real-time EM approach since the efficient parallelization of the EM adaptive indexing methods and real-time EM approach are open questions in literature. In this context, different approaches to reduce the execution

time of the parallel EM approaches executed in cluster environments have been investigated. Therefore, it was proposed:

1. An efficient EM approach using MapReduce based on a well-known state-or-the-art adaptive indexing method;

2. An EM adaptive indexing method that combines the traditional blocking method with the adaptive indexing method, as well as the approach for the efficient Spark-based parallelization of these new method;

3. An EM indexing method, in the context of batch bus trajectories matching, as well as the Spark approach for the efficient parallelization of this new method;

4. An EM adaptive indexing method, in the context of real-time bus trajectories matching, as well as the Spark streaming approach for the efficient parallelization of this new method.

Experimental evaluations were carried out to evaluate the efficacy and efficiency of the approaches and methods proposed in the work. Such evaluations indicate, in short, that the new approaches present solutions that significantly decrease the execution time of the EM task and achieve efficacy results equivalent to those achieved by competing methods. In addition, the parallel approaches proposed in this thesis were added as the main native library of the parallel Entity Matching tool described in Appendix B.

1.5 Research Indicators Achieved

So far, the following research indicators have been achieved:

1. Acceptance of the article "Estimating Inefficiency in Bus Trip Choices from a User Perspective with Schedule, Positioning and Ticketing Data" [9] at the IEEE Transactions on Intelligent Transportation Systems (2018).

2. Publication of the article "An Efficient Spark-based Adaptive Windowing for Entity Matching" [64] at the Journal of Systems and Software (2017).

3. Publication of the article "Towards the Efficient Parallelization of Multi-pass Adaptive Blocking for Entity Matching" [63] at the Journal of Parallel and Distributed Computing (2017).

4. Publication and presentation of the paper "Adaptive Sorted Neighborhood Blocking for Entity Matching with MapReduce" [62] in the Proc. of 30th ACM/SIGAPP Symposium on Applied Computing (2015), Salamanca, Spain.

1.6 Document Structure

The remainder of this document is organized as follows:

1. **Chapter 2 - Theoretical Foundation**: describes the topics for promoting the theoretical basis for understanding the work presented in this document. The text describes important concepts related to EM, such as indexing methods and real-time EM. In addition, the main concepts and definitions related to MapReduce and Spark-based EM indexing methods are presented.

2. **Chapter 3 - Related Work**: presents a bibliographical review regarding the main related work found in the literature and their respective proposals. Essentially, the works are divided into five groups: works describing Entity Matching and its applications, studies that investigate EM indexing methods, works involving MapReduce-based EM indexing methods, works involving Spark-based EM indexing methods and works that deal with the execution of the Bus Trajectories EM task.

3. **Chapter 4 - An Efficient MapReduce-based Approach for the Multi-pass Duplicate Count Strategy**: presents an efficient MapReduce-based approaches for a well-known state-of-the-art EM adaptive indexing method.

4. **Chapter 5 - Enhancing Entity Matching Efficiency through Adaptive Blocking and Spark-based Parallelization**: introduces the proposition of a new class of adaptive indexing methods that combines the following methods: traditional blocking and Sorted Neighborhood Adaptive in order to reduce the number of unnecessary compar-

isons. In addition, it presents efficient Spark-based approaches for the new class of adaptive indexing methods.

5. **Chapter 6 - An Effective Unsupervised Spark-based Map-Matching Technique to Identify Bus Trajectories**: introduces the proposition of a new EM adaptive indexing method able to identify bus trajectories. In addition, it presents efficient Spark-based approaches for this new EM adaptive indexing method which can be performed in batch or real-time modes.

6. **Chapter 7 - Conclusions**: presents the conclusions of the developed work and the main perspectives of the future work.

Chapter 2

Theoretical Foundation

In this chapter, we present the necessary topics for understanding our proposals. The approaches described in this document belong to the context where Entity Matching (EM) is performed over cluster computing environments. Therefore, a description of the EM and distributed programming models (*i.e.*, MapReduce and Spark) used in this work is necessary. The aim is to present their definitions and how EM can benefit from these models.

Topics are organized into three sections. Section 2.1 presents and formalizes the EM task and also describes the main EM indexing methods. Section 2.2 presents the most popular parallel programming models, known as MapReduce and Spark. In addition, it describes how the EM task can benefit from the usage of the MapReduce and Spark models and describes the limitations that must be taken into account when modeling EM approaches based on these models. Section 2.3 presents the Map-Matching problem and its notation to deal with bus trajectories matching.

2.1 Entity Matching

As mentioned in Chapter 1, Entity Matching (EM), also known as Entity Resolution, Deduplication, Record Linkage, or Reference Reconciliation [52], is the task of identifying entities referring to the same real-world object. It can be applied to both single and multiple distributed (including heterogeneous) data sources. Entities considered in these data sources commonly refer to people, such as patients, customers or tourists, but may also refer to publications or citations, products, business, among others. For example, in master data

management applications[1], a system needs to identify that the names "Jon S. Stark", "Stark, Jon" and "Jon Snow Stark" are potentially referring to the same person.

Before we formally define the Entity Matching problem, two aspects that are part of the EM task need to be clarified. These are: entity types and similarity functions.

Entity Types

The entity type refers to the set of elements (entities) that present the same semantic structure, e.g., a product or an employee, similar to the concept of object. The methods for Entity Matching in relational data sources assume that each tuple represents an entity and the values of its attributes describe it. Thus, similar values between two tuples indicate that the tuples are (potentially) duplicates. Unlike the EM methods used in relational data sources, the EM methods used for complex data structures and XML data are much more complex, due to the semi-structured and hierarchical nature of data representation [83]. The problem relies on the difficulty of determining when a child element represents part of an entity's description (as with relational attributes) or if it represents a related object (as it happens when there is a relationship with another table). In addition, elements describing the same type of entity are not necessarily structured in the same way.

Similarity Functions

Entity Matching requires a procedure to determine whether two entities are similar enough to represent the same object in the real world. A similarity function, or simply a matcher, specifies how the similarity between two entities is computed. The authors of [52] distinguish two types of matchers: attribute value matchers [12, 20] and context matchers [13] [7].

- **Attribute value matchers** use a similarity function (or distance) in the values of a pair (comparison) of the corresponding attributes or the concatenation attributes of the input data sources. Typically, it returns values between 0 and 1 indicating the degree of similarity (or distance) between two entities. The most common functions of this matcher type are the token-based similarity (e.g., the Jaccard coefficient, cosine similarity and similarity using q-grams), edit distance-based similarity (e.g., Jaro and

[1] A set of tools that consistently define and manage non-transactional entities (master data) in an organization

Jaro-Winkler distance) and hybrids (e.g., the extended Jaccard similarity, the Monge-Elkan measure and the simple TF-IDF measure) [21].

- **Context matchers** consider the context or semantic relationships between different entities for the similarity computation. For example, to know whether a publication is similar to another whose first author (of the two publications) has the same name, it is necessary to analyze the contextual information such as its affiliations or co-authors.

Thus, the Entity Matching problem, to any approach or tool that has to solve it, can be defined as follows:

Formal Definition: Given two sets of entities $A \in R$ and $B \in S$ of comparable entities from data sources R and S, the Entity Matching (EM) problem is to identify all correspondences between entities in $A \times B$ representing the same real-world object. The definition includes the special case of finding pairs of equivalent entities within a single source ($A = B, R = S$). Thus, the relation of comparable entities denotes that $R_E = (a_1, a_2, \ldots, a_n)$, where each a_i corresponds to a comparable attribute of the entity. An entity belonging to relation E has an assigned value for each attribute. This means that an entry source S contains a finite number of entity sets $e = [(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)]$. The similarity detection result is represented by a set of correspondences in which each correspondence $c = (e_i, e_j, s)$ correlates two entities e_i and e_j, from the sources R and S followed by a similarity value $s \in [0, 1]$ which indicates the similarity degree between the two entities.

The formal definition presents in its description one of the biggest EM problems: the identification of all entities referring to the same real-world object in $A \times B$. The problem relies on the heaviness of the EM task execution since there is a need for the application of similarity functions over the Cartesian product of all entities from the sets A and B. The quadratic complexity $O(n^2)$, resulting from this process, indicates that the task is highly ineffective when the number of entities from the sets A and B reaches a size in the order of millions [53].

Nevertheless, recent works have shown that both academy and industry are no longer interested in spending days executing the EM process [16] due to the financial cost associated with the waiting time, even if the data sources involved in the process are voluminous. For

instance, in the case where a data source presents a high number of redundant entities for a long period, the lack of confidence and usability of such data and, consequently, clients dissatisfaction will increase. Therefore, when we deal with EM over large data sources, the usage of EM indexing methods becomes an essential strategy to reduce the search space of entities and, thus, leverage the EM task execution.

2.1.1 Indexing Methods or Blocking

To avoid the prohibitively expensive workload caused by the Cartesian product execution and still maintain the matching quality, a straightforward manner is to reduce the search space by applying indexing (blocking) techniques [14, 61, 62, 93]. Such methods work by partitioning the input data into blocks of similar entities, according to an entity attribute, or a combination of them, commonly called "blocking key", and restricting the EM process to entities that share the same blocking key (*i.e.*, belong to the same block). For instance, it is sufficient to compare entities of the same publication year when matching published articles.

A complete survey [15] performed by Christen provides a detailed discussion of six indexing methods (with a total of 12 variations of them). The work also describes a theoretical analysis of the methods' complexity and an empirical evaluation of them over a variety of both real and synthetic data sets. The empirical investigation showed that the Standard (traditional) Blocking [5], (fixed) *Sorted Neighborhood* [39], *Sorted Blocks* [26] and Adaptive Windowing [100] achieved the highest duplicate detection rates (F-Measure) for a vast number of parameter settings. For this reason, these indexing methods have been considered in this work.

Standard Blocking Method

The Standard Blocking Method (SBM) uses a blocking key to partition the set of entities into disjoint partitions (or blocks) [14]. Thereafter, the matching technique is applied over the Cartesian product of all entities belonging to each disjoint block. A blocking key can also be composed of more than one attribute, *e.g.*, the publication year attribute can be combined with the publication authors attribute.

Thus, given one or more data sources, with n entities, which must be submitted to an

EM task, the SBM will generate b blocks. Supposing that all blocks generated have the same number of entities n/b, the total number of comparisons between entity pairs will be O(n^2/b) [5]. Obviously, this is a generalization to calculate the complexity of the SBM in the best case of distribution of entities (among the blocks). However, this assumption rarely holds when dealing with real data sources because the number of entities in each block can vary. For example, in the context of movies, where the blocking key is the *year of production*, it is likely that the number of movies produced in the year 1970 is less than the number of movie produced in 2016. Thus, the number of entity comparisons in the block of 2016 will be greater than the number of comparisons of entities in the block 1970. This difference in the block sizes may lead the problem of data skewness [49], which will be better explained in Section 2.2.3.

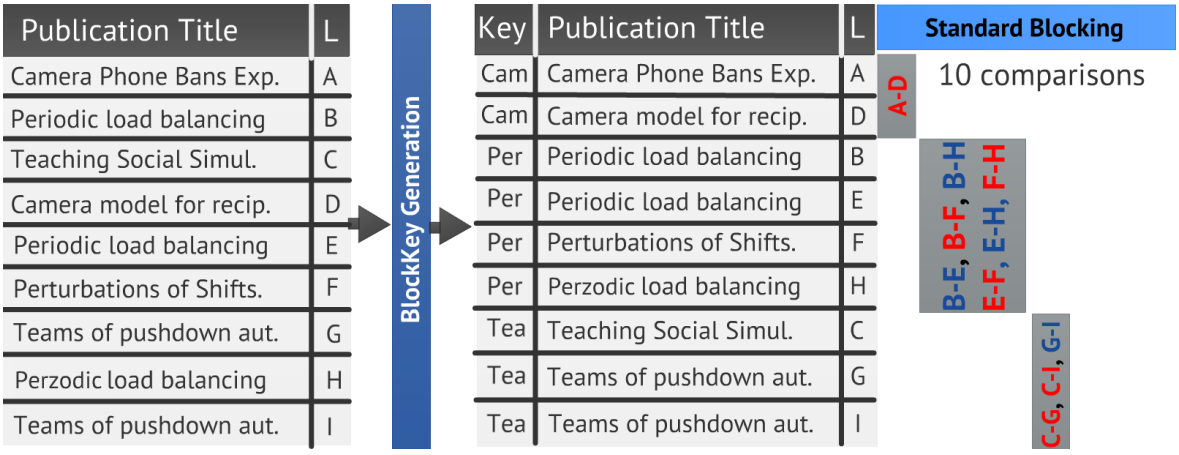

Figure 2.1: Execution example of the Standard Blocking Method.

For example, Figure 2.1 shows that the input set $PublicationTitle$ consists of n = 9 entities (from *Camera Phone Bans Exp.* to *Teams of pushdown aut.*) represented by letters L (from A to I). All the entities are grouped according to their blocking key K (Cam, Per, or Tea) which in turn is composed by the first three letters of the publication title. Note that the pairs of entity comparisons generated correspond to the Cartesian product of all entities sharing the same blocking key. Thus, the number of comparisons performed is 10 (1 for block Cam, 6 for block Per, and 3 for block Tea). Assuming an uniform distribution of entities per block, the approximate number of comparisons generated by SBM is $\frac{1}{2} \cdot (\frac{n^2}{b} - n)$, where b is the number of blocks. The running example presented in this subsection will be used throughout Chapter 2.

Note that, even using EM blocking strategies, EM remains a heavy task when processed over large volumes of data [50]. If the data is very large, even blocking the set of entities into disjoint partitions, such partitions may also have a large volume and the EM task will still be computationally costly [96]. In this case, distributed computing is an important resource for efficiently processing data and computationally intensive application tasks in the era of Big Data [50].

Fixed windowing: Sorted Neighborhood Method

Sorted Neighborhood Method (SNM) [38] is one of the most popular indexing methods. It sorts all entities using an appropriate blocking key, *e.g.*, the first three letters of the entity name, and only compares entities within a predefined (and fixed) distance window w. SNM thus reduces the execution complexity to $O(n \cdot w)$ for the actual matching. Figure 2.2 shows an execution example of SNM for a window size w = 3. Initially, the window includes the first three entities (A, D, B) and generates three pairs of comparisons $[(A - D), (A - B), (D - B)]$. After that, the window is slided down (one entity) to cover the entities D, B, E and two more pairs of comparisons are generated $[(D - E), (B - E)]$. The sliding process is repeated until the window reaches the last three entities (C, G, I). In this process, the number of comparisons generated is $(n - w/2) \cdot (w - 1)$ which results in 15 comparisons for the example.

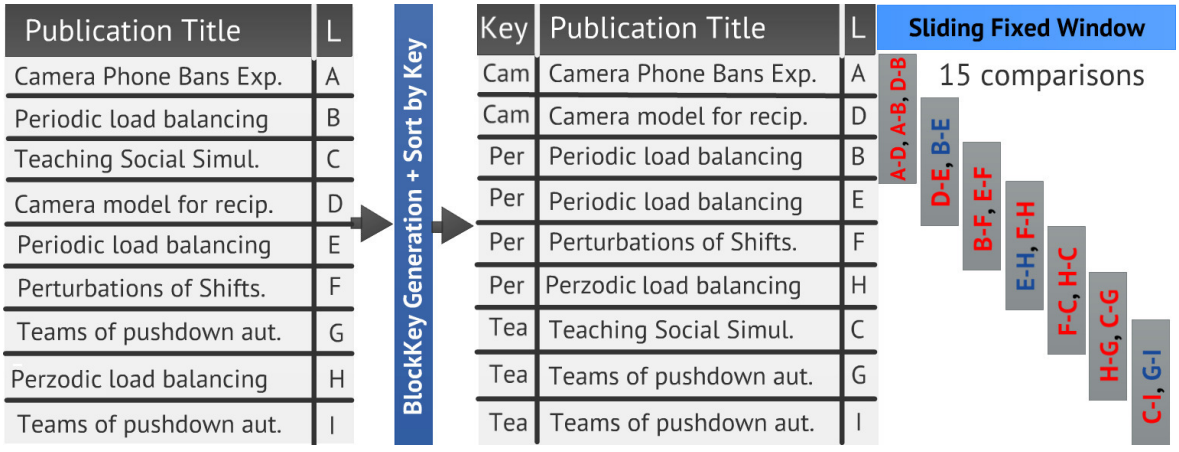

Figure 2.2: Execution example of the Sorted Neighborhood method with fixed window size w = 3.

SNM presents a critical performance disadvantage due to the fixed and difficult to config-

ure window size: if it is selected too small, some duplicates might be missed. For example, in Figure 2.2, the similar pair $B - H$ is not detected. On the other hand, a too large window leads to unnecessary comparisons. For instance, in Figure 2.2, the computation of the pair $A - B$ is unnecessary. Note that if detection quality is more relevant than the execution time, the ideal window size should be equal to the size of the largest duplicate sequence in the data set. Thus, it is common to request the intervention of a data specialist to solve this tradeoff (small/large window size).

Sorted Blocks Method

The Sorted Blocks [26] rises as an attempt to combine the best features of the Standard Blocking and the fixed Sorted Neighborhood methods. It consists in sorting the entities based on a sorting key, but instead of sliding a fixed size window over all entities, blocks are created and the entities are compared within these blocks. In other words, it compares each next entity with the rest of the entities of the current partition aiming to avoid the Cartesian product of all entities in the current partition.

The Sorted Blocks has two variants. The first one enables fixing the maximum partition size. This means that a new partition is created every time the maximum partition size is reached. By doing this, the variant prevents too large blocks and allows all entities to be compared with its predecessors and successors according to the sorting order. The second variant, *i.e.*, Sorted Blocks using window, uses the maximum partition size as the window size to slide a fixed window over the entities within a partition. The first element in the current window is removed every time the maximum number of entities within one partition is reached (end of the partition). This second variant is similar to the (fixed) Sorted Neighborhood Method. In the evaluation experiments presented in [26], the Sorted Blocks using window method presented the most promising performance result.

Figure 2.3 shows an execution example of Sorted Blocks using a window size w = 3 and a max. partition size equals to 4 (the size of largest block - Per). Initially, the window includes the first three entities (A, D, B) and generates two pairs of comparisons $[(A - D), (D - B)]$. After that, the window is slided down to the next partition (or block) since the end of the actual partition is reached. The next window covers entities B, E, F and two more pairs of comparisons are generated $[(B - E), (B - F)]$. The sliding process is repeated until the

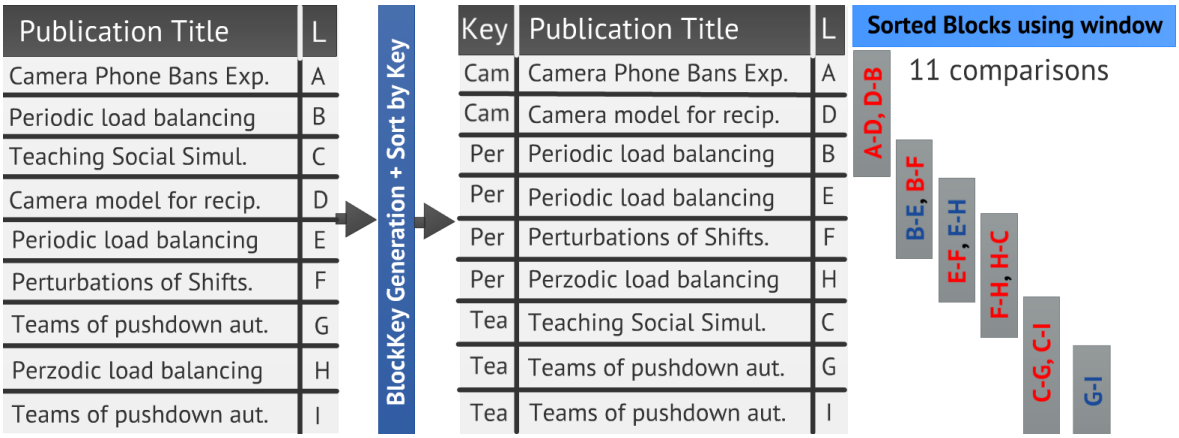

Figure 2.3: Execution example of the Sorted Blocks method using the window size w = 3.

window reaches the last three entities (C, G, I). Thus, the number of comparisons performed is 11.

Adaptive Windowing: Duplicate Count Strategy (DCS)

To overcome the tradeoff disadvantage of the fixed window size, the authors of [28] proposed an efficient SNM variation denoted as Duplicate Count Strategy (DCS) that follows the idea of increasing the window size in regions of high similarity and decreasing it in regions of low similarity. They also proved that their improved variant of DCS, known as DCS++, overcomes the performance of traditional SNM by obtaining at least the same matching results with a significant reduction in the number of entity comparisons.

The Duplicate Count Strategy (DCS) is based on the SNM and adapts the window size according to the number of already identified duplicate entities. The more duplicates of an entity are found within a window, the larger is the window. On the other hand, if no duplicate of an entity within its neighborhood is found, then DCS assumes that there are no duplicates or the duplicates are very far away in the sorting order of entities. Since the window size increases and decreases according to the number of already identified duplicates, the set of compared entities may be different from the original SNM. Adapting the window size sometimes implies in additional comparisons, but it can also reduce the number of comparisons. The DCS basic strategy consists in increasing the window size by one entity. Let d be the number of detected duplicates within a window, c the number of comparisons and ϕ_{ddr} a threshold of duplicate detection rate with $0 < \phi_{ddr} \leq 1$. DCS increases the window

size as long as $\frac{d}{c} \geq \phi_{ddr}$. Thus, ϕ_{ddr} defines the average number of detected duplicates per comparison.

DCS++ Strategy

According to the authors of [28], DCS++, a multiple entity increase variant, consists in an improvement of the DCS basic strategy. Instead of increasing the window by just one entity, DCS++ adds for each detected duplicate the next $w-1$ adjacent entities of that duplicate to the window while $\frac{d}{c} < \phi_{ddr}$. Entities are added only once to that window and the window is no longer increased when $\frac{d}{c} < \phi_{ddr} \leq \frac{1}{w-1}$. DCS++ calculates the transitive closure to save some of the comparisons: let us assume that the pairs (t_i, t_k) and (t_i, t_l) are duplicates, with $i < k < l$. Calculating the transitive closure returns the additional duplicate pair (t_k, t_l). Hence, it is not necessary to verify the window $W(k, k+w-1)$ and thus this window can be skipped. The key idea used in DCS++ to save unnecessary comparisons is to skip windows (comparisons) by transitive closure. Every time the window slides to the next entity, the size of the window is set to the initial value. An experimental evaluation showing the performance advantages of DCS++ in relation to the traditional SNM and DCS methods is shown in [28].

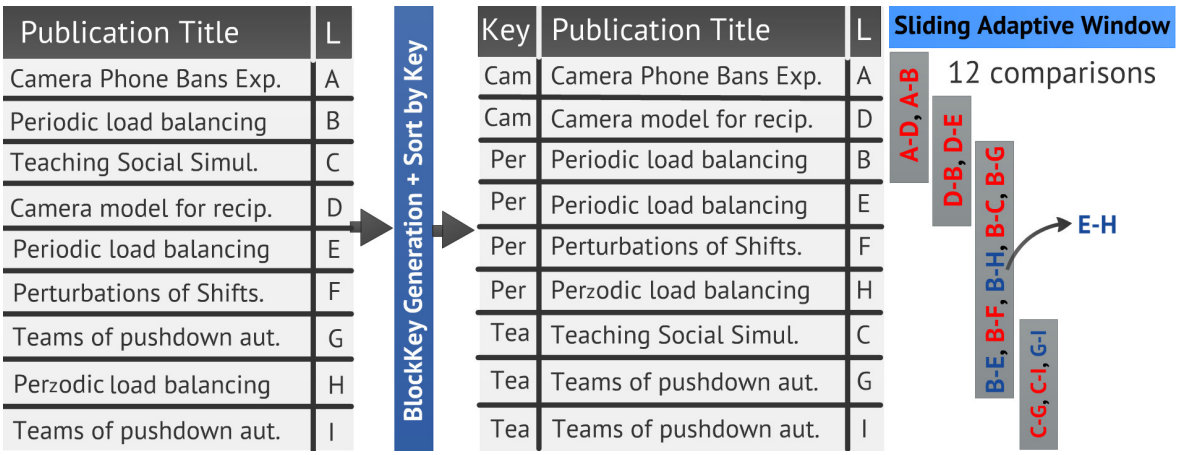

Figure 2.4: Execution example of the DCS++ method with initial adaptive window size $w_{initial} = 3$.

Figure 2.4 shows an execution example of Adaptive Windowing (DCS++) for a window size $w_{initial}$ = 3. Note that, initially, the window includes the first three entities (A, D, B) and generates two pairs of comparisons $[(A-D), (A-B)]$. Since $A-D$ and $A-B$ are regarded as non-match and since no duplicate entities are identified, there is no need to

increase the window size. After that, the window slides to the next entity (B). From B, the next comparison $B - E$ is regarded as a match. Thus, the following relation is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of already detected duplicates within the window and c is the number of comparisons already done also within the window, then w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (E). Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of E. Now, w covers B, E, F and H; the new comparisons generated are B-F and B-H. Since B-F is regarded as a non-match, the w increasing test is not satisfied. Then, the comparison B-H is performed.

Since B-H is regarded as a match, the w increasing test is satisfied once more resulting in true ($\frac{2}{3} \geq 0.5$) and the window is increased by two again. This time, the window is increased from H, the last duplicate found. Now, w covers B, E, F, H, C and G; B is compared with the rest of the entities within the window. The window is no longer increased due to the lack of new matches. Note that the pair E-H is regarded as a match due to the transitive closure assumption. After that, w is set to the initial value (three) and the window slides to the next entity (C). The windows starting with the entities E and F were skipped also due to the transitive closure assumption. Thus, the number of comparisons performed is 12.

2.1.2 Multi-pass Indexing Methods

It is important to highlight that the usage of EM indexing methods introduces an extra challenge related to choosing an ideal/effective blocking key, which may appear especially when dealing with dirty (*i.e.*, inaccurate, incomplete or erroneous) input data. In this case, it may not be sufficient to use a single blocking key to find all duplicates. To overcome this challenge, multiple blocking keys (*e.g.*, using multiple or the combination of entity attributes) are considered [38]. For each generated blocking key, a new "round" of entity comparisons is performed over the set of blocked entities according to the respective blocking key. Thus, the multi-pass variant is an important resource for the cases in which the effectiveness of the similarity detection is essential. The multi-pass variant execution of each method described in this work was considered in our evaluation.

2.1.3 Quality Measurement of EM Indexing Methods

In order to conduct the measurement of EM quality of the indexing methods, involved in this work, when applied over single or multiple data sources, this subsection describes the set of metrics utilized. To calculate the value of these metrics, it is necessary to use a data source that represents a truth table which contains the true results of the similarity relationships between all possible pair of comparisons. This truth table can be obtained by the manual or automatic labeling of the pairs of comparison of real data sources [91]. Based on this truth table, the entity pairs blocked by a given EM indexing method can be classified into one of the following categories [17]:

- **true-positives** (TP): are those pairs of entities that have been classified as similar and, in fact, they refer to the same entity;
- **false-positives** (FP): are those pairs of entities that have been classified as similar, however, according to the truth table, they do not refer to the same entity;
- **true-negatives** (TN): are those pairs of entities that have been classified as non-similar and, in fact, they do not refer to the same entity;
- **false-negatives** (FN): are those pairs of entities that have been classified as non-similar, however, according to the truth table, they refer to the same entity.

It is common that the number of true-negatives in EM results is much greater than the sum of the true-positives, false-negatives, and false-positives numbers. The reason for this is the nature of the comparison process, given that there are much more entity pairs that refer to different objects than those pairs of entities that refer to the same object [14]. An interesting result of an EM indexing method is when its indexing scheme enables the correct classification of the true-positives and, at the same time, promotes the reduction of possible false-positives and false-negatives.

Based on the number of true-positives (TP), true-negatives (TN), false-positives (FP) and false-negatives (FN), different quality metrics can be evaluated. Next, we show the most common metrics and how they should be utilized to assess the quality of the EM task.

Efficacy Metrics

The efficacy of the EM task is commonly evaluated using standard metrics, such as accuracy (A), recall (R), precision (P) and F-measure (F). The evaluation uses the truth table to determine how close to the "perfect" result a given method can achieve. These metrics are defined as follows.

- **Recall** (R): percentage of entity pairs correctly classified as true-positive over the total of truly positive entity pairs. This quality metric is calculated as follows: $R = \frac{TP}{TP+FN}$.
- **Precision** (P): percentage of entity pairs correctly classified as true-positive over the total of entity pairs classified as positive. This quality metric is calculated as follows: $P = \frac{TP}{TP+FP}$.
- **F-measure** (F): harmonic mean of precision and recall. This quality metric is calculated as follows: $F = \frac{2 \times P \times R}{P+R}$.

Performance Metrics for the Indexing Methods

To evaluate the indexing methods, three metrics are commonly used: pairs completeness, reduction ratio, F-score, execution time and speed-up. These metrics are defined as follows.

- **Pairs Completeness** (PC): indicates which part of the truly positives entity pairs have been preserved by the indexing method. Thus, PC corresponds to the recall metric and the achievement of high values, in this metric, is crucial to the method be considered of high efficacy;
- **Reduction Ratio** (RR): indicates the fraction of all possible entity pairs that have been automatically considered as non-matching by the indexing method due to its assumptions of reducing the entities search space;
- **F-score** (Fs): harmonic mean of pairs completeness and reduction ratio. This quality metric is calculated as follows: $F = \frac{2 \times PC \times RR}{PC+RR}$.
- **Execution Time** (ET): indicates the time during which a indexing method is running (executing).

- **Speed-Up** (SU): indicates the improvement in execution speed of an EM task executed on two similar architectures with different resources, *e.g.*, number of available nodes (this metric was particularly used in this work to measure the performance gain of the EM parallel approaches).

The aforementioned quality and performance metrics were utilized to evaluate the quality and performance of the strategies proposed in this work.

2.2 Distributed Computing and the Entity Matching Task

Distributed computing has received a lot of attention lately to perform data-intensive tasks. Extensive powerful distributed hardware and service infrastructures capable of processing millions of these tasks are available around the world and have being used by industry to streamline its heavy data processing. To make efficient use of these distributed infrastructures, sophisticated parallel programming models, such as the MapReduce (MR) [23] and Spark [104] emerge as a major alternatives. The reason is that these programming models can efficiently perform the distributed data-intensive tasks through map and reduce-like operations, can scale parallel shared-nothing data-processing and is broadly available in many distributions.

2.2.1 MapReduce

MapReduce is a programming model designed for parallel data-intensive computing in shared-nothing clusters with a large number of nodes [23]. The key idea relies on data partitioning and storage in a Hadoop Distributed File system, known as HDFS[3]). Entities are represented by (key, value) pairs. The computation is expressed with two user-defined functions:

$$map : (key_{in}, value_{in}) \rightarrow list(key_{tmp}, value_{tmp})$$

$$reduce : (key_{tmp}, list(value_{tmp})) \rightarrow list(key_{out}, value_{out})$$

[3]HDFS is a stable distributed file storage system for large volumes of data that comes as standard Hadoop

Each of these functions can be executed in parallel on disjoint partitions of the input data. For each input key-value pair, the map function is called and outputs a temporary key-value pair that will be used in a shuffle phase to sort the pairs by their keys and send them to the reduce function. Unlike the map function, the reduce function is called every time a temporary key occurs as map output. However, within one reduce function only the corresponding values $list(value_{tmp})$ of a certain key_{tmp} can be accessed. A MR cluster consists of a set of nodes that run a fixed number of map and reduce jobs. For each MR job execution, the number of map tasks (m) and reduce tasks (r) is specified. The framework-specific scheduling mechanism ensures that, after a task has finished, another task is automatically assigned to the released process.

Although there are several frameworks that implement the MapReduce programing model, in the scientific community, Hadoop is the most popular implementation of this paradigm. For this reason, the approaches presented in this work were implemented and evaluated using Hadoop.

2.2.2 EM Indexing Methods using MapReduce

Parallel EM implementation using blocking approaches with MR can be proposed in a simple or complex manner [50]. In a simple way, the approaches can be proposed to execute the EM task in parallel without major concerns regarding the MapReduce model limitations, such as the workload balancing and bottlenecks generated by the high consumption of distributed infrastructure resources (the MapReduce limitations will be discussed in Section 2.2.3). In a complex way, sophisticated strategies are utilized by the approaches to mitigate the influence of the MapReduce model limitations. In this section, for the sake of understanding, a simple MR-based SBM, denoted as Basic [49], will be described and exemplified.

In the Basic approach, the map process defines the blocking key for each input entity and outputs a key-value pair (blockingKey, entity). Thereafter, the default hash partitioning in the shuffle phase can use the blocking key to assign the key-value pairs to the proper reduce task. The reduce process is responsible for performing the entity matching computation for each block. An evaluation of the Basic approach showed a poor performance due to the data skewness caused by varying size of blocks [49]. The data skewness problem occurs when the match work of large blocks of entities is assigned to a single reduce task. It can

lead to situations in which the execution time may be dominated by a few reduce tasks and thus enable serious memory and load balancing problems when processing too large blocks. Therefore, concerns about lack of memory and load imbalances become necessary.

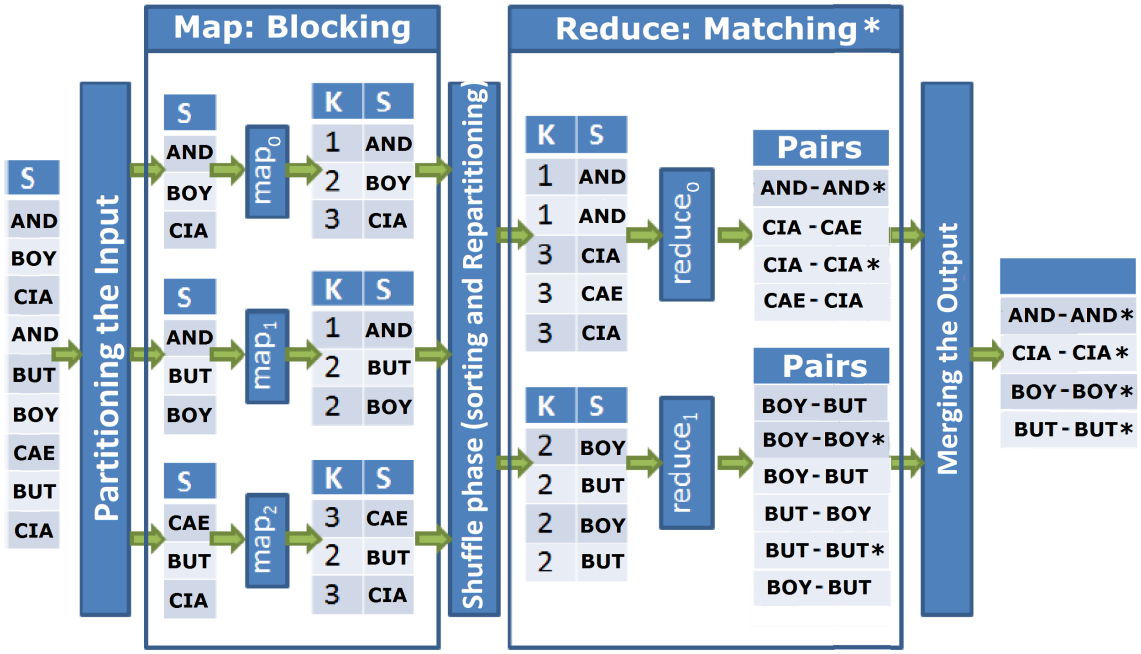

Figure 2.5: A dataflow example of a MR-based SBM, denoted as Basic ($n = 9$ input entities, $m = 3$ map tasks and $r = 2$ reduce tasks).

Figure 2.5 illustrates an example for $n = 9$ entities from a data source input S using $m = 3$ map tasks and $r = 2$ reduce tasks. Note that, first, in the partitioning step, the input set S is split into m partitions according to each map task. Then, each map task reads the data in parallel and determines the value of the blocking key K for each input entity of its partition according to the first letter of the entity. Thus, the entities starting with the letter A receive 1 as blocking key value, the ones starting with the letter B receive 2 as blocking key value and so on. For instance, AND has 1 as a blocking key value, because the entity starts with the letter A. After the blocking step (Map phase), all entities are dynamically distributed by a partitioning function in such a way that all entities sharing the same blocking key value are sent to the same reduce task. Note that in the example of Figure 2.5, entities having blocking keys 1 and 3 are sent to the reduce task 0 and the remaining entities are sent to the reduce task 1. The reduce functions (which process the reduce tasks) group the entities which are arriving locally and perform the matching of entities in parallel. For example, the reduce task

0 detected the similar pairs, marked with an "*", (AND, AND) and (CIA, CIA). Finally, the outputs of the *reduce* are merged and returned as a general EM result of the input S.

2.2.3 Limitations of the MapReduce-based EM Indexing Methods

Although the implementation of several MR-based approaches seems intuitive, it is important to note that there are three limitations inherent to the MR model capable of deteriorating or, even, making the execution of such approaches impracticable. Dealing such limitations can increase the complexity of the MR approaches which intends to be fully effective in terms of performance and should, therefore, be taken into account when developing new approaches. The limitations are described as follows.

Disjoint Data Partitioning

In the MR model, each map function reads the entities belonging to a single data partition and sends the entities also to a single output partition without possibility exchange information between the map functions (even with those that have not yet been initialized). Such situation can cause complications in the entities mapping when there is a need to compare entities belonging to different partitions. An example of this limitation can be seen in the use of the SBM when comparisons of a large block need to be split into two comparisons sets to be performed by two distinct tasks. It is necessary to ensure that the entities involved in such comparisons, which will be executed by the corresponding reduce task, be assigned appropriately to avoid the loss of comparisons and thus change the original behavior of the indexing method.

Load Balancing

Load balancing is related to the uniformity of the workload distribution between two or more distributed tasks in order to optimize the utilization of the resources, maximize performance, minimize response time and avoid overhead problems. Thus, in terms of load balancing for MR-based EM, what is sought is that each reduce task executes approximately the same number of entity comparisons. For this, the usage of some strategies, such as entity replications and fixing the number of entities per input partition, is generally indispensable to

promote the uniformity of the workload distribution.

The exact number of comparisons depends on various factors relevant to the in question problem (in this case, EM), such as data skewness (difference in sizes of entity blocks), number of available reduce tasks, EM indexing method used, among others. Thus, it is evident that the balancing solution must be treated in the proposed approach and, in this case, the default mechanism of the MR model for load balancing do not help in the solution. For example, in the systematic of the Basic approach (presented in Section 2.2.2), depending on how the blocking key is defined, there may be the generation of blocks with very large and small number of entities, which causes data skewness. Then, since all entities of the same block are sent to the same reduce task, the computation of pairs of large blocks can occupy a node for a long time and thus leave other nodes idle due to the disparity (imbalance) of the tasks size.

Memory bottlenecks

As mentioned earlier, all entities of the same block are sent for a single reduce task. Since a reduce task only can process a data row each time, similar to a SQL cursor, this means that all entities of the same block must be loaded into memory before being processed by the reduce function. Thus, the memory of a given node may not be sufficient to accommodate the entity pool storage of a very large block and the MR process may present instability. The problem of memory is strongly related to the problem of load unbalance, that is, gains in the load balancing optimization imply gains in the decrease of memory consumption per node.

2.2.4 EM Indexing Methods using Spark

MapReduce is one of the most popular cluster computing programming models. However, the model has some drawbacks. In addition to the fact that each MR job reads its input data, processes it, and then writes the output back into the distributed HDFS file system, iterative applications commonly used in the machine learning and graph analysis contexts, which require several iterations over the same data, may have their execution time deteriorated due to the repetitive (unnecessary) access to the HDFS. To improve these constraints of the MR model, Spark is driven by the Resilient Distributed Datasets (RDDs) that provide in-memory

(main) data structures to be used as immediate data cache throughout the nodes cluster. Because RDDs can be kept in memory, programs can iterate over RDDs often efficiently. Some scientists risks to say that Spark arises to succeed the MR in the efficient parallelization of computationally intensive tasks [55].

However, there are those who say that Spark was developed to improve, not replace, the Hadoop processing stack [82]. Like other storage systems, such as HBase and Amazon S3, Spark was also designed to read and write data in the Hadoop Distributed File System (HDFS). In this way, Hadoop users can enrich their processing capabilities by combining Spark with MapReduce, HBase, and other frameworks for processing and storing large volumes of data. In addition, it is becoming easier for every Hadoop user to take advantage of Spark's capabilities. The Hadoop resource manager, known as Yet Another Resource Negotiator (YARN), acts as a pivot for consistent operations transmission, providing security, tools for data governance through the Hadoop cluster and Hadoop-Spark integration.

Spark has APIs in Scala, Java and Python and libraries for streaming, graph processing and machine learning [104]. Unlike the double-step topology of MR [23], Spark has an advanced acyclic directed graph execution (DAG) engine that supports cyclic data flows. Spark also presents a programming model and offers a functional API based on Resilient Distributed Datasets (RDDs) [104]. RDD is the basic data structure of Apache Spark. It is an immutable collection of objects which computes on the various nodes of the cluster. The RDD structure is: resilient, due to the fault-tolerant with mechanisms that can recompute missing or damaged partitions according to node failures in Spark; distributed, since the data resides on multiple nodes; and a dataset, since it represents records of the data under processing. The abstraction of RDDs provides transformations (for example, map, flatmap, reduce, reduceByKey, filter, group-by, and join) and actions (for example, count and collect) that operate on partitioned data sets distributed along cluster nodes. A Spark program executes a sequence of transformations that end with an action and returns a value as a result (for example, a list of similar entities referenced by the RDD) to the Spark manager program, which can initiate another sequence of RDD transformations.

A parallel implementation of Spark-based EM using the SNM method can be developed using existing window sliding implementations. In a simplified way, SNM can be implemented using the SlidingRDD model, which returns an RDD from grouping entities of its

parent RDD in fixed size blocks by passing a sliding window over them. The SlidingRDD model is part of the MLlib[4], *i.e.*, the main scalable machine learning library of the Spark. The Spark-based SNM implementation uses the SlidingRDD model to run the fixed size sliding window on a set of entities partitions sorted according to a specific blocking key.

2.2.5 Limitations of the Spark-based EM Indexing Methods

Of the three limitations inherent to the MR model which are capable of deteriorating or even prevent the execution of the aforementioned approaches, only the mandatory use of disjoint partitions is no longer a problem in the Spark model. With the possibility of using Broadcast variables, it is possible to allow the nodes to communicate with the others cluster nodes. Broadcast variables allow the maintenance of a (read-only) variable in cache on each worker node, rather than sending a copy of the data to each worker. These variables can be used, for example, to efficiently provide each worker a copy of a look-up data source, thereby avoiding the mappers and reducers to read complementary data of the HDFS. However, the load balancing and memory bottleneck problems are still difficulties that must be treated during the proposition of Spark-based approaches.

2.3 Map-Matching

In this section, we present the necessary topics for understanding our proposal to address an EM task related to the matching of trajectories from the sequences of noisy geospatio-temporal open data sources. Since the task involves the geo-spatial EM over a digital map, this EM task is known in the literature as map-matching task [73]. In its most common formulation, map-matching aims to identify the segments of a street graph that represent the true state of noisy position observations such as those coming from GPS [73]. Thus, our proposal, developed to address this (EUBra-BIGSEA project) task, is applied as practical experience in the usage of the (adaptive) indexing methods and distributed computing in another EM contexts.

[4]http://spark.apache.org/mllib/

2.3.1 Matching Public Bus Trajectories

It is increasingly common the availability of data sources related to the transit services operations. Considering public bus services, two data sources are most often available by the transit department of several cities: (i) shapes and schedules of bus operations; and (ii) automatic vehicle location either at a certain moment or historically. In the following, we describe formats and other characteristics of each data source and discuss challenges in matching entities within these data sources.

Transit Routes and Schedule

The standard for the description of transit routes and schedule is the General Transit Feed Specification (GTFS). This specification was developed by Google and defines formats for files to be provided by an operator or authority to describe transit supply at three levels. In the first level, *routes* describe meta-information such as route name, transit mode, textual description, and the operator of the different services. In the second level, *Predefined bus trajectories* capture variations of a service over a given route represented by shape linestrings. The shape linestrings are of two types: *complementary shapes* that must join other shapes to form a complete route; and *circular shapes* that describe a complete route. The third level described in GTFS is the *stop times*, which describe the time at which a trip is expected to stop at reference locations.

In addition to this description, the location and meta-information of bus stops are typically specified, and GTFS files from a city normally specify system operation in different situations, such as weekdays and weekends, or public holidays. Among the data sources considered in this work, GTFS is by far the most often available. It is also worthwhile to comment that, since public transport systems must adapt and evolve according to multiple factors, GTFS information in a city must be dynamic. Naturally, there is often some delay for the information available in GTFS files to reflect operational changes. Section A.1 of Appendix A describes the schema of the GTFS data sources.

Vehicle Location

Automatic Vehicle Location (AVL) systems typically track the position of the fleet providing public transport in a city using the Global Positioning System (GPS). GPS devices on buses send data to a server that is commonly able to construct a real-time view of the system, as well as to create a historical record of vehicle movement. The open data made available to transport system authorities normally contains, for each message sent by a vehicle, a timestamp, the vehicle id, coordinates, and sometimes the route associated to the ongoing trip. The periodicity of data transmission from the vehicles to the server is configurable, but often uses a value in the order of dozens of seconds. In the city we consider in this thesis, this amounts to over 200MB of GPS data in an ordinary day. Section A.2 of Appendix A describes the schema of the AVL data sources.

AVL data normally contains no reference for the trajectory or schedule of the ongoing trip the vehicle sending data. In other words, by standard there is no key to directly associate a bus on a trip with a trip in the GTFS schedule or with a trajectory among those that comprise a route. This association is further complicated by the fact that sometimes vehicles deviate from prescribed trajectories. This may happen for example due to a traffic change or to an emergency. For this reason the construction of bus trajectories histories as a result of matching shapes (predefined bus trajectories) with GPS (Global Positioning System) data can be considered a challenging task. The shapes income consists of predefined bus trajectories described by multiple stops, each one represented by a set of coordinates. This problem is treated in the literature as map-matching task and is a very-well studied task in the field of Geographic Information Systems (GIS) [73]. Map-matching attempts to identify the segments of a street graph that represent the true state of noisy position observations [73].

2.3.2 The Map-matching Bus Trajectories Problem

In the context of map-matching public bus trajectories, the literature usually considers that there exists only one shape (in our case one predefined bus trajectory) per route (*e.g.*, [25] [88] [89] [8] [86] [76]). This literature has addressed this problem through several techniques to identify the trajectory performed by a bus on a digital map using sophisticated map-matching machine learning techniques.

Nevertheless, to the best of our knowledge, no technique has addressed how to efficiently match noisy position observations to multiple shapes that have different starting geo-spartial points. Figure 2.6 shows an example from the Curitiba GTFS specification where this problem arises: there are two shapes describing trajectories with the same direction that a bus from route 022 may follow during the day. Similar settings happen in other routes amounting to up to six trajectories, and we have observed similar multiplicities in the specifications of the cities, such as Curitiba (Brazil) and São Paulo (Brazil).

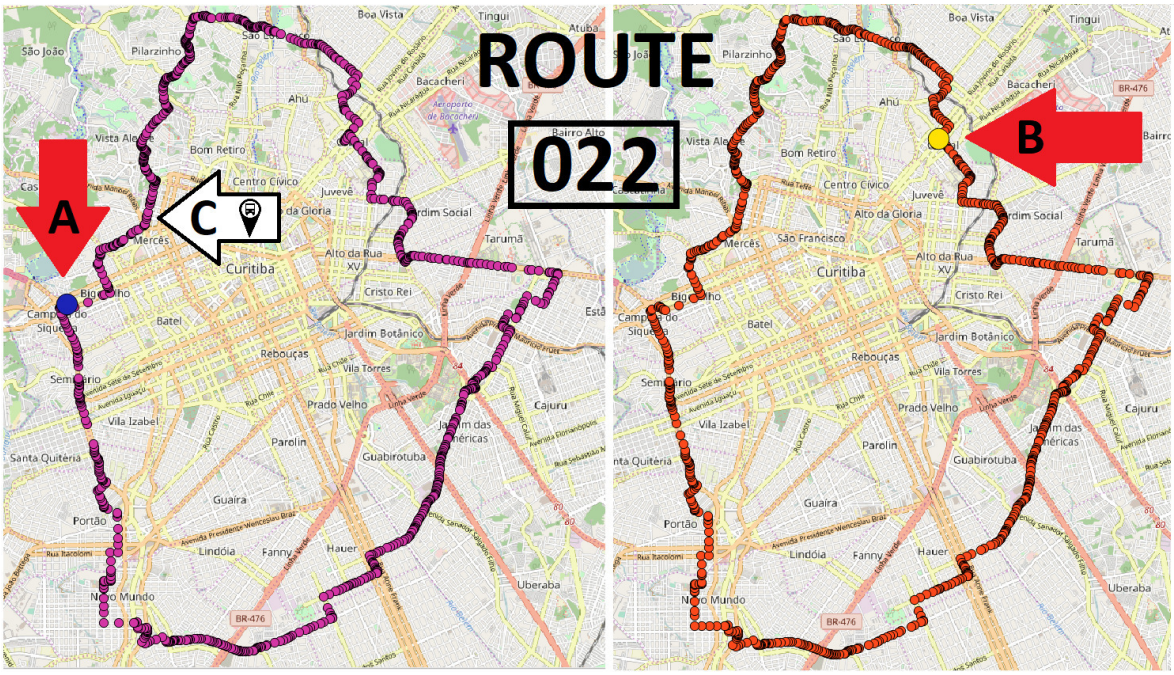

Figure 2.6: Example of a route containing two shapes representing two predefined trajectories. Each of the trajectories starts at a different point, and thus leads to a different sequence of bus stops to be visited by the bus on the route.

Most of the state-of-the-art techniques are able to detect whether or not the bus in our example is performing route 022. However, none of them are able to indicate if a bus in this setting is performing a trajectory from point A or B. This renders it impossible to integrate bus trips matched by such techniques with scheduled trips in GTFS or to use predictive models to estimate the time a bus will arrive at a given stop, as it may indeed be performing a trajectory that will not pass through that stop. For instance, in Figure 2.6, if a classification error occurs during the generation of the bus trajectories history (*e.g.*, a bus becomes associated with the shape depicted on the right of Figure 2.6 instead of the - correct - left one), a

predictive algorithm can be trained to estimate an arrival time at point C (erroneously), since the bus is programmed to finish its trip at point A (according to shape depicted on the left).

In Chapter 6, we addressed such matching bus trajectories problem by proposing two novel unsupervised techniques that are capable of matching a bus trajectory (in batch and real-time, respectively) with the correct shape when there are multiple shapes for the same route. We also proposed efficient Spark-based approaches for these new techniques.

2.4 Final Considerations

The topics presented in this chapter aimed at promoting the theoretical basis necessary to facilitate the understanding of the following chapters. For that, the main aspects related to the context of EM with and without parallelism were described. Firstly, the Entity Matching problem was described and formally defined, and a discussion was presented on the blocking methods, where several indexing methods were highlighted, since they are the methods discussed in this work. Then, the models of MapReduce and Spark were described, and how these models can be used to benefit the EM task execution. The limitations of the MR and Spark models were presented and discussed from the standpoint of using these models in the context of EM.

Finally, the necessary topics for understanding our proposal to solve a map-matching task related to the identification of bus trajectories from the sequences of noisy geospatio-temporal data sources were described. In the following chapter, the related work regarding several works involving the context of EM indexing methods, MapReduce and Spark-based EM and matching bus trajectories mentioned in this chapter will be presented.

Chapter 3

Related Work

In this chapter, we present the main works found in the literature. Essentially, the related work is divided into five groups: works describing Entity Matching and its applications, studies that investigate EM indexing methods, works involving MapReduce-based EM indexing methods, works involving Spark-based EM indexing methods and works that utilize EM approaches to solve the specific problem of matching bus trajectories regarding the presence of multiple shapes (with different starting geo-spatial points) related to the same route.

3.1 Entity Matching

Entity Matching (EM) is a very studied research topic for more than forty years [67]. The task has been investigated under various names including record or data linkage, entity resolution, object identification, field matching, merge/purge, deduplication, reference reconciliation, and others (see [14, 18, 67, 70] for recent overviews). One of the major EM difficulties is the computationally expensive execution, mainly when comparison functions involved in the EM process have a computation complexity that is quadratic in the lengths of the attribute values (that most commonly are strings). Figure 2.1 helps to illustrate that even when matching small samples, the majority of comparisons between records will correspond to non-matches (in the example, there are 36 possible comparisons and only four are true matches).

Thus, the aim of EM indexing is to reduce the number of entity pairs that are compared in detail as much as possible, by removing entity pairs that unlikely correspond to true matches.

At the same time, all entity pairs that possibly correspond to true matches (*i.e.*, where the two entities of a pair refer to the same object) need to be considered for the detailed comparison. Without the indexing step, the matching of two data sources that contain m and n entities, respectively, would result in $m \times n$ detailed entity pair comparisons. For large databases, this is clearly not feasible. In the recent years, many EM indexing methods have been proposed and evaluated as described in recent works [2, 14, 52, 70, 83, 94].

3.2 EM Indexing Methods

The study of EM traditionally focused on improving the accuracy or execution time performance of EM over static data sources (assuming that the logic and data are immutable during the matching). In this context, the Standard Blocking [5] is the most traditional of all EM indexing methods. As discussed in Subsection 2.1.1, it uses a blocking key to partition the set of entities into disjoint blocks and restricts the matching over the Cartesian product of all entities belonging to each disjoint block. Thus, the overall number of comparisons depends on the number of blocks and the block sizes. The authors of [52] defined the blocking approaches into disjoint (using a blocking key predicate, *e.g.*, the zip code of a person record) and overlapping (*e.g.*, Sorted Neighborhood Method).

Several approaches of the SNM have been proposed with the feature of multiple windows passes. The multi-pass feature is necessary to improve the accuracy due to the high dependence on the sorting key discussed in Subsection 2.1.2 [67]. The authors of [100] proposed an Adaptive Sorted Neighborhood Method that generates blocks with different sizes using a window resizing and performs the pairwise comparisons between entities belonging to these blocks. The assumption is that, in a sorted data set, similar entities are likely to be closer, although the sorting is done lexicographically and not by distance. They present two methods and compare them with the (fixed) SNM. The *Incrementally Adaptive-SNM* (IA-SNM) is a method that incrementally increases and decreases the window size according to a specified threshold. The *Accumulative Adaptive-SNM* (AA-SNM) on the other hand follows the idea of generating windows that overlap one entity. This strategy enables AA-SNM to group multiple adjacent windows into one block through transitivity.

However, the work presented in [28] showed that both IA-SNM and AA-SNM do not im-

prove the performance of SNM. Moreover, the authors proposed a new Adaptive Windowing method (DCS++) that adapts the window size according to the number of already identified duplicate entities. As mentioned earlier, the method follows the idea that the more duplicates of an entity are found within a window, the larger is the window. On the other hand, if no duplicate of an entity within its neighborhood is found, then DCS++ assumes that there are no duplicates or the duplicates are very far away in the sorting order of entities. They also compared DCS++ against the (fixed) SNM, the IA-SNM and AA-SNM methods and showed by empirical evaluation that DCS++ outperforms the three competitor methods.

The authors of [26] proposed a generalization of Standard Blocking and the (fixed) Sorted Neighborhood. This hybrid method, known as Sorted Blocks, rises as an attempt to combine the best of the Standard Blocking and the (fixed) Sorted Neighborhood method. The strategy is to divide the sorted entities into disjoint blocks and overlap some defined boundary area to approximate windowing. The results presented in this work suggest that the Sorted Blocks variant that creates new partitions when the maximum partition size is reached outperforms the Standard Blocking, (fixed) Sorted Neighborhood, IA-SNM, and all other Sorted Blocks variant methods. However, the difficulty in this hybrid method, for all Sorted Blocks variants, is that they require more parameters than the Standard Blocking or the (fixed) Sorted Neighborhood, which makes configuration slightly more difficult. The evaluation also lacks evidence about the sensibility of these parameters. Thus, the Sorted Blocks Method was included in the experimental evaluation presented in Chapter 5.

Table 3.1 presents a comparative table of the indexing methods considered in this work. The methods are compared according to the following features:

- Indexing type: the type of indexing strategies utilized by the methods. For instance, if the type is blocking, windowing or both;

- Search space: the manner in which the methods generate the candidate pair comparisons. For instance, if the search space is generated by the employ of the Cartesian product or a (fixed or adaptive) window sliding;

- Windowing adaptability: the type of a windowing method adaptability. For instance, if the adaptability is based on a multiplying factor (*e.g.*, linear or geometric increasing) or matching rate (*e.g.*, duplicate detection rate in a certain region of entities). The

Table 3.1: Comparative table of indexing methods

Methods | Criteria	**Indexing Type**	**Search Space**	**Windowing Multiplying Factor**	**Adaptability Matching Rate**
Standard Blocking [5]	Blocking	Cartesian Product in each block	-	-
Sorted Neighborhood [38]	Windowing	Slides a fixed window	NO	NO
IA-SNM and AA-SNM [100]	Windowing	Slides an adaptive window	YES	NO
Sorted Blocks [26]	Blocking and Windowing	Performs a blocking and slides an fixed window	NO	NO
Duplicate Count Strategy ++ [28]	Windowing	Slides an adaptive window Overcomes IA-SNM [100]	NO	YES, only on matching detection

hyphens ("-") means not applicable.

As it can be seen, although Table 3.1 presents two types of adaptive indexing methods (in relation to the windowing adaptability), there are at least two characteristics that were not properly exploited by the methods presented in this subsection. The first is regarding the adaptation of window size in regions with low duplicate detection rates. The second is regarding the increase of the window to perform adjacent block overlapping in order to increase the detection of pairs of similar entities. These features were not yet exploited by existing indexing methods in the literature.

3.3 MapReduce-based EM Indexing Methods

As modern databases are becoming larger, deduplicating or matching them requires increasingly massive amounts of computing power and storage resources. Researchers have begun to investigate how modern parallel and distributed computing environments can be employed to reduce the time required to conduct large-scale entity matching projects [6, 47, 54, 87, 92]. A considerable number of approaches that consider parallel EM have been proposed in recent years [77], but there is still much to be done in the era of Big Data.

The first works in order to evaluate the parallel Cartesian product of two sources are described in [46, 80]. The approaches proposed in these works were designed to promote a simple parallelization, since the objective was only to emit entity pairs for the available

nodes in the distributed infrastructure without emphasizing the use of optimization mechanisms, such as load balancing and reduction of memory consumption. To address this gap, the authors of [47] propose a generic model for parallel entity matching based on general partitioning strategies that take memory and load balancing requirements into account.

In this context, when we deal with MapReduce-based large-scale Entity Matching, two well-known data management problems must be treated: load balancing and skew handling. MR has been criticized for neglecting the problem of data skewness [24]. For this reason, works such as the parallel hash join processing [57], a skew handling mechanism available in parallel database systems, spent time injecting treatments for the data skewness into user-defined MR functions, such map, reduce, part, and group. Our work does not aim to prioritize skew handling, but to minimize its effects on performance by efficiently distributing entity replication among all nodes. Another approach to address the load imbalances promoted by the presence of data skewness was proposed by the authors of [69]. They applied a static load balancing mechanism, but it is not suitable due to arbitrary join assumptions. The authors employ a previous analysis phase to determine the data sources' characteristics (using sampling) and thereafter avoid the evaluation of the Cartesian product. This approach focus on data skew handle in the map process output, which leads to an overhead in the map phase and large amount of map output.

MapReduce has already been employed for EM (*e.g.*, [45,92,105]). In the work [92] only one mechanism of near duplicate detection by the PPjoin paradigm adapted to the MapReduce framework can be found. In addition, the authors of [92] have not conducted studies considering the robustness of their approach regarding load balance and data skewness. The works [19, 40, 43, 50, 60, 61, 101] studied load balancing and skew handling mechanisms to MapReduce-based EM for the standard blocking approach but do not consider the sorted neighborhood method. The work [90] presents another approach for parallel processing entity matching on a cluster infrastructure. This study does not involve the Sorted Neighborhood Method, but explains how a single token-based string similarity function performs with MR. The approach is based on a complex workflow consisting of several MapReduce jobs. It also suffers from load imbalances since some reduce tasks process more comparisons than others.

The authors of [49] study load balancing for MR-based traditional single- and multi-pass

Sorted Neighborhood Method (SNM). The *RepSN* approach follows a different blocking approach (fixed window size) that is by design less vulnerable to skewed data. It comprises two MR jobs. The first one calculates the *Key Partitioning Matrix (KPM)* that collects and specifies the number of entities per key and pass separated by input partitions. The matrix is used by the map tasks of the second MR job to automatically tailor entity redistribution among the reduce tasks. The second job performs the automatic partitioning using the *KPM* in the map phase and the window sliding in the reduce phase. However, its fixed window size design is the reason of the serious disadvantage mentioned earlier in Section 2.1.1, *i.e.*, if the window size is defined too small, some duplicates might be missed whereas a too large window is defined, many unnecessary comparisons will be executed. Due to its proximity with our work, we compared the *RepSN* approach [49] (state of the art) with our work in the experimental evaluation presented in Chapter 4.

Finally, the work [58] presents a MapReduce-based multi-pass approach (Partition-Sort-Map-Reduce) with adaptive sliding window. The idea consists in changing the window size through a fuzzy strategy of assigning unexplained weights as a factor to increase and decrease the window size according to the duplicate detection rate. Although this work is strongly related to ours, we did not consider it in our experimental evaluation due to three main reasons. The first one is the impossibility to reproduce. The work lacks of major details about the approach's model and implementation (*e.g.*, the rectifying procedure). The second one is the lack of theoretical or experimental evidences that the strategy proposed by the authors overcomes the *DCS* and *DCS++* strategies. Lastly, there is no experimental study about how the approach handles data skew and load balancing issues.

Table 3.2 presents a comparative of the windowing approaches considered in this work. The approaches are compared according to the following features:

- Windowing search space: the manner in which the approaches generate the candidate pair comparisons;
- #MapReduce jobs: the number of MapReduce jobs performed by the approach;
- Load balancing: indicates whether (or not) the approach presents load balancing handling mechanisms;

Table 3.2: Comparative table of windowing models

Approaches | Criteria	**Windowing Search Space**	**#MapReduce Jobs**	**Load Balancing Handling**	**Reproducibility**
SNM [22]	Slides a fixed window	ZERO	NO	YES
DCS++ [16]	Slides a fixed window	ZERO	NO	YES
RepSN [30]	Slides a fixed window	TWO	YES	YES
Partition-Sort-Map-Reduce [24]	Slides an adaptive window	ONE	NO	NO

- Reproducibility: indicates whether (or not) the approach can be duplicated from its publication.

As it can be seen, although Table 3.2 presents one MR-based adaptive indexing method (in relation to the windowing search space), the load balancing feature was not yet explored by existing adaptive windowing approaches in the literature.

3.4 Spark-based EM Indexing Methods

Nevertheless, the usage of Hadoop MapReduce as a platform for EM has some drawbacks [82]. Hadoop MapReduce is not designed for an interactive usage. It is made for Big Data batch processing since it lacks the ability to cache intermediate results (in main memory) to be used for multiple alternatives. For these reasons, it is common to find models of MR-based EM approaches using more than one MapReduce job [50].

In a previous work [62], it was proposed an approach for the MapReduce-based Duplicate Count Strategy. The solution provides an efficient parallelization of the *DCS++* method [28] by using multiple MR jobs. However, the solution presented in [62] neither considered RDDs nor broadcast variables[5], since MR does not provide such features. Their usage can decrease the number of jobs employed by MR approaches. In Chapter 5, we compare the MR-based

[5]Broadcast variables allow the program execution to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

DCS++ (*MR-DCS++*) approach [62] (state of the art) with our work in an experimental evaluation.

Apache Spark emerged as an alternative to provide features such as interactive usage and in-memory (main memory) data structures and broadcast variables [4, 10, 78, 79, 81, 84]. However, to the best of our knowledge, no work involving Spark-based adaptive windowing technique for EM has been proposed.

3.5 Map-matching Bus Trajectories

Large volumes of geospatial-temporal data including GPS sequences have become popular data sources for various machine learning and pattern recognition algorithms. These algorithms, when designed to execute the Map-matching objects represented in a digital map, tend to be optimized in order to achieve a desired numerical calculation speed of the map-matching and polling frequency of the location data [1, 11, 41]. Most map-matching related work deals with high-frequency GPS polling, which is GPS routing commonly used by automobiles. The authors of [73] provide a review of various traditional high-frequency GPS polling map-matching techniques. Most of the techniques seem to have been designed considering Hidden Markov Models (HMM) using the Viterbi algorithm introduced by works such as [42, 44, 51, 85, 102]. Lately, more sophisticated techniques capable of mapping noisy and sparse GPS sequences of vehicles into road sequences have been proposed. The survey presented in [106] describes some of these sophisticated techniques and how they benefit from the usage of HMM, such as the ones proposed in [25] and [88].

Less map-matching literature is devoted to low-frequency GPS polling. Many of the sophisticated techniques which deal with high-frequency GPS polling, such as [8, 72], rely heavily on the immediate past of the trace, heavily weighting the probability that a vehicle has stayed on the same road. However, the techniques which deal with low-frequency GPS polling cannot benefit from such probability strategy since a vehicle can perform two or more turns between polls. For this reason, the works [103] and [99] present techniques to address the low-frequency GPS polling map-matching problem by first collecting the subset of similar matches between the bus (GPS) trajectory and the route. Then, they use a coordinate point to indicate the direction of the bus trajectory and search for the most similar route to that bus

(GPS) trajectory. The authors of [103] note that there could be many routes between two coordinates and suggest the usage of the shortest path (using Dijkstra's algorithm) between them as the most similar route to the bus (GPS) trajectory.

Several researchers have studied the algorithmic logic used for choosing correct trajectories in map-matching algorithms, which is complicated due to the presence of overpasses, sharp turns, errors in measurements, and ambiguity [73]. In [73], the authors noted that the wrong associations (matching) between trajectories and the routes may lead to a sequence of bad matches. In a probabilistic framework that handles multiple hypotheses, the authors of [71] utilized topological data such as road connection, direction, and road facility information to identify trajectories.

Regarding map-matching of bus routes, it is generally a trivial task to know where a bus should go. Basically, it consists in following the bus assigned route and programmed arrival times when the bus is supposed to travel along that route. In the GTFS terminology, the bus has a route identifier. Buses may have to make detours for various reasons, but the general assumption is that they will follow their route. This difference simplifies the map-matching process from projecting onto a graph the linkage of a road network with the polylines as defined by the GTFS data. In this sense, the authors of [8] developed a system architecture that estimates the route shapes depicted by the bus GPS traces aiming to map-match and complete trip prediction on the GPS points. They also considered high-frequency GPS data and HMM in the development of the map-matching and trip prediction algorithms. The authors of [86] proposed a method to improve the positioning accuracy by exploiting the information of speed bumps readily available in bus parking garages to tune its position and velocity. This matching algorithm provides an accuracy of around five meters, which matches the detected landmark to the right bumps.

As mentioned in Subsection 2.3.2, none of the aforementioned techniques are able to deal with multiple shapes (with different starting geo-spatial points) related to the same bus route. Nevertheless, the work presented in [76] is close to ours. It shows a method for inferring transit network topology from commonly available data feeds. It makes use of a Bag-of-Roads strategy which is a sparse vector containing the number of road segments traversed by a bus b, where its i-th element denotes the frequency of bus b traversing the road segments. Then, it selects the top-k nearest predefined routes according to the Euclidian

Table 3.3: Comparative table of Map-matching algorithms

Works | Criteria	***Map-matching object***	***Indexing type***	***Frequency of GPS polling***	***Parallel approach***
Yang J, et al. (2005) [103]	Digital map	NONE	LOW	NONE
Wu D, et al. (2007) [99]	Digital map	NONE	LOW	NONE
Dong W, et al. (2009) [25]	Digital map	NONE	HIGH	NONE
Thiagarajan A, et al. (2009) [88]	Digital map	NONE	HIGH	NONE
Biagioni J, et al. (2011) [8]	Digital map	Attribute-based indexing (roads)	LOW	NONE
Tan G, et al. (2014) [86]	Digital map	Attribute-based indexing (roads)	HIGH	NONE
Raymond R, et al. (2016) [76]	Single Predefined Trajectories (shapes) per route	Top-k nearest predefined routes	HIGH	NONE

distance and Cosine similarity. Their results show good performance using routing between the bus trajectory stops as a form of map-matching. Although the authors of [76] provide interesting techniques that are able to detect the correct route performed by a bus, their technique is also not able to deal with multiple shapes (with different starting geo-spatial points) related to the same route (as exemplified in Subsection 2.3.2). In Chapter 6, we show experimentally their lack of robustness in treating the problem of multiple shapes referring the same route effectively.

Table 3.3 presents a comparative table of the bus trajectory Map-matching algorithms considered in this work. The algorithms are compared according to the following features:

- Map-matching object: the object compared (to match) with the GPS trajectory of the vehicle. For instance, if the algorithm considers a Digital map or (single or multiple) predefined trajectories per route to match with the GPS trajectory of the vehicle;
- Indexing type: the indexing type utilized by the algorithm to reduce the search space. For instance, if the search space is generated by the employ of top-k nearest routes, attribute-based indexing, among others;
- Frequency of GPS polling: Frequency of each vehicle GPS polling. For instance, if the frequency is high or low;
- Parallel approach: the parallel approach utilized by the work. For instance, None, MapReduce, Spark or others.

As it can be seen, although Table 3.3 presents two types of Map-matching objects, *i.e.*, Digital map and single predefined trajectories (shapes) per route, and two types of indexing methods, there are at least three characteristics that were not properly exploited by the algorithms presented in this subsection. The first is regarding the map-matching of the GPS trajectory of the vehicle with multiple predefined trajectories (shapes) per route. The second is regarding the application of new indexing strategies in order to perform the Map-matching task within a short time interval, and still maintain a high Map-matching quality. Finally, the third is regarding the efficient parallelism of the algorithms. These features were not yet exploited by existing map-matching works in the literature.

3.6 Final Considerations

The results presented in this chapter show in detail a bibliographical review regarding the main related work found in the literature and their respective proposals. Essentially, the works are divided into five groups: works describing Entity Matching and its applications, studies that investigate EM indexing methods, works involving MapReduce-based EM indexing methods, works involving Spark-based EM indexing methods and works that utilize Map-matching techniques to solve the specific problem of matching bus trajectories regarding the presence of multiple shapes (with different starting geo-spatial points) related to the same route. In the following, Chapters 4 and 5 will present the contributions of this work with respect to the execution of the EM task in parallel based on MapReduce and Spark. Chapter 6 will present the contributions of the work regarding the execution of the Bus Trajectories map-matching task in parallel based on Spark.

Chapter 4

An Efficient MapReduce-based Approach for the Multi-pass *Duplicate Count Strategy*

As mentioned in Section 2.1, the SNM presents a critical performance disadvantage due to the fixed and difficult to configure window size: if it is selected too small, some duplicates might be missed. On the other hand, a too large window results in unnecessary comparisons. To overcome this disadvantage, the authors of [28] proposed an efficient SNM variation, denoted as Duplicate Count Strategy (*DCS*). The method follows the idea of increasing the window size in regions of high similarity and decreasing the window size in regions of low similarity. They also proved that their improved variant of *DCS*, known as *DCS++*, overcomes the performance of traditional SNM by obtaining at least the same matching results with a significant reduction in the number of entity comparisons. For this reason, this chapter presents efficient MapReduce-based approaches for the *DCS++*.

Even with the significant advances in the SNM design, EM remains a critical task in terms of performance when applied to large data sources. Thus, this chapter presents the contributions regarding a MR-based approach capable of combining the efficiency gain achieved by the *DCS++* method with the benefit of efficient parallelization of data-intensive tasks in cluster infrastructures. The aim is to decrease even more the execution time of EM tasks performed with the SNM (briefly, combine the best of the two worlds). In this sense, we make the following contributions in this chapter:

- We propose the multi-pass **M**ap**R**educe-based **D**uplicate **C**ount **S**trategy (*MultiMR-DCS++*), a MR-based approach that provides an efficient parallelization of the multi-pass *DCS++* method [28] by using multiple MR jobs and applying a tailored data replication during data redistribution to allow the resizing of the adaptive window. The approach also addresses the data skewness problem with an automatic data partitioning strategy that is combined with *MultiMR-DCS++* to provide a satisfactory load balancing across the available nodes;

- The approach *MultiMR-DCS++* is evaluated against the MR-based multi-pass SNM state-of-the-art approach *RepSN* [50] (fixed window). The evaluation shows that our approach provides a better performance by diminishing the overall EM execution time. The experiments are performed using a real cluster environment and real-world data sources.

Initially, we proposed an approach for the single-pass MapReduce-based Duplicate Count Strategy. The work is presented in [62] and provides an efficient parallelization of the *DCS++* method [28] by using multiple MR jobs. However, this solution presented in [62] does not support the performance of a multi-pass *DCS++* variant within the same MR jobs. This means that, to perform the multi-pass *DCS++*, the MR process must be repeated for each pass. In this chapter, we present a more sophisticated model in terms of robustness and extensibility that addresses both the single- and multi-pass *MR-DCS++* without the need of the MR process serialization.

4.1 General Multi-pass MR-based DCS++ Workflow

Occasionally, when dealing with dirty (inaccurate, incomplete or erroneous) input data, it is not sufficient to use a single blocking key to generate satisfactory EM results. Even using an adaptive window size, if the blocking is not ideal, it is quite common that the similarity between distant entities remains considerable. Multi-pass SNM addresses this problem by employing multiple blocking keys (*e.g.*, using multiple entity attributes) and matching passes in order to combine the duplicates identified by the different passes. One advantage of using multiple passes is that, if the blocking is not ideal, individual passes can be done

with relatively small window sizes and consequently improve the EM effectiveness and efficiency [38].

A basic strategy when thinking about multi-pass DCS++ is to utilize a naive approach to implement the multiple passes, *i.e.*, run the single-pass *MR-DCS++* p times and employ one of the p different blocking key functions per pass. However, this approach requires p scans of the input data source and introduces additional overhead for executing the p MapReduce processes.

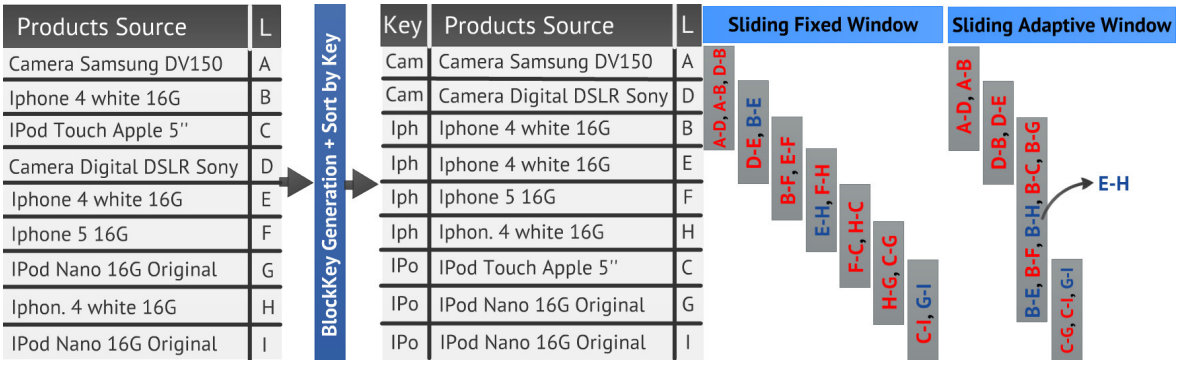

Figure 4.1: Overview of the multi-pass *MR-DCS++* matching process workflow.

In practice, new challenges arise when proposing adaptive MR-based EM approaches. An interesting line of reasoning when we deal with MR-based EM is to define an efficient MR approach (with load balancing handling) by knowing previously the number of entity comparisons generated by the serialized blocking (windowing) method. In this sense, how do we define an efficient MR approach when the blocking (windowing) method adapts according to the duplicate detection rate (like the *DCS++* strategy)? Also, how do we assign entity comparisons to the proper reduce tasks with load balancing handling without knowing all the necessary entity comparisons? Finally, how can we build a windowing model that is capable of performing multiple passes in the same MR job execution? Our better answer to solve these research questions is to propose a Multi-pass MR-based *DCS++* approach for EM processing.

Thereby, our approach uses three MR jobs as illustrated in Figure 4.1. The first MR job sorts all the entities (SNM requirement) and splits them to generate sorted partitions containing approximately the same amount of entities according to each pass. The aim of this job is to enable the generation of an approximately equal number of entities comparisons

per sorted partition. The idea is to provide an effective load balancing by sliding the adaptive window over the same number of entities in each EM parallel execution unit (reduce task). After that, the job emits the first and last entities of each sorted partition to the second MR job. In turn, the second MR job performs early comparisons between the first and last entities of each partition aiming to define the partitions that must be attached to each other, and thus, enable the growth of the adaptive window without losing relevant comparisons. Finally, the third MR Job performs the adaptive window sliding to detect the duplicate pairs.

In the remainder of this section, each MR job is detailed with a running example involving 12 entities (product descriptions) and 3 blocking keys in each pass.

4.1.1 First MR Job: Sorting and Selecting

SNM assumes an ordered list of entities based on the chosen blocking key. Since the input data source consists of an unordered entity collection, a preprocessing job is necessary to sort entities p times (according to the number of passes) and select the boundary entities of the sorted partitions generated for each pass (more details in Section 4.1.2). To this end, the map function determines the p blocking keys for each entity and outputs p key-value pairs $(pass \cdot blockKey, entity)$ with a map output key = $(pass \cdot blockKey)$, where $pass$ means the (window) pass index, and value = (entity), as shown in lines 3 to 7 of Algorithm 1. The key-value pairs are partitioned based on the average number of entities per reduce task, as follows:

$$\xi = p * \frac{\Sigma|\Pi|}{r} \tag{4.1}$$

where p is the number of passes, Π represents the set of partitions of Source R and r corresponds to the number of reduce tasks. The average number of entities (ξ) is necessary to treat the skewness problem by allocating the same number of entities to each reduce task. The reduce task's key-value pairs are sorted by the key (according to SNM) and the reduce function generates the additional output partitions with the sorted entities. The reduce function also generates an output with the boundary entities of each sorted partition according to the proper pass. It selects the last entity of the sorted partition 0, the first and the last entities of intermediate sorted partitions and the first entity of the last sorted partition according to each pass (lines 16 to 20 of Algorithm 1). Note that the sorted partition index value is equal

to the reduce task index.

Figure 4.2 shows the computation of the sorting and selecting MR job. The 12 entities ($A - L$) of source R (initially unsorted) are divided (according to the number of map tasks available) into three input partitions (Π): 0, 1, and 2. Each entity has two blocking keys, the first (BK_1^*) is generated with the three first letters of the product description (cam, iph, and ipo) and the second corresponds to the two first letters of product manufacturer (apple = ap, samsung = sa, sony = so). The map output keys of E is $0.cam$ and $1.so$ because E's blocking key in the first pass is cam and in the second pass is so. The first key (of E) is assigned (by the combiner function) to the second reduce task because the amount of entities assigned to the first reduce task for the first pass reached the average number of entities per reduce task ($\xi = \frac{4+4+4}{3} = 4$). The second key (of E) is assigned to the third reduce task because the amount of entities assigned to the first and second reduce tasks for the second pass reached the average number of entities per reduce task. The second reduce task generates an additional output sorted partition with the sorted entities E, F, G and H of the first pass and J, K, L and B of the second pass. An output with the sorted partition boundary entities E, H (first pass), J and B (second pass) preceded by 0 or 1 as the pass index and 1 as the corresponding sorted partition index (the second reduce task index) is also generated.

4.1.2 Second MR Job: MultiPAM Partition Allocation Matrix Generation

As mentioned in Section 2.1, an important step of the second MR job is to detect where the boundary pairs are located in the sorted partitions. This job is responsible for indicating the sorted partitions that must be replicated to the proper reduce task in such a manner that the *DCS++* adaptive window can be performed without changing the duplicate detection rate presented by the serialized algorithm. To achieve this, the algorithm uses a boundary pair inference strategy to know the exact moment to stop indicating sorted partitions.

The output of this MR job is the Multi-pass Partition Allocation Matrix (MultiPAM). The key idea of the MultiPAM generation is to process the boundary entities of each sorted partition which shares the same pass index. The MultiPAM is a $Pass$ x $m - 1$ x $m - 1$

Algorithm 1 First MR Job: Sorting and Selecting

1: **function** map_configure($jobConf$)
2: $p \leftarrow getNumberOfPasses()$;
3: **function** map($k = unused, v = entity$)
4: **for** $j = 1 \rightarrow p$, $j{+}{+}$ **do**
5: $pass \leftarrow j - 1$; //pass index
6: $blockKey \leftarrow generateBlockingKey(j, entity)$;
7: **emit**($k = pass \cdot blockKey, v = entity$);

▷ ———————————————————————————————————

Shuffle: combines the records (with a combiner function) under the average ($\xi = p * \frac{\Sigma|\Pi|}{r}$) number of entities per reduce task constraint, partitions the records by the first part of the (combined) key (redIndex) and sorts the records by the entire key.

———————————————————————————————————

8: **function** reduce_configure($jobConf$)
9: $numPartitions \leftarrow getNumberOfPartitions()$; ▷ *//getNumberOfPartitions* - returns the number of partitions generated from the data source
10: **function** reduce($k = redIndex \cdot pass \cdot blockKey, list(v) = list(entity)$)
11: $entityIndex \leftarrow 0$;
12: **for each** $entity \in list(entity)$ **do**
13: *//To HDFS*
14: **additionalOutput**($pass \cdot blockKey \cdot entityIndex, entity$)
15: $entityIndex{+}{+}$;
16: **if** $redIndex \neq 0$ **then**
17: $firstE \leftarrow getFirstEntity(list(entity))$;
18: **emit**($k = pass \cdot redIndex, v = firstE$); ▷ *//redIndex* - the reduce task index
19: **if** $redIndex < numPartitions - 1$ **then**
20: $lastE \leftarrow getLastEntity(list(entity))$;
21: **emit**($k = pass \cdot redIndex, v = lastE$);

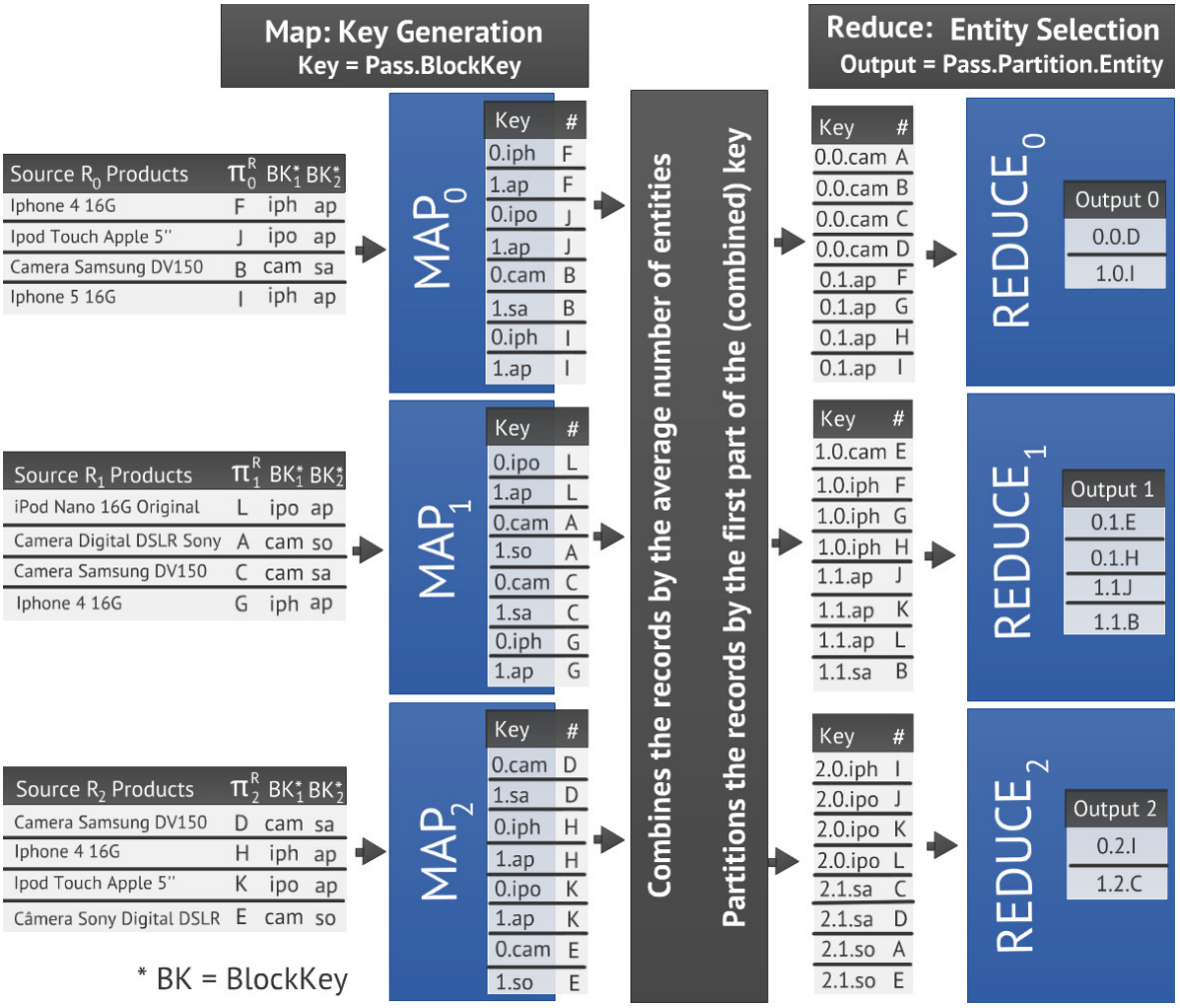

Figure 4.2: Example dataflow for computation of the entity sorting with two window passes.

matrix (where m is the number of map tasks) that specifies which sorted partitions must be replicated and attached to other sorted partitions according to the respective pass index.

The MultiPAM computation using MR is straightforward. The map function determines the reduce task that will process each entity and outputs a key-value pair with a composite map output key = (*RedIndex* · *Pass* · *PartitionIndex*) and the value = (entity). The key-value pairs are partitioned based on the $RedIndex$ to ensure that the first entity of each partition is assigned to each reduce task whose *RedIndex* < *PartitionIndex*, as shown in lines 8 and 16 to 18 of Algorithm 2. The last entity of each partition is assigned to the reduce task whose $RedIndex = Pass + 1$ (if $Pass + 1$ refers to an out of bounds reduce task then $RedIndex = 0$), as shown lines 21 to 24 of Algorithm 2. In the shuffle phase, the reduce task's key-value pairs are grouped by the first two parts of the key (RedIndex · Pass) and sorted by the entire key. Thereafter, the reduce function performs the comparisons between the entity whose *RedIndex* = *PartitionIndex* and each entity sharing the same pass index ($Pass$) and whose *RedIndex* <> *PartitionIndex* aiming to find the comparisons that return a similarity value below Φ_{min} (lines 28 to 30 of Algorithm 2). For each comparison performed where the similarity value is above Φ_{min}, the reduce function outputs triples in the form ($Pass$, partition_target -> partition_origin, similarity). The reduce function stops comparing when a similarity value below Φ_{min} is found (lines 31 to 34 of Algorithm 2).

Continuing with the running example, in Figure 4.3, the map function output key of D is 0.0.0 because D is assigned to the reduce task whose index is 0, belongs to pass index 0 and its partition is equal to 0. This key is assigned to the first reduce task that processes comparisons between D and the first entity of each partition belonging to pass index 0 aiming to find the comparison that returns a similarity value below Φ_{min}. Thus, D is compared to E (the first entity of the second partition) and I (the first entity of the third partition). Since the comparison between D and E returns a similarity value above Φ_{min}, in the first pass (0), E's partition (1) must be replicated and attached to the D's partition (0) to allow the growth of the adaptive window without any possibility of comparison loss. Then, the MultiPAM is updated with the new assignment (multiPAM[0,1,0] = 1). Note that the comparison between D and I returns a similarity value below Φ_{min}. This indicates that E's partition contains the boundary pair and thus it is no longer necessary to perform new comparisons.

Algorithm 2 Second MR Job: MultiPAM Generation

1: **function** map_configure($jobConf$)
2: $numPartitions \leftarrow getNumberOfPartitions()$;
3: $reduceTasks \leftarrow jobConf.numReduceTasks()$;
4: $entityTurn \leftarrow 0$; ▷ // *entityTurn* - the turn of the entity (identifies the first or last entities of a partition)
5: **function** map($k = unused, v = pass \cdot partitionIndex \cdot entity$)
6: **if** $partitionIndex = 0$ **then**
7: $redIndex \leftarrow getNextReduceTask()$; ▷ //*getNextReduceTask* - returns the reduce task with the fewest number of assigned entity comparisons
8: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
9: **else**
10: **if** $partitionIndex = numPartitions - 1$ **then**
11: **for** $j = 0 \rightarrow partitionIndex - 1, j{+}{+}$ **do**
12: $redIndex \leftarrow getNextReduceTask()$;
13: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
14: **else**
15: **if** $entityTurn = 0$ **then**
16: **for** $j = 0 \rightarrow partitionIndex - 1, j{+}{+}$ **do**
17: $redIndex \leftarrow getNextReduceTask()$;
18: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
19: $entityTurn{+}{+}$;
20: **else**
21: **for** $j = 0 \rightarrow partitionIndex - 1, j{+}{+}$ **do**
22: $redIndex \leftarrow (pass + 1)$;
23: **if** $redIndex > reduceTasks - 1$ **then**
24: $redIndex \leftarrow 0$;
25: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
26: $entityTurn{-}{-}$;

▷ ---

Shuffle: sorts records by (redIndex · pass) and shuffles them.

27: **function** reduce($k = redIndex \cdot pass \cdot partitionIndex, list(v) = list(entity)$)
28: $firstE \leftarrow getFirstEntity(list(entity))$;
29: **for** $j = 1 \rightarrow list(entity).size(), j{+}{+}$ **do**
30: $simValue \leftarrow$ **matches**($firstE, list(entity).get(j)$)
31: **if** $simValue < \Phi_{min}$ **then**
32: $break$; //STOP
33: **else**
34: $multiPAM[pass][redIndex][j] \leftarrow 1$;

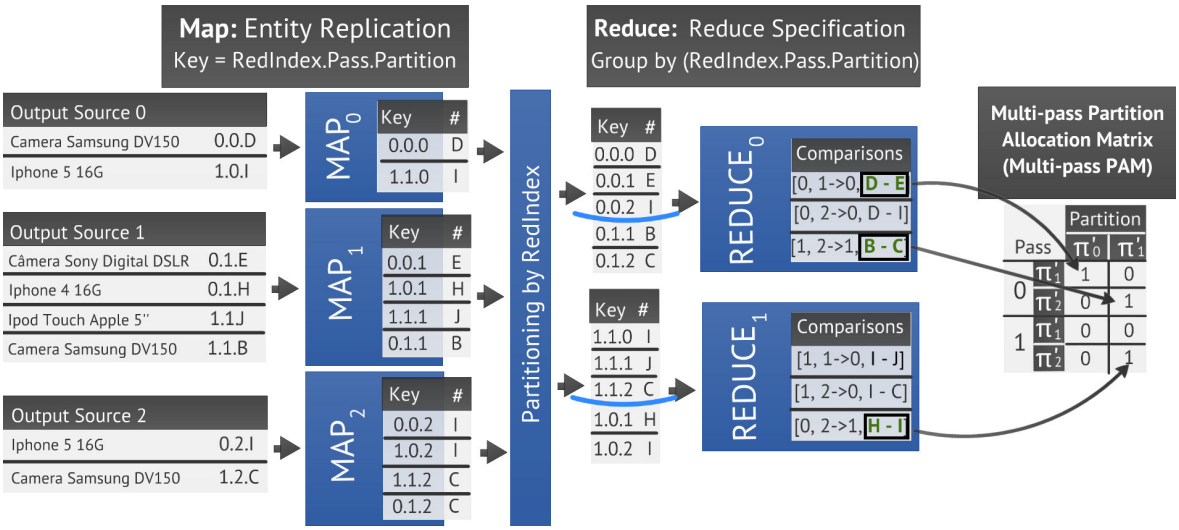

Figure 4.3: Example dataflow for computation of the Partition Allocation Matrix (PAM) for two window passes.

4.1.3 Third MR Job: Multi-pass MR-DCS++

The third job performs the distributed *DCS++* denoted as *MultiMR-DCS++*. It assigns the sorted partitions to the reduce tasks in such a manner that each reduce function can perform the *DCS++* adaptive windowing multiple times without missing any relevant comparison. To be more specific, our approach uses the following key ideas:

- *MultiMR-DCS++* assigns and attaches sorted partitions to perform the *DCS++* adaptive windowing to each pass without any loss of relevant comparisons. To prevent the data skewness problem (memory bottlenecks), the sorted partitions are set to have the same number of entities;

- *MultiMR-DCS++* improves the load balancing by fixing the same number of entities to each sorted partition. Furthermore, for each reduce task, *MultiMR-DCS++* fixes the maximum number of window's slides per pass according to the fixed number of entities in each sorted partition aiming to increase even more load balancing.

The execution of *MultiMR-DCS++* makes use of the MultiPAM as well as the composite map output keys. Each map function generates a well-defined composite key that (together with the associated pass and partition) allows the partition to be assigned to the proper reduce tasks. The composite key thereby combines information about the target reduce task(s) and

the entity itself. ***MultiMR-DCS++***'s mappers output key-value pairs with key = ($RedIndex \cdot Pass \cdot PartitionIndex \cdot EntityIndex$) and value = (entity). The reduce task index has a value between 0 and r - 1 and is used by the MR function *part* to perform assignments to the reduce tasks. Since the reduce index, pass index and partition index are part of the key, the MR function *group* takes only the reduce index into account and the key-value pairs are sorted by the entire key, it is ensured that each reduce function only receives entities in the correct order.

In the map phase, each map task m reads the MultiPAM and verifies to which reduce task the entities of its corresponding sorted partitions must be assigned. The number of replications for each entity is defined according to the number of reduce tasks indicated by the MultiPAM. For example, if the MultiPAM indicates that partition 3 (of the pass index 0) must be assigned to the reduce tasks 1 and 2, the entities belonging to partition 3 (of the pass index 0) must be replicated twice (*i.e.*, to the reduce tasks 1 and 2). For the purposes of memory bottlenecks avoidance and load balancing optimization, the map function allocates the tasks generated by the odd passes from the first to the last reduce task and the even passes from the last to the first reduce task. This is calculated by $pass$ *mod* 2 as shown in line 30 of Algorithm 3.

In the reduce phase, each reduce task r receives the assigned entities grouped by the entire key. Since the key has the information of the pass index, partition index and entity index, the entities are placed in the correct order in such a manner that ***MultiMR-DCS++*** can process the adaptive window slide in each pass, as shown in Algorithm 3. The size of the first window is passed as a context parameter (w) to the reduce function. Its value is increased according to the definitions (of DCS++) discussed in Section 2.2. However, for load balancing purposes, the window only slides until the last entity whose partition index is equal to the reduce task index. Since after every window's slide the window size is set to the initial value, there is no problem on splitting the window sliding among the reduce tasks.

In our running example, as shown in Figure 4.4, the MultiPAM (generated in the second MR job) indicates that the sorted partition Π'_1 (for the pass index 0) must be attached to Π'_0 (the process of reading the MultiPAM is by line for each pass index) and Π'_2 has to be attached to Π'_1 (see Figure 4.3). The map task replicates the entities of Π'_1 and Π'_2 once. The map task also sets to the $redIndex$ of each replicated entity key the same value of the $pass$

Algorithm 3 Third MR Job: Multi-pass MR-DCS++

1: **function** map_configure($jobConf$)
2: $multiPAM \leftarrow getMultiPAM()$;
3: $reduceTasks \leftarrow jobConf.numReduceTasks()$;
4: **function** map($k = unused, v = pass \cdot blockKey \cdot entity$)
5: $partitionIndex \leftarrow getCurrentMapIndex()$; ▷ *//getCurrentMapIndex* - returns the index of the current map task
6: **if** $partitionIndex = 0$ **then**
7: **if** $pass\ mod\ 2 = 0$ **then**
8: $redIndex \leftarrow 0$;
9: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
10: **else**
11: $redIndex \leftarrow reduceTasks - 1$;
12: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
13: **else**
14: **if** $partitionIndex = reduceTasks - 1$ **then**
15: **if** $pass\ mod\ 2 = 0$ **then**
16: $redIndex \leftarrow partitionIndex$;
17: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
18: **for** $j = partitionIndex - 1$ **downto** $0, j--$ **do**
19: **if** $multiPAM[pass][partitionIndex][j] = 1$ **then**
20: **emit**($k = j \cdot pass \cdot partitionIndex, v = entity$);
21: **else**
22: $redIndex \leftarrow 0$
23: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
24: **for** $j = 1$ **to** $reduceTasks - 1, j++$ **do**
25: **if** $multiPAM[pass][partitionIndex][j] = 1$ **then**
26: **emit**($k = j \cdot pass \cdot partitionIndex, v = entity$);
27: **else**
28: $redIndex \leftarrow partitionIndex$
29: **emit**($k = redIndex \cdot pass \cdot partitionIndex, v = entity$);
30: **if** $pass\ mod\ 2 = 0$ **then**
31: **for** $j = partitionIndex - 1$ **downto** $0, j--$ **do**
32: **if** $multiPAM[pass][partitionIndex][j] = 1$ **then**
33: **emit**($k = j \cdot pass \cdot partitionIndex, v = entity$);
34: **else**
35: **for** $j = partitionIndex + 1$ **to** reduceTasks-1, $j++$ **do**
36: **if** $multiPAM[pass][partitionIndex][j] = 1$ **then**
37: **emit**($k = j \cdot pass \cdot partitionIndex, v = entity$);

▷ ---

Shuffle: partitions the records by the first part of the key (redIndex), groups by the two first parts of the key (redIndex · pass), and sorts the records by the entire key.

38: **function** reduce($k = redIndex \cdot pass \cdot partitionIndex \cdot entityIndex, list(v) = list(entity)$)
39: **do** $performDCS++(list(entity))$;
40: **while** $partitionIndex = getCurrentReduceTask()$;

and $partitionIndex$ indicated by the MultiPAM. For example, E belongs to partition 1 (of the pass index 0) and thus is assigned to the reduce task 1, and since the MultiPAM indicates that Π'_1 has to be attached to Π'_0 (of the pass index 0), E is replicated and assigned to the reduce task 0.

Once all entities are positioned, the reduce function starts sliding the window. In our example, we use Φ_{max} = 0.9 (the similarity threshold that indicates if a pair of entities is similar), α = 0.5 (the minimum interval value that the window size can increase and keep performing relevant comparisons) which implies that $\Phi_{min} = 0.9 - 0.5$ = 0.4 (the minimum threshold that indicates if a pair is out of the limits of the boundary pair). In practice, the Φ_{max} and α values are defined according to the characteristics of the data. We also use w = 3 which implies that, according to the *DCS++* strategy, the increasing condition threshold due to the duplicate detection rate is $\Phi_{ddr} = \frac{1}{w-1} = \frac{1}{3-1} = 0.5$.

Thus, in the reduce task 0 and pass index 0, the first window generated covers entities A, B, and C. This results in the following comparisons: A-B and A-C. Both comparisons are regarded as non-matches and since no duplicate entities are identified, there is no need to increase the window. Therefore the window slides to the next entity (B). From B, the next comparison B-C is regarded as a match (framed). Thus, the following relation is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of already detected duplicates within the window and c is the number of comparisons already done also within the window, then w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (C). Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of C. Now, w covers B, C, D and E; the new comparisons generated are B-D and B-E. Since B-D is regarded as a match, the w increase test is loaded once more resulting in true ($\frac{2}{2} \geq 0.5$) and the window is increased by two again. This time, the window is increased from D, the last duplicate found. Now, w covers B, C, D, E and F; B is compared with the rest of the entities within the window. The window is no longer increased due to the lack of new matches. After that, w is set to the initial value (3) and the window slides to the next entity (C). Although the windows starting with C and D present the partition index equal to the reduce index, they are skipped due to the transitive closure related to B (B is similar to C and D). The thick curve in Figure 4.4 indicates the end of the window's sliding. Also note that the window starting with E is executed in the reduce task 1.

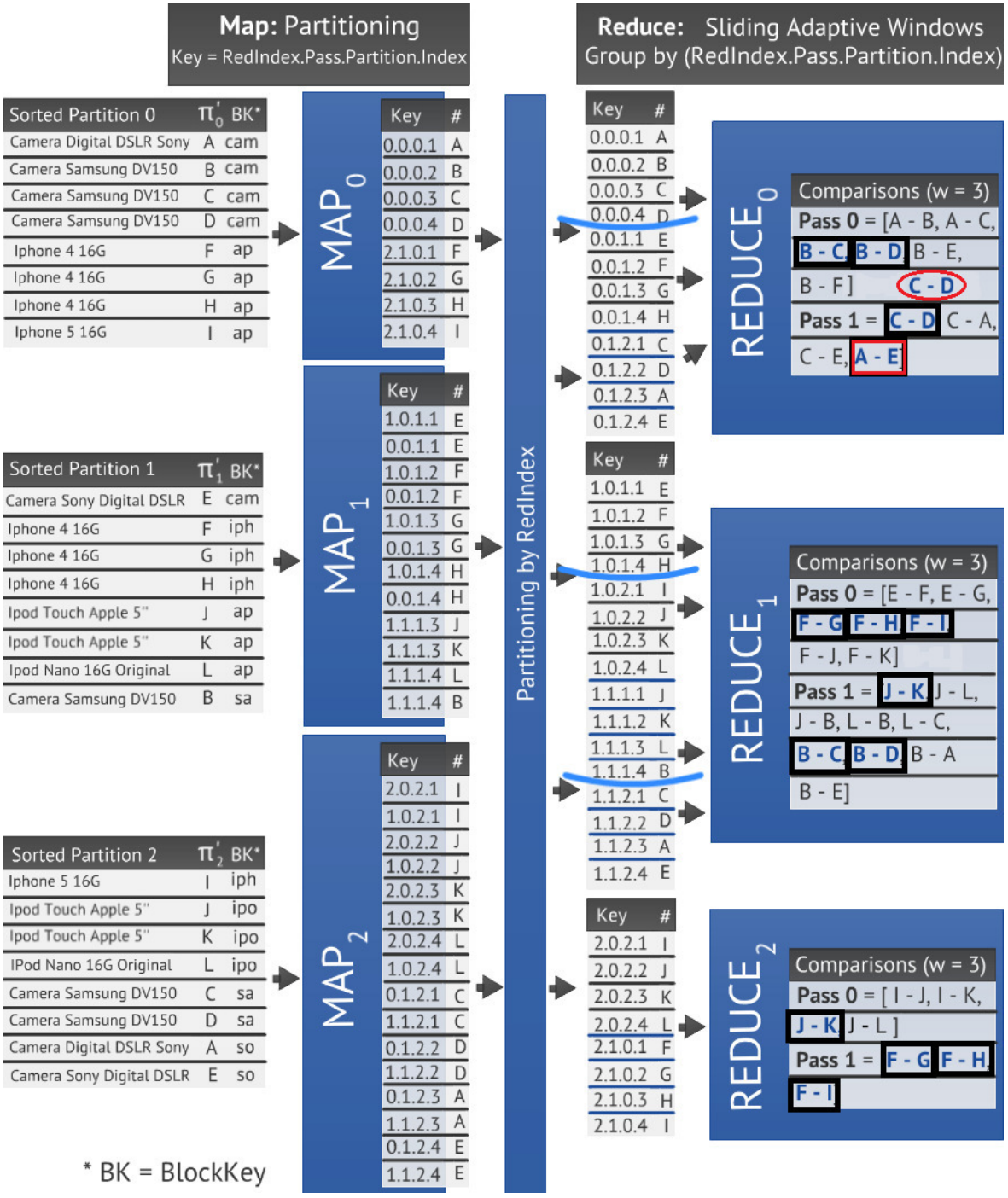

Figure 4.4: Example dataflow for the Multi-pass *MR-DCS++* strategy with $w_{initial} = 3$.

The same strategy is performed in the pass index 1 of the reduce task 0. Note that, in the second pass, a new match that was not found in the first pass was detected, *i.e.*, A-E, showing the importance of ideal blocking keys selection. If the blocking is not ideal, this example shows that the multi-pass SNM can be an important resource to improve the similarity detection effectiveness. Moreover, our evaluation shows that multi-pass SNM can achieve high match quality even with smaller window sizes.

4.2 Evaluation

In the following, we evaluate the single- and multi-pass *MR-DCS++*[6] against the single- and multi-pass *RepSN*[7] approaches, regarding three critical performance factors: degree of skewness (Section 4.2.1), the efficiency in the usage of the nodes available (n) in the cluster environment (Section 4.2.2) and the trade-off between the matching quality and execution time (Section 4.2.3). In each experiment, we broadly evaluate the algorithms aiming to investigate their robustness against data skew, how they can scale with the increasing of the number of available nodes and their robustness in maintaining the EM quality while their execution time decreases.

We ran our experiments on a 20-node HP Pavilion P7- 1130 cluster. Each node has one Intel I5 processor with four cores, 4GB of RAM and one 1TB of hard disk. Thus the cluster consists of 80 cores and 20 disks. Each node was configured with Windows 7, 64-bit, JAVA 1.6, cygwin, and Hadoop 0.20.2. Each node runs at most two map and reduce tasks (1 core/1GB of RAM for each task) in parallel (default configuration of Hadoop). The replication factor used in the HDFS was configured with "3x" (by default).

We used three real-world data sources. The first data source (DS1) is a sample of the Ask's database that contains about 214,000 ($8.8{\cdot}10^7$ bytes) question records. The second data source (DS2) is by an order of magnitude larger and contains about 1.46 million ($2.96 \cdot 10^8$ bytes) publication records. The third data source (DS3) is small and contains about 7,800 ($1.6 \cdot 10^6$ bytes) DBLP and Google Scholar publication records (based on [53]) presenting the following attributes (fields): id, title, authors and publication year. This third data source

[6]The codes and data sources are available in https://sites.google.com/site/demetriomestre/activities

[7]Executed using the deduplication tool (Dedoop) available in the author's homepage at http://dbs.uni-leipzig.de/howto_dedoop

was utilized due to the absence of a gold standard for DS1 and DS2 necessary to evaluate the approaches' matching quality.

For the experiments whose purpose was to investigate the execution time of the approaches, the similarity between two entities was computed using the Jaro-Winkler distance [48] of their comparing attributes (*i.e.*, the question for DS1 and the publication title for DS2). Those pairs with a similarity $\Phi_{max} \geq 0.7$ were regarded as matches. We utilized α = 0.4 since the comparisons with a similarity distance below 0.3 ($\Phi_{min} = 0.7 - 0.4$) have proven to be unpromising. For the third experiment, two entities were regarded as matches if their comparing attributes (*i.e.*, publication title) have a q-gram similarity $\Phi_{max} \geq 0.75$. Since EM quality analysis is important in this experiment, the matcher was modified due to the better accuracy provided by the q-gram strategy for these kinds of attributes. In this case, we utilized α = 0.5 since the comparisons with a similarity distance below 0.25 ($\Phi_{min} = 0.75 - 0.5$) have proven to be unpromising.

4.2.1 Robustness: Degree of skewness

In this experiment, we study the robustness of the load balancing approaches to handle data skew. In this case, data skew occurs when there is a region of high similarity that enables the increasing of the adaptive window. This window increasing generates new entity comparisons and thus is supposed to leverage the EM execution time during the window sliding on that region. Since the adaptive window increases only when a new duplicate entity is found, for this study, we control the degree of data skew by modifying the number of duplicates in the data source. The aim is to replace non-duplicated entities by duplicated ones purposely according to an input percentage. Given a fixed number of entities e in the data source, the number of duplicate entities d is equal to $d = e \cdot s$, such that $s \geq 0$ is the percentage that represents the degree of skew (the degree of duplication). To exemplify, suppose we have 100 entities (e) and we set s = 0 (no skewness) then there are no duplicate entities in the data source. In turn, if we want 10% of duplicate entities, then we set s = 0.1, and thus, $d = 100 * 0.1 = 10$. To compare the load balancing approaches for different data skews, we are interested at the execution time of the approaches when the data source presents regions with high similarity detection.

The execution time of the approaches for different data skews of DS1 (n = 20, m = 40,

r = 40, w = 1000) is shown in Figure 4.5. As we can see, the execution time of *MultiRepSN* is similar in all scenarios. On the other hand, since *MultiRepSN* employs a fixed window size, the workload balance is uniform. Also note that *MultiMR-*$DCS++$ outperforms *MultiRepSN* in the scenarios which there is duplicate detection. Only in the case that there is no duplicates in the data source, the execution time of the approaches was similar. This result confirms that even with the increasing of the window size in regions of high duplicate detection (which leads to more entity comparisons), the usage of the transitive closure mechanism works by minimizing the execution time. According to our experiment, the reason why the execution time is decreased as the degree of skewness grows is that the time saved due to the reduction of unnecessary comparisons (promoted by the transitive closure mechanism) is higher than the time spent by the additional entity comparisons (generated due to the increasing of the window size). Therefore, the results we have carried out indicate that there is no scenario in which MultiRepSN outperforms *MultiMR-*$DCS++$ in terms of execution time due to unbalanced workloads.

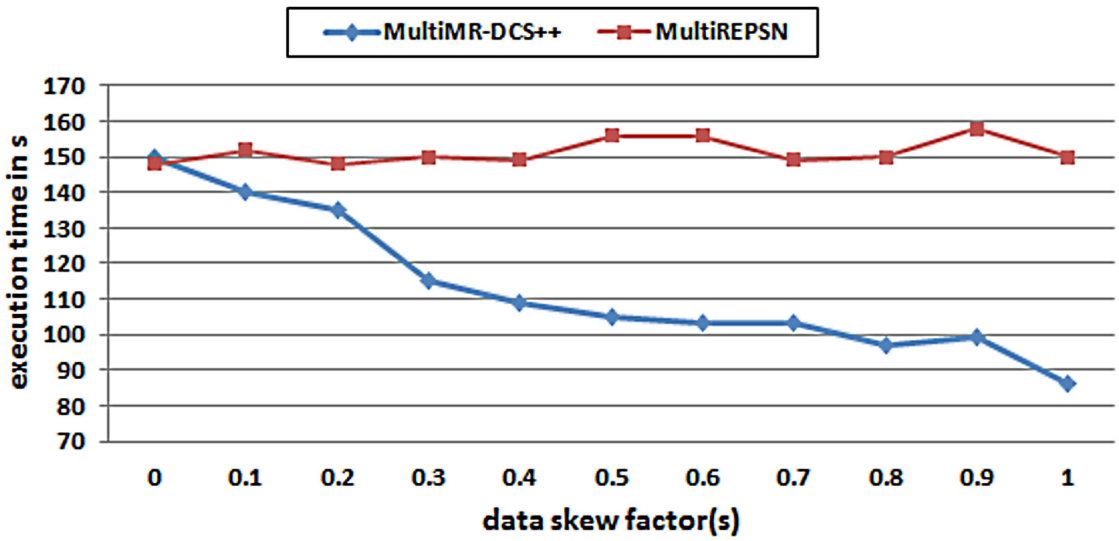

Figure 4.5: Execution times for different data skews using $w = 1000$ (n=20, m=40, r=40).

4.2.2 Scalability: Number of Nodes Available

Scalability is important for many reasons and one of them is the financial issue. For instance, the number of nodes should be carefully estimated since distributed infrastructure suppliers

usually charge per hired machines even if they are underutilized [50]. Some suppliers also charge per processing cycles, which highlights the need to avoid unnecessary entities comparisons. To evaluate the approaches' scalability by varying the number of available nodes, we study their behavior for the single-pass and multi-pass executions.

Single-pass Execution

To analyze the scalability of the two multi-pass approaches executing just one pass, we vary the number of nodes from 1 up to 20. Following the Hadoop's documentation, for *n* nodes, the number of map tasks is set to $m = 2 \cdot n$ and the number of reduce tasks is set to $r = m$. The values of the execution times are shown in Figure 4.6 (DS1) and Figure 4.7 (DS2). Since the number of comparisons also grows with the window size (increasing the execution time), we defined the same window size for both approaches according to the magnitude of the data source size to avoid benefiting a specific approach. For DS1 and DS2, we utilized w = 100 and w = 1000, respectively, aiming at verifying the scalability of the two approaches when performing a small and large ($w \geq 400$ [50]) window size.

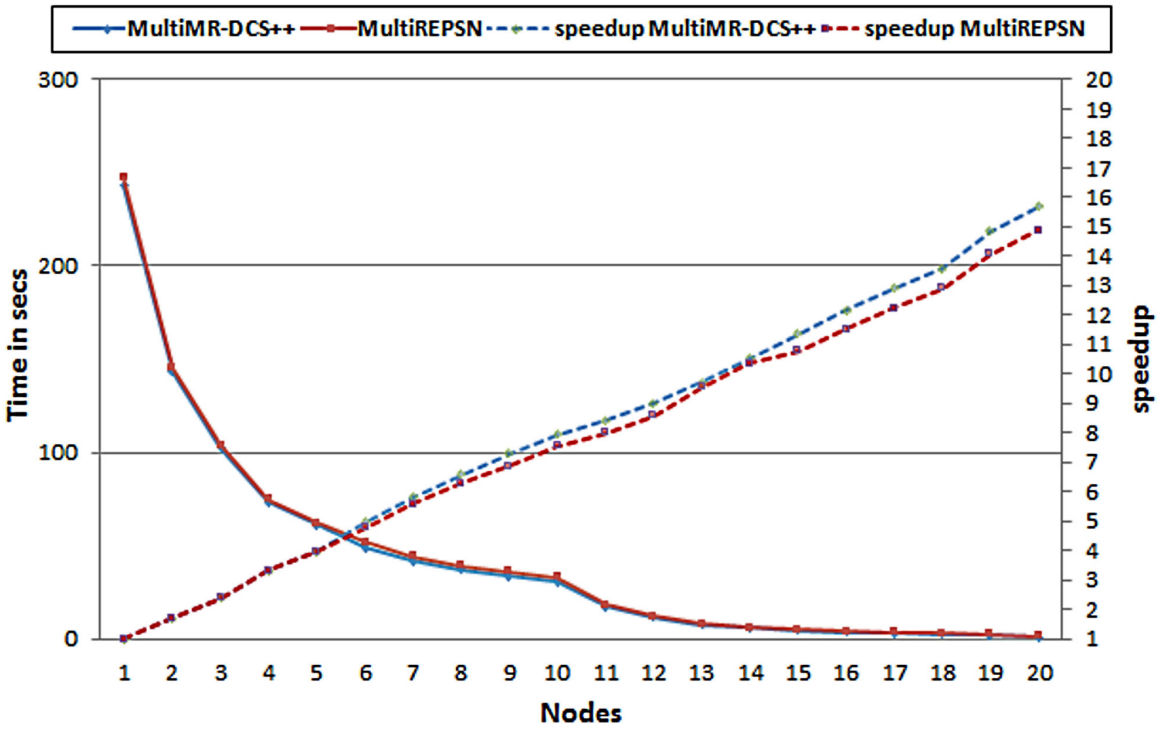

Figure 4.6: Execution times and speedup for both approaches using DS1 (w = 100).

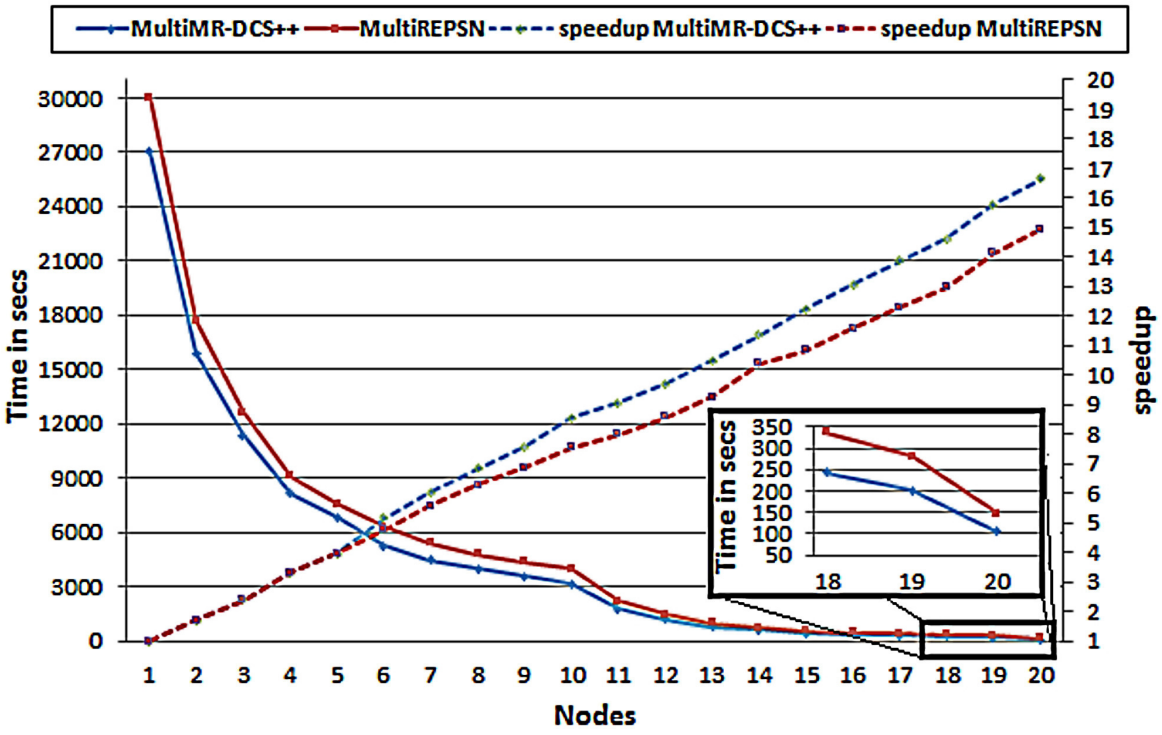

Figure 4.7: Execution times and speedup for both approaches using DS2 (w = 1000).

Both *MultiMR-*$DCS++$ and *MultiRepSN* scale almost equally for the smaller and larger data sources DS1 and DS2, respectively. These results show their ability to evenly distribute the workload across reduce tasks and nodes. However, due to the difficulties presented by the traditional SNM related to the fixed (and difficult to configure) window size, its performance is depreciated by the execution of many unnecessary comparisons. This problem is the main reason why the *MultiRepSN* approach always performs slower than *MultiMR-DCS++* even presenting by design an uniform load balancing mechanism. The difference is highlighted by the speedup.

In the evaluation with DS1, *MultiMR-DCS++* saved $102,038$ comparisons due to the adaptive window, which resulted in a decrease of execution time. Also, note that with the increase in number of nodes, the execution time difference between the approaches decreases due to a better division of the unnecessary comparisons among the nodes performed by *MultiRepSN* approach.

However, in the evaluation with DS2 (Figure 4.7), the performance difference between the two approaches is almost two nodes in favor of *MultiMR-DCS++*, according to the speedup with n = 20. This means that a huge data source and a large window size can

depreciate fairly the execution of *MultiRepSN*.

Multi-pass Execution

For this experiment, we firstly compare the *MultiMR-DCS++* approach against the naive strategy, *i.e.*, the repeated execution of single-pass *MR-DCS++* [62], aiming to verify if there is any difference in the execution time of the strategies. Figure 4.8 shows the result of 1 to 3 passes employing w = 1000. We utilize the following three blocking functions in each pass: first letter, first three letters, and first five letters of the publication title. In practice, the usage of these blocking functions is not common, however since the purpose of this evaluation is to measure execution time, they are valid because they change the sorting order of the entities.

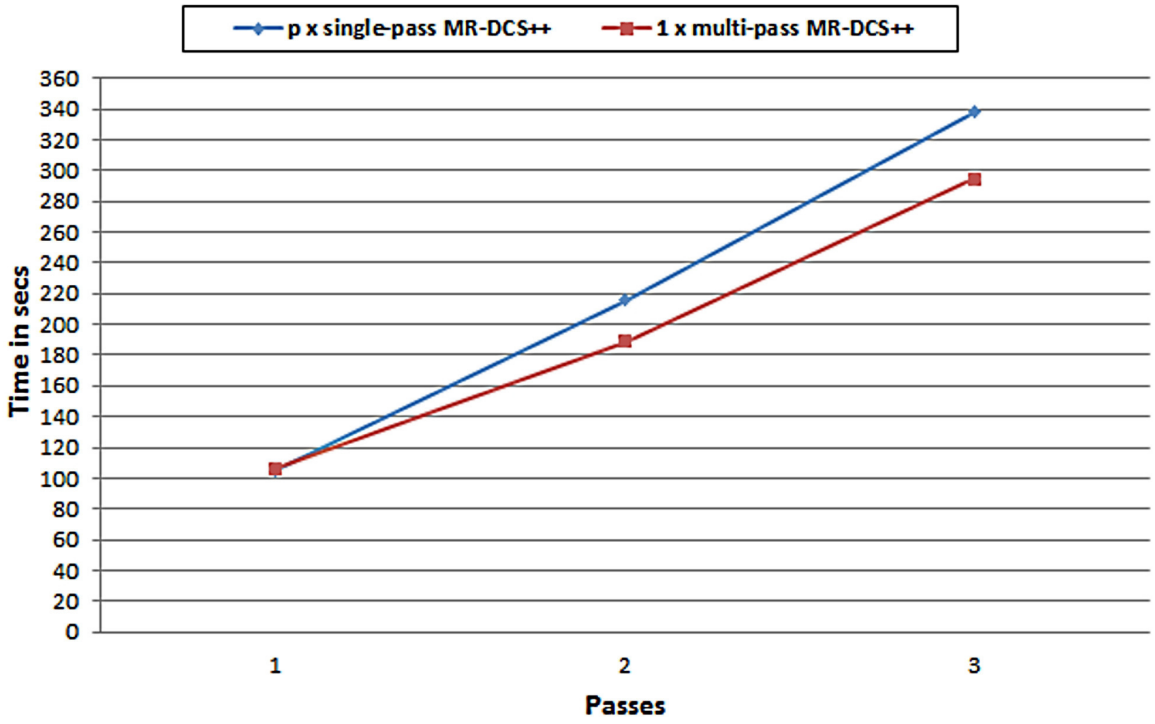

Figure 4.8: Comparison of the approaches for multi-pass SN using MR-DCS++ (w = 1000) for DS2 with n = 20 nodes. p x single-pass *MR-DCS++* [62] performs single-pass *MR-DCS++* p times whereas 1 x multi-pass *MR-DCS++* performs multi-pass *MR-DCS++* one time.

Note that the execution time grows linearly with the number of passes because the number of processed pairs is linear to the number of entities and passes. The p x single-pass

MR-DCS++ [62] strategy mainly suffers from the fact that the input data needs to be read and parsed p times whereas the additional overhead of multiple *MR-DCS++* executions is relatively low. *MultiMR-DCS++*, on the other hand, avoids the p-fold input reading and clearly outperforms the p x single-pass *MR-DCS++* strategy. The execution time improvements increase with more passes, *e.g.*, from about 14% for two passes to 17% for three passes.

We also compare the *MultiMR-DCS++* approach against the *MultiRepSN* one varying the number of available nodes and passes aiming to verify their behavior with respect to the execution time as the number of nodes and passes increase. In this experiment, we utilize the larger data source DS2, employ w = 1000 and apply the same three (passes) blocking functions mentioned at the beginning of the section.

As we can note, the *MultiMR-DCS++* approach outperforms *MultiRepSN* approach in the three scenarios (*i.e.*, one, two and three passes). This result is due to the difficulties presented by the traditional SNM related to the fixed (and difficult to configure) window size, its performance is depreciated by the execution of unnecessary comparisons. As the number of passes increases, the number of unnecessary comparisons is considerably increased. Note that, with n = 20, the execution time difference between the two approaches, in the scenario with three passes (around 77 secs), is much greater than the execution time difference in the scenario with one pass (around 42 secs). This problem is the main reason why the *MultiRepSN* approach always performs slower than *MultiMR-DCS++* even presenting by design an uniform load balancing mechanism. Also, note that, even with the number of unnecessary comparisons being increased due to the increasing number of passes, there is a kind of compensation being held in the execution time difference between the approaches due to a better division of the unnecessary comparisons among the nodes as the number of nodes is increased.

4.2.3 Matching Quality vs. Execution Time

Finally, we study the trade-off between the match quality, in terms of F-Measure (*i.e.*, the harmonic mean of precision and recall [14]) and the execution time. The data sources used so far could not be applied for this evaluation due to the absence of a gold standard necessary for the match quality calculations. As aforementioned, we utilized another publication data

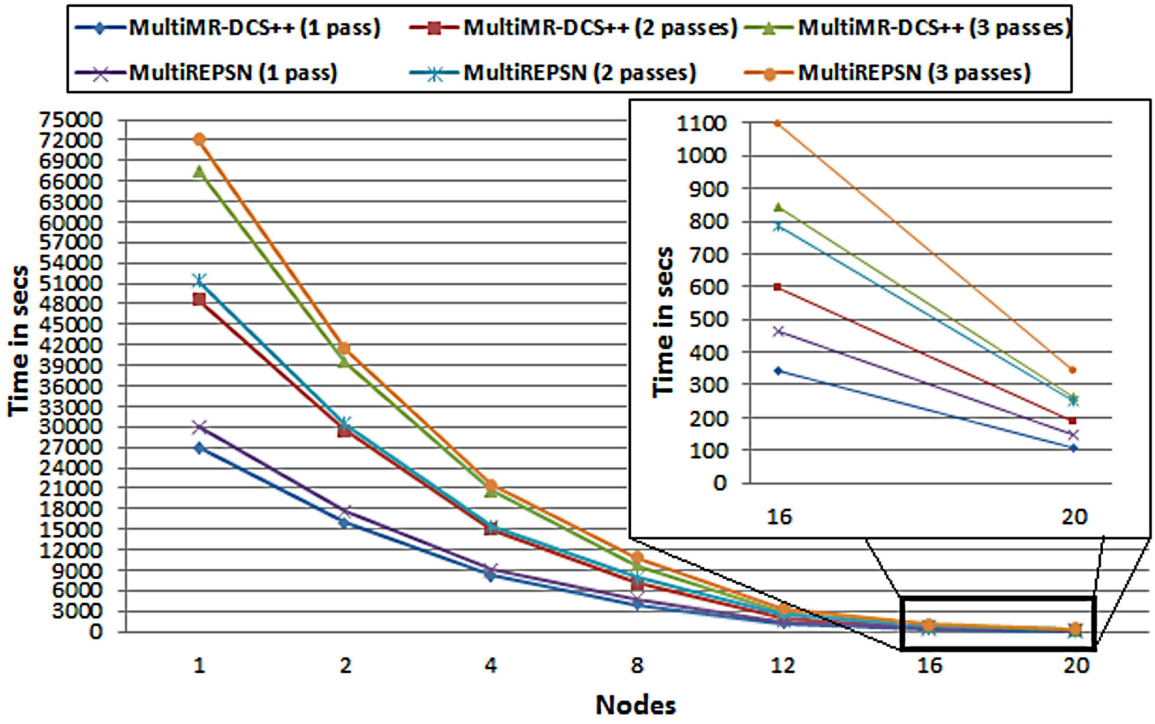

Figure 4.9: Execution times for both multi-pass approaches using DS2 (w = 1000) varying the number of nodes and passes.

source (based on [53]) that contains about 7,800 records and along with the data source there is the gold standard needed to compute the precision, recall and F-Measure metrics. Two window sizes $w = 500$ and $w = 1000$ for the single-pass *MultiRepSN* and *MultiMR-DCS++* approaches are employed and compared with other three multi-pass *MultiRepSN* and *MultiMR-DCS++* strategies with two passes (passes=2) and window sizes $w_1 = w_2 = 100, w_1 = w_2 = 200$ and $w_1 = w_2 = 500$, respectively. The blocking keys used this time are the first word of the publication title (pass #1) and the first author's name (pass #2), respectively. As mentioned earlier, two entities are considered a match if their titles have a q-gram similarity $\Phi_{max} \geq 0.75$. In this case, we utilize α = 0.5 since the comparisons with a similarity distance below 0.25 ($\Phi_{min} = 0.75 - 0.5$) have proven to be unpromising. This time, we run our experiments on four nodes ($m = r = 8$) due to the small size of the data source. Otherwise, if a large number of nodes were utilized, due to the small data source, the executions would be instantaneous and it would be difficult to highlight the differences between the execution times.

The left part of Figure 4.10 shows the observed execution times along with the number

of comparisons and the precision, recall and F-measure collected values for the execution of the two approaches with window sizes $w = 500$ and $w = 1000$ (one pass each). The right part shows the results of the execution of the two approaches for two passes with window sizes $w_1 = w_2$ of 100, 200 and 500. Precision, recall and F-Measure are computed including all pairs that have been directly identified and those generated by transitive closure.

Single-pass					Multi-pass (passes=2)					
	MultiRepSN		MultiMR-DCS++		MultiRepSN			MultiMR-DCS++		
Window size	w = 500	w = 1000	w = 500	w = 1,000	w_1=w_2=100	w_1=w_2=200	w_1=w_2=500	w_1=w_2=100	w_1=w_2=200	w_1=w_2=500
#Comparisons	≈3.68•10^6	≈7.12•10^6	≈2.45•10^6	≈4.43•10^6	≈1.51•10^6	≈2.99•10^6	≈7.36•10^6	≈1.07•10^6	≈2.11•10^6	≈4.98•10^6
Execution time (in secs)	93	124	72	98	67	82	138	60	69	101
Matching Quality					Matching Quality					
Precision	92.33%	91.57%	88.97%	88.22%	93.26%	93.10%	92.21%	87.63%	90.55%	85.37%
Recall	63.91%	64.63%	70.39%	71.03%	66.86%	68.09%	70.66%	75.70%	78.25%	80.30%
F-Measure	75.53%	75.77%	78.59%	78.69%	77.88%	78.65%	80.01%	81.23%	83.96%	82.76%

Figure 4.10: Comparison of quality and execution time for multi-pass RepSN and MR-DCS++ with one and two passes using different window sizes.

In all cases, both for single and multi-pass, Figure 4.10 shows that the number of comparisons performed by *MultiRepSN* is significantly higher than the amount performed by *Multi-MR-DCS++* (under the same window size condition). For instance, single-pass *MultiRepSN* for $w = 1000$ performs about 2.7 million more comparisons than *MultiMR-DCS++*. In this case, the difference is also highlighted by the execution time since *MultiMR-DCS++* (98 secs) outperforms *MultiRepSN* (124 secs) by 21%. This indicates that *MultiMR-DCS++* performs efficiently the task distribution among the nodes since *MultiRepSN* by design has a robust load balancing mechanism (due to the fixed window size) since the execution time difference between the two approaches is still significant given the gain with the saved comparisons. In terms of matching quality, although the slight decrease in the precision, due to the false negatives generated by the wrong transitive closure assumptions, *MultiMR-DCS++* outperforms *MultiRepSN* in all cases with respect to recall and F-Measure due to its ability to increase the window in regions of high similarity and perform better guesses about which comparisons not to perform (by skipping them). Furthermore, the recall of *MultiRepSN* approach is compromised since it misses some real duplicates due to the usage of a fixed window size.

Note that both approaches achieve a significant better F-Measure when performed with two passes due to the improved recall. For instance, the *MultiMR-DCS++* F-Measure value

using one pass with $w = 1000$ is 78.69% whereas using two passes with $w = 100$ (38 secs faster) is 81.23%, about 3% higher. Hence, in this case the consideration of multiple blocking keys helps to find many more matches despite the smaller window sizes.

The shown results confirm that *MultiMR-DCS++* is able to preserve with small losses the quality levels of EM results even with smaller window sizes and that it can outperform single-pass *MR-DCS++* not only in preserving the quality levels of EM results but also in efficiency due to reduced window sizes.

4.3 Final Considerations

In this chapter, we proposed a novel Multi MR-based approach for solving the problem of the adaptive SNM parallelization, multi-pass *MR-DCS++*. The solution provides an efficient parallelization of the $DCS++$ method [28] by using multiple MR jobs and applying a tailored data replication during data redistribution to allow the resizing of the adaptive window. The approach also addresses the data skewness problem with an automatic mechanism of data partitioning that can be combined with *MR-DCS++* to ensure a satisfactory load balancing across all available nodes. Our evaluation on a real cluster environment using real-world data demonstrated that *MultiMR-DCS++* scale with the number of available nodes. We compared our approach against an existing one (*RepSN*) for single and multi-pass and verified that *MultiMR-DCS++* in both cases overcomes *RepSN* in performance (execution time and matching quality) terms. In the following chapter, we will investigate if a combination of two competing methods, *i.e.*, blocking and the adaptive windowing, as well as its Spark-based model, can improve the efficiency of the Entity Matching task.

Chapter 5

Enhancing Entity Matching Efficiency through Adaptive Blocking and Spark-based Parallelization

As mentioned in Chapter 1, detecting similar or matching entities (records) is difficult due to the need of applying specific matching techniques on the Cartesian product of all input entities which leads to a computational cost in the order of $O(n^2)$. This means that the application of such approach is ineffective, specially in scenarios with predefined time restrictions [66]. The main way to minimize the workload caused by the Cartesian product execution and still maintain the matching quality is to reduce the search space by applying indexing (blocking) techniques [14], [61], [94], [63] (broadly discussed in Chapter 2).

A complete survey shown in [15] provides a detailed discussion of six indexing methods (with a total of 12 variations of them). The work also describes a theoretical analysis of the methods' complexity and an empirical evaluation of them over a variety of both real and synthetic data sources. The empirical investigation showed that Standard (traditional) Blocking [5] and Adaptive Windowing [100] achieved the highest duplicate detection rates (F-Measure) according to all the parameter settings. The study provides evidence that a generalization involving these two methods can possibly enable the reduction of the overall number of comparisons and still maintain the high values of duplicate detection rate.

Moreover, it is known that open source parallel programming models for data-intensive tasks emerged as a new paradigm to run jobs which previously used to be complex and

costly to run, such as Entity Matching over Big Data sources [14]. These new programming models, such as Spark [104], can run on commodity hardware and provide high scalability, fault tolerance and flexibility to distributed systems. Thus, given the pair-wise comparison nature of the problem, EM is a data-intensive and performance critical task that also demands studies on how it can benefit from the Spark framework and its efficient resources.

Therefore, given that a generalized method involving the Standard Blocking and the Adaptive Windowing as well as its efficient Spark-based parallelization has not yet been proposed, to the best of our knowledge, we make the following contributions in this chapter:

- We propose a new generalized indexing method, denoted as **B**locked **A**daptive **W**indowing (BAW), which combines the best features of the Standard blocking and Adaptive Windowing methods;

- We propose two variants of BAW, denoted as **O**verlapped **B**locked **A**daptive **W**indowing (OBAW), which enables blocking overlap, and **R**etrenched OBAW (ROBAW), which combines both the blocking overlap and a retrenchment strategy to also adapt the window size in regions of entities with low duplicate detection rates;

- We propose the **S**park-based **B**locked **A**daptive **W**indowing (S-BAW), a Spark-based approach that provides an efficient parallelization of the blocking method proposed in this chapter. The proposed approach also addresses the data skew problem with an automatic data partitioning strategy that provides a satisfactory load balancing across all available nodes;

- We evaluate the new generalized algorithm and its variants against the main state-of-the-art indexing methods (*i.e.*, Standard Blocking, fixed Sorted Neighborhood, Adaptive Windowing and the generalization of the Standard Blocking with the fixed Sorted Neighborhood) and show that the new generalized method provides better performance by diminishing the overall number of comparison pairs and still maintaining high values of duplicate detection quality. Furthermore, we evaluate *S-BAW* and its variants against the main state-of-the-art parallel approaches and show that the formers achieve better performance by diminishing the overall EM execution time. The evaluation employs a real cluster environment and uses real-world data sources.

5.1 Blocking the Adaptive Windowing

In this section, we present a new generalized method, denoted as **B**locked **A**daptive **W**indowing (BAW), which combines the search space reduction promoted by the Standard Blocking method with the adaptability efficiency of the Adaptive Windowing method (DCS++). In addition, we also present two variants of BAW that improve both the efficiency and efficacy of this method.

5.1.1 Blocked Adaptive Windowing

The **B**locked **A**daptive **W**indowing (BAW) method is based on the combination of Standard Blocking (SB) [5] and Adaptive Windowing (DCS++) [28] aiming to mix the assumptions that blocking and sorting the entities, based on a blocking/sorting key, approximates entities that have a higher probability of being duplicates. Basically, BAW follows the idea of:

- sorting the entities based on a sorting key;
- blocking the entities according to a blocking/sorting key; and
- performing the DCS++ method within each block.

The motivation behind this strategy is to decrease even more the number of non-matching comparisons by bounding the areas where the adaptive window can act in such a manner to avoid unnecessary comparisons (from different blocks). For instance, it does not make sense to compare publication titles from different years. Thus, the Adaptive Windowing method (DCS++) slides until the end of each block is reached. When the end of a block is reached, the initial size of the window in the subsequent iterations is decreased by 1 and the window size is no longer increased. Algorithm 4 shows the pseudo-code of the BAW strategy. Note that, as shown in lines 3 to 5, DCS++ is performed considering each block as an independent data source.

To illustrate the BAW method, Figure 5.1 shows the same execution example presented in Section 2.1.1. For each block, the Adaptive Windowing (DCS++) utilized a window size $w_{initial}$ = 3. Note that, initially, the window includes the first two entities (A, D) and generates one pair of comparison $[(A - D)]$. After that, the window slides to the next entity

Algorithm 4 Blocked Adaptive Windowing (BAW)

```
1: function BAW(entities, blockingKey key, initial window size w, threshold φ_ddr)
2:   sorts entities by key;
3:   blockedEntities[] ← extractBlockedEntitiesByBlockingKey();
4:   for b = 1 → blockedEntities.size(), b++ do
5:     DSC++(blockedEntities[b], w, φ_ddr); //performs DCS++
```

(B) belonging to block Per. From B, the next comparison $B - E$ is considered a match. Thus, the following relation of DCS++ is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of earlier detected duplicates within the window and c is the number of comparisons already done also within the window. Thus, w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (E), as long as the block boundaries are not surpassed. Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of E. Now, w covers B, E, F and H; the new comparisons generated are B-F and B-H. Since B-F is regarded as a non-match pair, the w increasing test is not loaded. Then, the comparison B-H is performed. Since H is the last entity of block Per, the w increasing test is no longer loaded (to avoid the comparisons of entities belonging to different blocks).

Note that the pair E-H is regarded as a match due to the transitive closure assumption. The windows starting with entities E and F were also skipped for the same reason. After that, w is set to the initial value (three) and the window slides to the next block (Tea), starting from entity C. The example also shows that the number of comparisons to detect the four duplicates decreases by five (comparisons) with respect to the DCS++ method (see Subsection 2.1.1). At the end, the number of comparisons performed is seven.

5.1.2 Blocked Adaptive Windowing Variants

As we can notice, BAW is recommended when an ideal/effective blocking key is provided due to the delimitation of proper regions of entities. By doing that, the adaptive window can act in such a manner to avoid unnecessary entity comparisons (from different blocks). However, when dealing with dirty (*i.e.*, inaccurate, incomplete or erroneous) input data, it may not be possible to use an effective blocking key. Even using an adaptive window size, if the blocking is not ideal, it is quite common that the similarity value between entities belonging to different blocks remains considerable. For this reason, we propose the first BAW variant, denoted as **O**verlapped **B**locked **A**daptive **W**indowing (OBAW), to enable

blocking overlap.

Overlapped Blocked Adaptive Windowing - OBAW

The idea of overlapping the Blocked Adaptive Windowing is to allow the adaptive Windowing to surpass the end of a block and cover entities from adjacent blocks. The strategy works as follows: the adaptive sliding window is performed within the block, but, instead of stopping to increase the window when the end of the block is reached (such as BAW), OBAW enables the adaptive window to increase beyond the limits of the current block. In addition, it decreases the adaptive window size when the end of the block is near and only overlaps the blocks if the adaptive window size needs to increase. Algorithm 5 shows the pseudo-code of the OBAW strategy. Note that, as shown in lines 7 and 8, DCS++ is performed in each block and if the window has to increase at the end of the block; entities from the next block are placed together as far as the window increases.

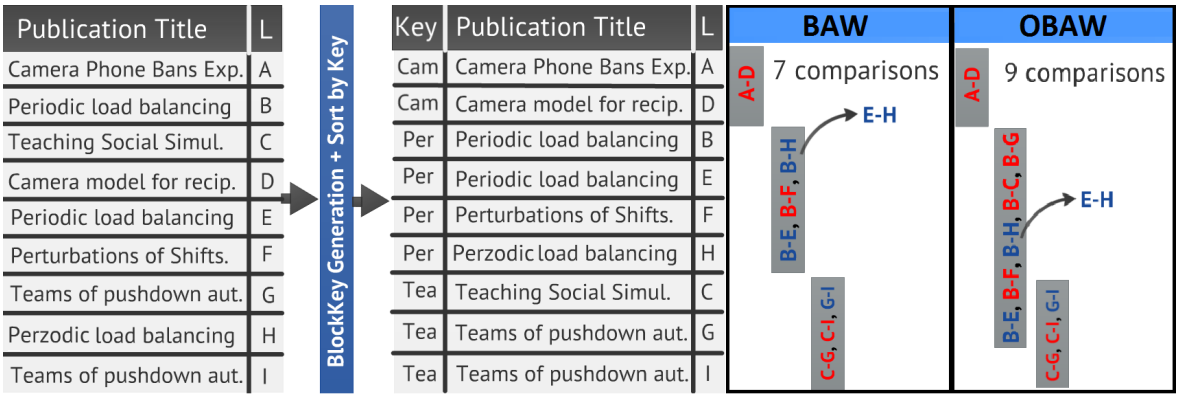

Figure 5.1: Execution example of the BAW and OBAW methods with adaptive initial window size $w_{initial} = 3$.

In the running example of the OBAW method shown in Figure 5.1, for each block, the Adaptive Windowing (DCS++) utilized a window size $w_{initial}$ = 3. Initially, the window includes the first two entities (A, D) and generates one pair of comparison $[(A - D)]$. After that, the window slides to the next entity (B) belonging to block Per. From B, the next comparison $B - E$ is regarded as a match. Thus, the following relation according to DCS++ is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of previously detected duplicates within the window and c is the number of comparisons already done also within the window, then w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (E).

Algorithm 5 Overlapped Blocked Adaptive Windowing (OBAW)

```
1: function OBAW(entities, blockingKey key, initial window size w, threshold φ)
2:   w_initial = w;
3:   sorts entities by key;
4:   blockedEntities[] ← extractBlockedEntitiesByBlockingKey();
5:   for b = 1 → blockedEntities.size(), b++ do
6:     DSC++(blockedEntities[b], w, φ_ddr); //performs DCS++
7:     //The following lines are placed at the end of DSC++
8:     //getRemainingEntities returns the remaining entities of block b that were not yet considered in the adaptive window sliding
9:     if w > blockedEntities[b].getRemainingEntities() and w > w_initial then
10:      blockedEntities[b].addEntities(getEntities(blockedEntities[b + 1], w − blockedEntities[b].getRemaining
       Entities()));
```

Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of E. Now, w covers B, E, F, and H; the new comparisons generated are B-F and B-H. Since B-F is regarded as a non-match pair, the w increasing test is not loaded. Then, the comparison B-H is performed. Since B-H is regarded as a match, the w increasing test is loaded once more resulting in true ($\frac{2}{3} \geq 0.5$) and the window is increased by two again, ovarlapping the next block (Tea) by 2 entities. This time, the window is increased from H, the last duplicate found. Now, w covers B, E, F, H, C and G; B is compared with the rest of the entities within the window. The window is no longer increased due to the lack of new matches. Note that, the pair E-H is regarded as a match due to the transitive closure assumption. After that, w is set to the initial value (three) and the window slides to the next block (Tea), starting from entity C. The example shows that the number of comparisons (nine) required by OBAW to detect the four duplicates decreases by three comparisons with respect to the DCS++ method even with OBAW enabling block overlapping (see Subsection 2.1.1).

Retrenched OBAW - ROBAW

Retrenched Blocked Adaptive Windowing attacks a vulnerability of the DCS++ method related to the window inadaptability to decrease in regions of entities with almost no duplicate detection and, consequently, reduce the number of unnecessary comparisons. In order words, the execution of DCS++, in regions of entities with rare presence of duplicates, performs almost similar to the SNM with fixed window size (*i.e.*, DCS++ worst case) [28]. To overcome this vulnerability, we propose a retrenchment adjust in the adaptability of the window in regions of rare presence of duplicates. This window retrenchment is only applied when

a non-matching pair is detected and the constraint $d = 0$ is satisfied, where d is the number of detected duplicates within the current window. When a new duplicate is found and $\frac{d}{c} >= \phi_{ddr}$, the window size is set to its original size and the increasing strategy of DCS++ is applied normally.

The retrenchment strategy works as follows: every time a non-matching pair is detected and $d = 0$, the similarity value (sv) (*e.g.*, $sv = 0.5$) is utilized to set a global variable, called retrenchment factor (rf), with a new value according to $rf = \frac{rf+sv}{2}$. This retrenchment factor is then used to calculate the percentage of the window size that must be retrenched. The percentage of retrenchment ($WR_{percentage}$) is calculated as: $WR_{percentage} = \frac{100*rf}{\phi_{matching}}$. For instance, assume that the similarity threshold (to decide whether a comparison pair is a match or not) is $\phi_{matching}$ = 0.7, the actual retrenchment factor is $rf = 0.3$ and the new pair of entities is regarded as a non-match with a similarity value sv = 0.4. This means that $rf = \frac{rf+sv}{2} = \frac{0.7}{2} = 0.35$ and $WR_{percentage} = \frac{100*rf}{\phi_{matching}} = \frac{35}{0.7} = 50$. Thus, the window size must be retrenched in 50% of the number of remaining entities (which have not been compared yet) within the current window. The intuition behind this strategy is that, if a similarity value of a non-matching pair within the current window is becoming distant from $\phi_{matching}$, the current window size must be decreased proportionally.

The ROBAW variant was designed to overlap blocks and retrench the adaptive window size in regions of entities that do not present duplicate detection. Algorithm 6 shows the pseudo-code of the ROBAW strategy. Note that, as shown in lines 7 to 11, DCS++ is performed in each block and, if the window needs to increase at the end of a block, the entities from the next block are placed together as far as the window increases according to the DCS++ strategy. Lines 14 to 26 show that the retrenchment algorithm must be added to the DCS++ procedure responsible for adapting the window size to enable the retrenchment feature.

In Figure 5.2, some changes in the original running example (Figure 5.1) were made to enable a proper illustration of the ROBAW execution. Firstly, the block Cam was removed and the block Per received two more entities (A and D). Also, some of the existing entities (E, F and H) were changed aiming a better illustration. Finally, the Adaptive Windowing (DCS++) utilized a window size $w_{initial}$ = 5.

In Figure 5.2, we can see that the window includes the first five entities (A, D, B, E, F)

Algorithm 6 Retrenched Blocked Adaptive Windowing (ROBAW)

function ROBAW(*entities*, $blockingKey$ *key*, initial window size *w*, threshold ϕ_{ddr})
  $w_{initial} = w$;
  $sorts\ entities\ by\ key$;
  $wSizeBeforeRetrenchment = w$;
  $rf = 0$;
  $blockedEntities[] \leftarrow extractBlockedEntitiesByBlockingKey()$;
  for $b = 1 \rightarrow blockedEntities.size(), b{+}{+}$ **do**
    $DSC{+}{+}(blockedEntities[b], w, \phi_{ddr})$; $//performs\ DCS{+}{+}$

    //The following lines are placed in ROBAW to support overlapping blocks;
    if $w > blockedEntities[b].getRemainingEntities()$ **and** $w > w_{initial}$ **then**
      $blockedEntities[b].addEntities(getEntities(blockedEntities[b+1], w - blockedEntities[b].getRemainingEntities()))$;

    //The following lines are placed inside ROBAW (more specifically inside the method responsible for adapting the window if necessary) to enable the window size retrenchment;
    if $d == 0$ **and** $isNonDuplicate(currentPair)$ **then**
      $sv = getSimilarity(currentPair)$;
      $rf = (rf + sv)/2$;
      $WR_{percentage} = (100 * rf)/\phi_{matching}$;
      if $alreadyRetrenched() == false$ **then**
        $wSizeBeforeRetrenchment = w$;
        $retrenchBy(w.getRemainingEntities(), WR_{percentage})$;
        $lastWR = WR_{percentage}$;
      else if $WR_{percentage} > lastWR$ **then**
        $w.size = wSizeBeforeRetrenchment$;
        $retrenchBy(currentWindowRemainingEntities(), WR_{percentage})$;
    else
      $w.size = wSizeBeforeRetrenchment$;

and generates four pairs of comparisons $[(A-D),(A-B),(A-E),(A-F)]$. Since the first two comparisons were regarded as non-match ($d=0$), the process of retrenchment is started. Firstly, rf is updated: $rf = \frac{0+0.4}{2} = 0.2$. Afterwards, $WR_{percentage}$ is also updated: $WR_{percentage} = \frac{100*rf}{\phi_{matching}} = (100*0.2)/0.7 = 28$ (assume 0.7 is the threshold that determines whether a comparison pair matches or not). In other words, 28% of the remaining entities must be removed from the current window; in this case, the entity F. Thus, the comparison pair A-F is not evaluated. Since one retrenchment was performed, another retrenchment will be performed only if the next $WR_{percentage}$ is greater than 28. After that, the rest of the comparisons are performed and, as the end of the block is approaching, the window size is retrenched by one entity until it reaches the end of the block (as explained in Subsection 5.1.2). However, one duplicate was found (F-H) and since it is the first comparison, the window increasing evaluation requires the window size to be increased from that duplicate (H). Thus, the window is allowed to overlap the blocks and proceed the regular adaptive increasing. With this strategy, the ROBAW method can perform comparisons of entities from different blocks that have a high probability of being duplicates.

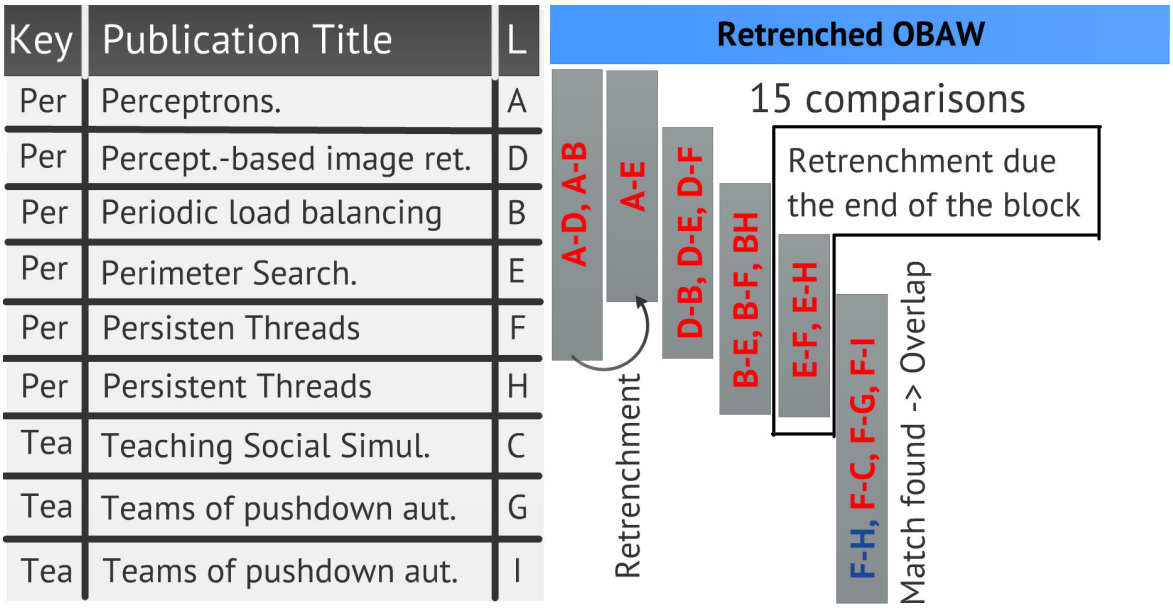

Figure 5.2: Execution example of the Retrenched BAW method with adaptive initial window size $w_{initial} = 5$.

5.2 Spark-based Adaptive Windowing

An interesting line of reasoning when we deal with Spark-based EM is to define an efficient approach (with load balancing handling) by knowing previously the entity comparisons generated by the serialized blocking (windowing) method. However, how do we define an efficient Spark-based approach when the blocking (windowing) method adapts according to the duplicate detection rate (such as BAW and its variants strategies)? How do we assign entity comparisons to the proper workers with load balancing handling without previously knowing all the necessary entity comparisons? To answer these research questions, we propose a Spark-based *BAW* for EM processing using multiple transformation steps. Each step is detailed with examples in the following subsections.

5.2.1 Spark-based BAW (*S-BAW*)

To provide an efficient workload distribution among the workers and also enable the BAW method to be efficiently executed in a Spark workflow, we perform the adaptive windowing EM processing employing three transformation steps as illustrated in Figure 5.3. The key idea is to improve the load balancing by fixing the same number of entities at each worker input partition. By doing this, the number of entities processed by each worker will be the same, leading the worker to perform a maximum number of window's slides (due to the fixed partition size). This mechanism improves load balancing because the definition of a maximum number of window's slides avoids the overload of a single worker.

Thus, the first step of the approach is a Spark mapper that receives as input from each worker a partition of the data source. As mentioned in Section 5.1.1, BAW assumes an ordered list of entities based on their blocking keys. Since the input data source consists in an unsorted entity collection, the mapper step determines the blocking key of each entity and outputs a key-value RDD pair containing an output key = (blocking key) and value = (entity) to enable the combination and sorting of entities by their key. During this mapper step, the key-value RDD pairs are partitioned based on the average (a) number of entities per available reducers (in the second step), $a = \frac{\Sigma|\Pi|}{r}$, such that Π represents the set of partitions of the data source and r corresponds to the number of reducers (workers). The aim of this step is to avoid the data skew problem by allocating the same number of entities to each worker.

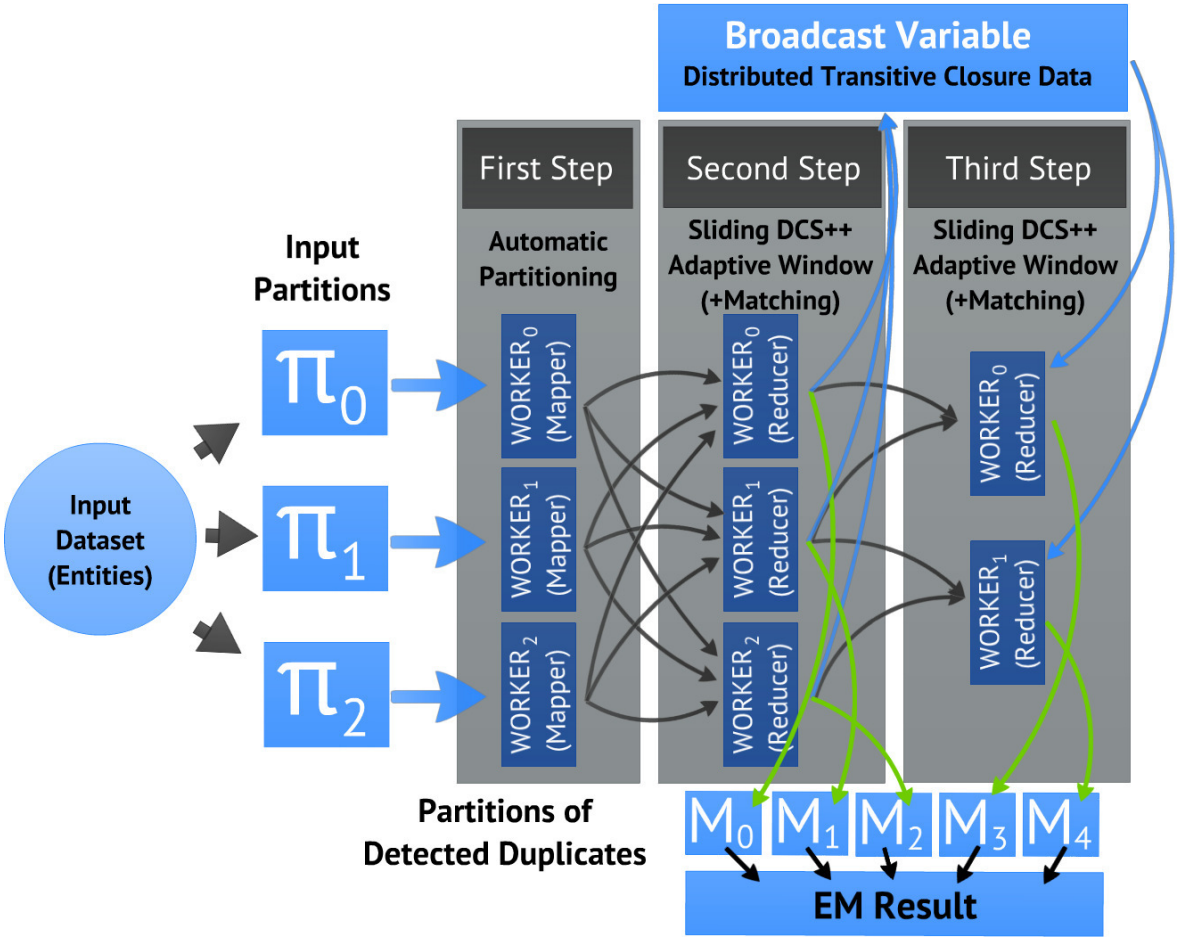

Figure 5.3: Overview of the *S-BAW* matching process workflow.

In the second step, the key-value RDD pairs of each reducer input partition are sorted by the key (according to *BAW*) and the reducer performs the adaptive sliding window over the sorted partition. The algorithm emits a partial result of the EM processing, saves the transitive closure information in a broadcast variable (that will be accessed by workers during the third step) and stores as a mapper output the entities of the partition that could not be processed during the sliding window due to the overlapping assumptions. By overlapping assumptions, we mean the cases in which the window may need to slide or to be increased in order to cover entities belonging to another reducer input partition when the sliding window achieves the end of the current partition. In this case, the EM processing stops and the entities belonging to the last window are stored in the memory as a mapper output to be recovered by the third step. From the subsequent reducer input partition, only the $w_{initial}$ entities are stored as a mapper output. In other words, one of the reducers of the third step will process the rest of the incomplete window processing attached to the $w_{initial}$ entities from the subsequent reducer input partition. This strategy ensures that, if a window needs to increase even more due to a new match found, the transitive closure information generated by the sliding window of the subsequent reducer input partition will cover the new window increased due to this new match.

In the running example shown in Figure 5.4, the mapper reads the entities and generates (for each entity) a key-value pair with a map output key = (blocking key) and value = (entity) during the first step. For instance, the map output key of E is cam since E's blocking key is cam. The first key (of E) is assigned to the second reducer because the amount of entities assigned to the first reducer reached the average number of entities per reducer ($a = \frac{4+4+4}{3} = 4$). Once all entities are positioned, the reduce phase starts sliding the adaptive window. In the example, we use Φ_{max} = 0.9 (the similarity threshold that indicates whether a pair of entities matches, or not) and w = 3, which implies that, according to the *DCS++* strategy, the increasing condition threshold due to the duplicate detection rate is $\Phi_{ddr} = \frac{1}{w-1} = \frac{1}{3-1} = 0.5$.

Thus, in the reducer 0, the first generated window covers the entities A, B and C. This results in the following comparisons: A-B and A-C. Both pairs are regarded as non-matches and, since no duplicate entities are identified, there is no need to increase the window size. Therefore, the window slides to the next entity (B). From B, the next comparison B-C is

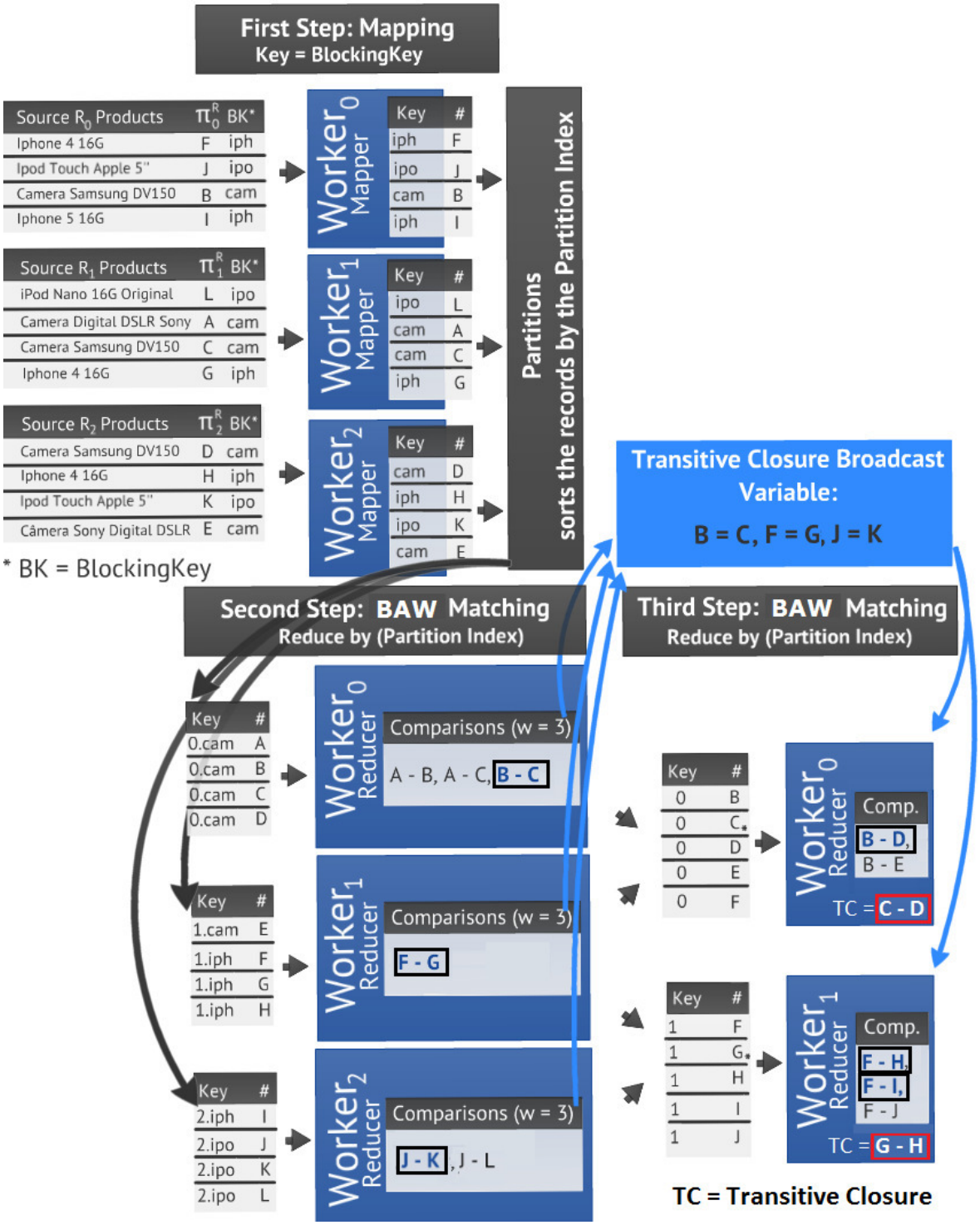

Figure 5.4: Example of dataflow for the *S-BAW* strategy with $w_{initial} = 3$.

regarded as a match (framed). Thus, the following relation is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of earlier detected duplicates within the window and c is the number of comparisons already done also within the window, then w is increased by $w_{initial}-1$ adjacent entities of that duplicate (C). Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of C. However, since the reduce partition does not have sufficient entities to cover an increase of size two, the execution of the EM process stops, the entity C is marked with an "*" and all entities of the window will be processed in the third step.

In the third step, the reducer 0 reads the last window performed by reducer 0 of the second step (B, C and D) and increases the reduce partition with the first $w_{initial} - 1$ entities (E and F) of the subsequent reduce partition (reducer 1 of the second step). Since the window execution during the second step ended in C (marked with an "*"), in the third step, the comparisons execution starts at D (the next entity). Now, w covers entities B, C, D, and E; the new comparisons generated are B-D and B-E. Since B-D is regarded as a matching pair, the w increase test is loaded once more resulting in true ($\frac{2}{2} \geq 0.5$) and the window is increased by two again. In turn, w covers entities B, C, D, E and F. However, F is not considered because it belongs to a different block (iph). Since B-E are regarded as non-matches, there is no need to increase the window. Furthermore, even if B-E was regarded as matches, since E is the last entity of the block cam, the adaptive window will no longer slide.

Note that the example presented in Figure 5.4 refers to the BAW method. However, the Spark-based Adaptive Windowing approach can also execute the BAW variants and other Adaptive indexing methods, such as DCS++. The evaluation conducted and discussed in Section 5.3 will compare the performance of each method.

5.3 Evaluation

In order to properly separate the evaluation discussion regarding the stand-alone EM indexing methods and the parallel approaches, this section is structured as follows. Section 5.3.1 presents the evaluation involving the proposed blocking methods without parallel mechanisms and Section 5.3.2 discuss the experiments regarding the parallel approaches.

5.3.1 Stand-alone Evaluation: BAW and its variants

In this experiment, we investigate the performance of BAW and its variants[8] (OBAW and ROBAW) against the following methods: Standard Blocking (SBM) [14], fixed Sorted Neighborhood (SNM) [38], Adaptive Windowing (DCS++) [28] and the best variant of the generalization of SBM and SNM (Sorted Blocks). To study the efficiency of our retrenchment strategy regardless the influence of the generalization, we also add the retrenched method inside the DCS++ method, which we denoted as RDCS++, and consider it in our evaluation. The methods are investigated regarding the trade-off between the matching quality and the number of comparisons performed by each method. In each experiment, we broadly evaluate the algorithms aiming to investigate their robustness in maintaining the EM quality while the number of pair comparisons decreases.

We ran our experiments on a HP Pavilion P7-1130 machine that has one Intel I5 processor with four cores, 4GB of RAM and 1TB of hard disk. Among the software installed at the machine, Windows 7, 64-bit, JAVA 1.8 and Dude 1.0 were utilized. For each round of the algorithms execution, a new instance of JVM was created to disable information reuse (in memory).We utilized one real-world data source. The DL data source (DS1) is small and contains about 7,800 DBLP and Google Scholar publication entities (based on [53]) presenting the following attributes (fields): id, title, authors, and publication year. DS1 was utilized due to the need of a gold standard to evaluate the methods' matching quality.

Matching Quality vs. Number of Comparisons

For the first experiment, whose purpose was to evaluate duplicate detection results, we analyzed a variety of evaluation metrics [17], [59]. As we intend to evaluate the robustness in the selection of candidate pairs generated by the methods, we use a perfect similarity classifier (C1). Using a look-up in the gold standard, the classifier was able to decide whether a publication entity was regarded as match or not. Thus, all candidate pairs were classified correctly as matches or non-matches (precision always equals to 1). For this reason, in the experiments using C1, we measure the recall (*i.e.*, the fraction of detected duplicate pairs and the overall number of existing duplicate pairs) in relation to the initial window size as a

[8]Executed using an extension of the Duplicate Detection Toolkit (Dude) [27] available at the author's homepage (http://hpi.de/naumann/projects/data-quality-and-cleansing/dude-duplicate-detection.html).

duplicate detection quality indicator.

However, in real-world scenarios, the assumption of a perfect classifier does not hold. Thus, we also studied the methods' behavior when dealing with an imperfect classifier (C2). The similarity between two entities (when applying the C2) was computed using the q-gram similarity over their comparing attributes (*i.e.*, the publication title) and those candidate pairs with similarity value greater than 0.65 were regarded as matches. The value 0.65 was chosen as the similarity threshold based on the execution of DCS++ with six different thresholds (from 0.55 to 0.80) on DS1, as depicted in Figure 5.5, aiming to find the proper threshold to be utilized in our study. Since the usage of an imperfect classifier commonly leads to a precision value less than 1, for the experiments using C2, we measured the F-Measure (*i.e.*, harmonic mean of precision and recall) as a duplicate detection quality indicator. As we can see, when utilizing a threshold value of 0.65, the highest F-Measure was achieved. All methods utilized the same classifiers and also the same $\phi_{matching}$ to decide whether an entity pair is regarded as a match or not. We use F-Measure as a quality metric to compare the different methods.

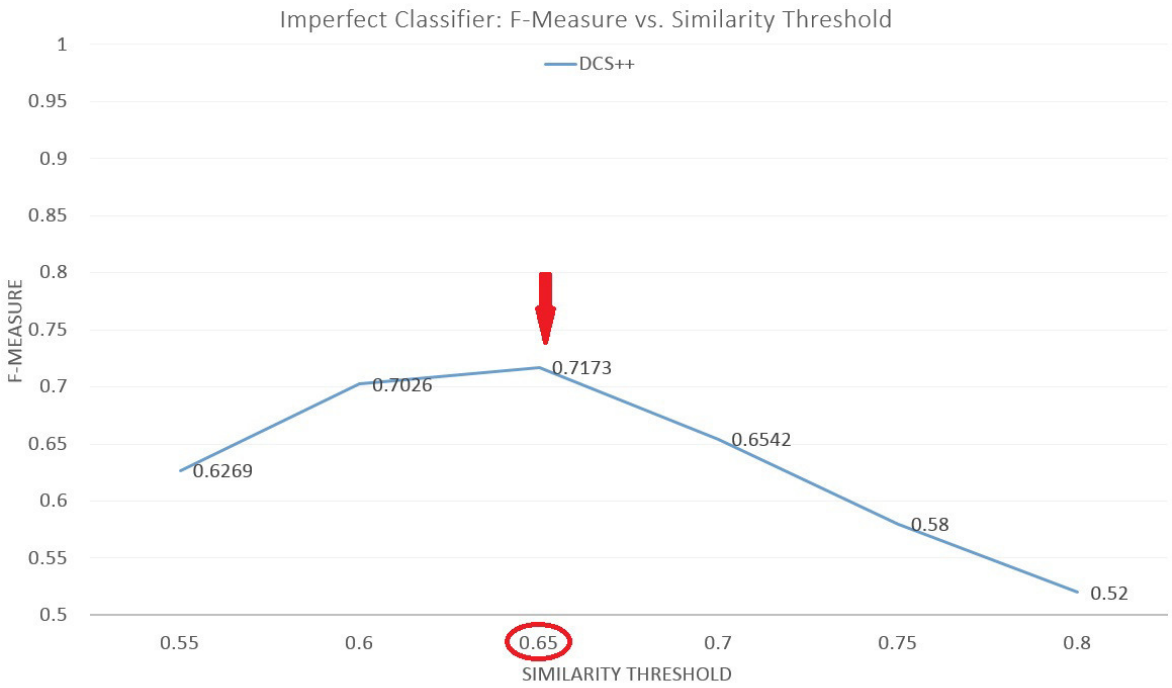

Figure 5.5: Best similarity threshold study for DCS++ varying the similarity threshold value using DS1.

Except for SBM, which works by partitioning the input data into blocks of similar entities, according to the blocking key, and restricting the Cartesian product to compare entities that share the same blocking key, all other methods perform a sliding window to select candidate pairs. In this sense, as window sizes we used the following values: 10, 25, 50, 75, 100, 250, 500 and 1000 for all DL data sources. The threshold ϕ_{ddr} for the DCS++ is $\frac{1}{w-1}$ as suggested in [28] and the sorting key used was the first three letters of the publication title.

Experiments with a Perfect Similarity Classifier

Figures 5.6 and 5.7 were generated based on the same EM execution and depict the performance results using a perfect classifier. Note that both the recall and the number of comparisons of SBM are the same for each window size, since SBM is not a windowing method. As we can see, in terms of recall, DCS++ and RDCS++ achieved the highest values with the largest window sizes, confirming their robustness in performing intelligent guesses about the window size increasing. DCS++ takes advantage of finding most duplicates in this experiment due to its efficient strategy to increase the window in regions of entities with high duplicate detection rate based on the transitive closure combined with the benefit of the perfect classifier usage.

Nevertheless, OBAW and ROBAW achieved high recall values considering the number of comparisons performed. Since OBAW and ROBAW are hybrid methods, the limitation to explore new candidate pairs out of the block bounds, even with larger window sizes, deteriorates a little their recall value. However, note that the recall values of OBAW and ROBAW when using a window size of 1,000 (which generated about 190,000 comparisons) were similar to the DCS++ values when using a window size of 500 (which generated about 3,600,000 comparisons). In addition, the recall values were higher when they were performed with a window size of 500 (which generated about 150,000 comparisons) and DCS++ with a window size of 250 (which generated about 2,249,000 comparisons). These results suggest that the number of comparisons can still be drastically decreased even performing OBAW and ROBAW with larger window sizes.

These execution results also show another interesting achievement. For all window sizes higher than 100, the recall value resulted from the BAW execution is similar to that one resulted from the SBM execution with less than the half number of comparisons performed

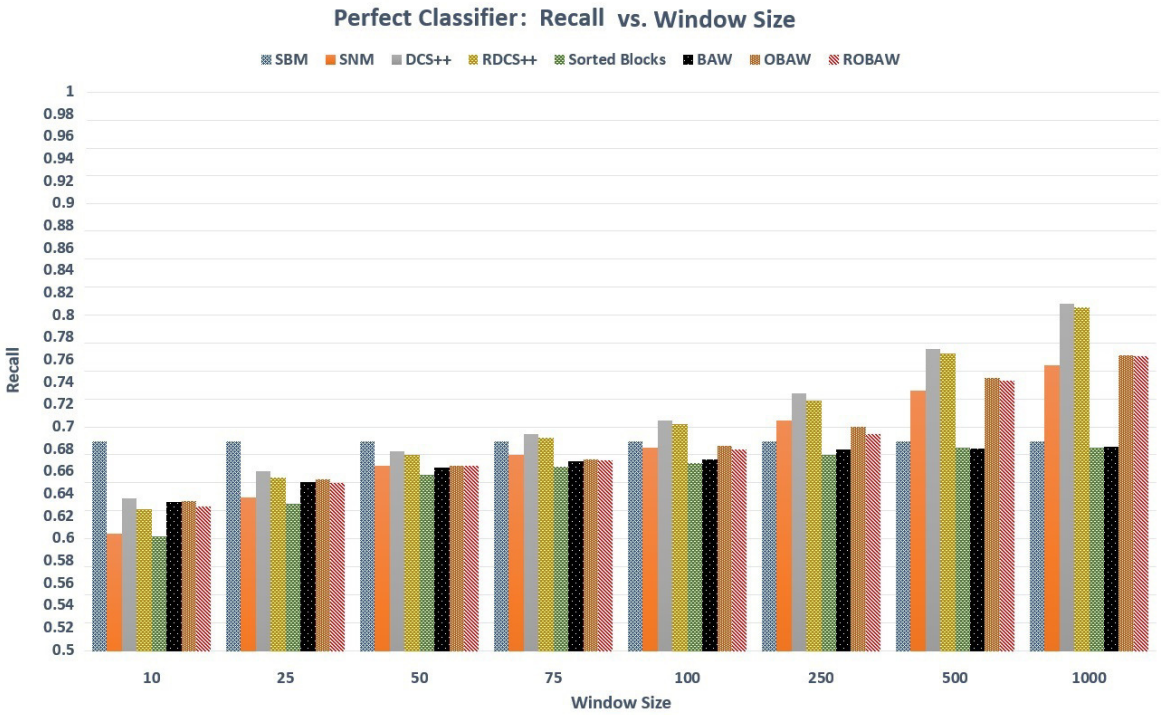

Figure 5.6: Recall collected for DS1 varying the initial window size.

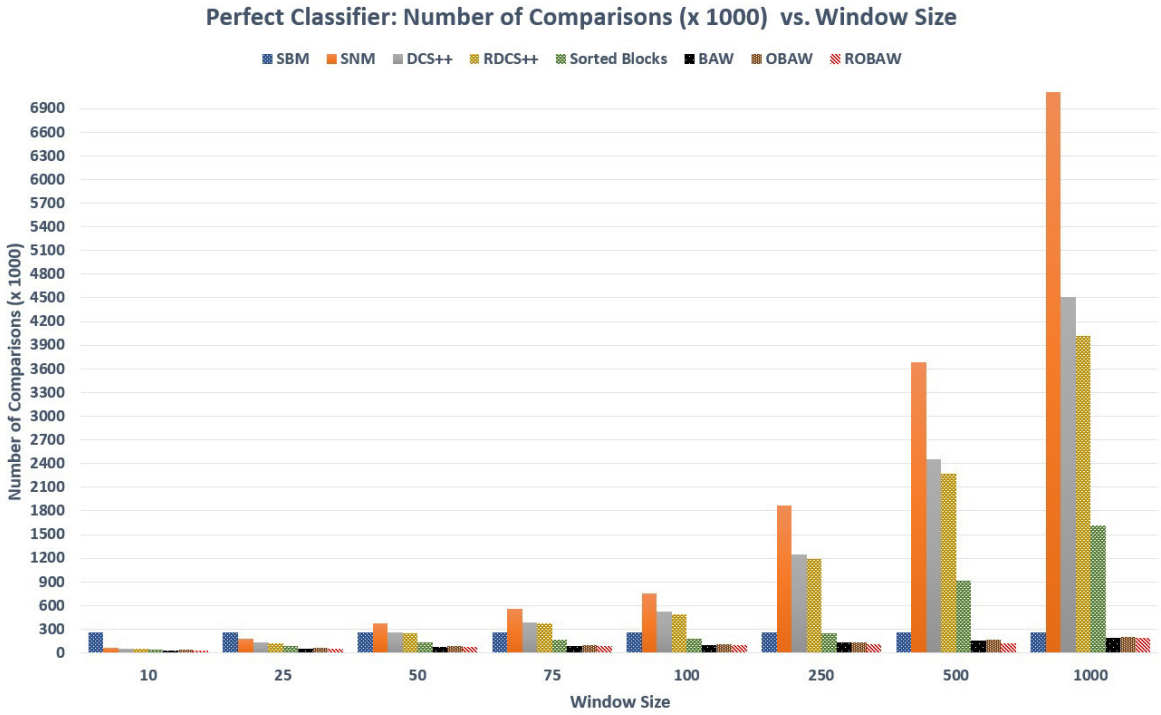

Figure 5.7: Number of comparisons for DS1 varying the initial window size.

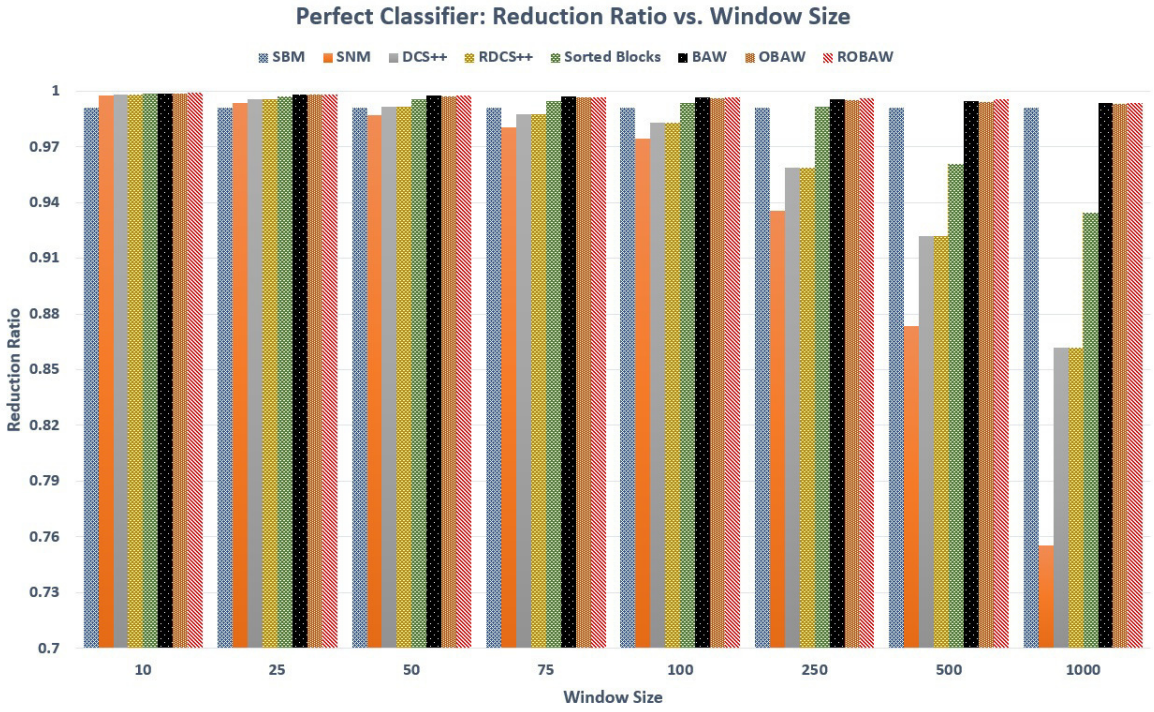

Figure 5.8: Reduction ratio for DS1 varying the initial window size.

by SBM. This result suggests that, depending on the window size, BAW can be an interesting alternative instead of SBM. Actually, the recall values achieved by BAW, SBM and Sorted Blocks were similar for all window sizes higher than 100, however these values were not competitive against the overlapping methods.

Regarding the number of comparisons of each method per window size, depicted in Figure 5.7, the execution of BAW and its variants presented impressive results. The competitive recall values were achieved with window sizes higher than 100 with less than 10% of the number of comparisons performed by DCS++. The main reason for this achievement is that, after the sorting and blocking, the majority of the similar entities are placed together, thus enabling the achievement of a high recall. This is the main reason why the number of comparisons of the ROBAW execution is always less than the one of the OBAW execution. Another important remark is that the retrenchment strategy saved about 10% the number comparisons of the DCS++ method when the window size was set to 1,000 and a high recall value was maintained. Figure 5.8 shows the performance results in terms of reduction ratio (rr), *i.e.*, $rr = 1 - \frac{cc}{allc}$, where cc is the number of candidate entity comparisons generated by an indexing technique and $allc$ is the possible entity comparisons (full naive pair-wise

comparison of all pairs) of each method, per window size. The results reflect the reduction in the number of comparisons discussed earlier.

Experiments with an Imperfect Similarity Classifier

So far we have assumed a perfect classifier function, however it is hard (almost impossible) to develop this in practice. In this subsection, we analyze the effects of an imperfect classifier on the results of the indexing methods. Figures 5.9 and 5.10 were generated from the same execution and depict the performing results using an imperfect classifier. As mentioned earlier, this imperfect classifier decides whether a candidate pair is regarded as match based on a threshold that evaluates the output of a q-gram similarity function (more specifically when the threshold is greater than 0.65).

Note that OBAW, ROBAW, DCS++ and RDCS++ achieved the highest values of F-Measure for the largest window sizes, confirming their robustness in performing block overlapping and intelligent guesses about the window size increasing. This time, DCS++ does not outperform OBAW and ROBAW due the absence of the perfect classifier. With an imperfect classifier, the methods that make use of the transitive closure to increase the window size have their capacity to expand the searching space compromised. In other words, the occurrence of false matching inferences decreases the precision of these methods. This is the main reason why the values of F-Measure achieved in the execution of these four methods are very close for all window sizes.

Regarding the number of comparisons of each method per window size, depicted in Figure 5.10, the execution of BAW and its variants still presented impressive results. The competitive F-Measure values were achieved with window sizes higher than 100 with less than 10% of the number of comparisons performed by DCS++. Specifically in the scenario with a window size equals to 1,000, ROBAW needed only 7% of the number of comparisons performed by DCS++ to achieve a similar F-Measure value. Figure 5.11 shows the performance results in terms of reduction ratio of each method per window size. These results reflect the reduction in the number of comparisons achieved when leading with large window sizes.

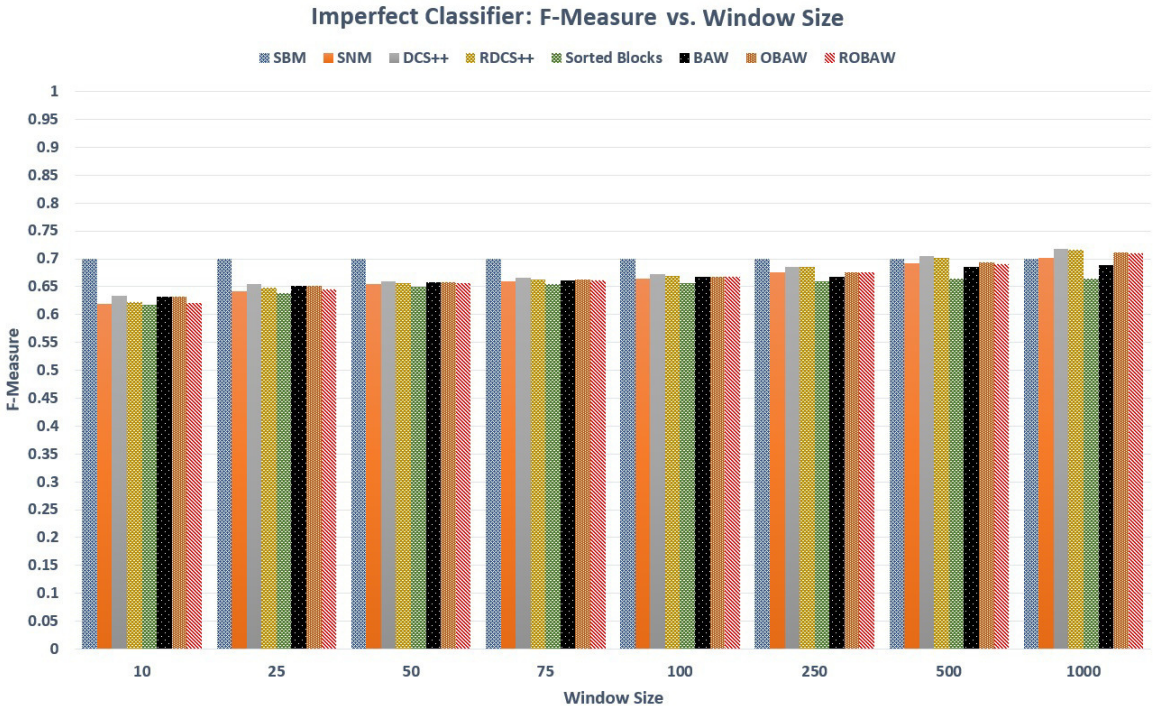

Figure 5.9: F-Measure collected for DS1 varying the initial window size.

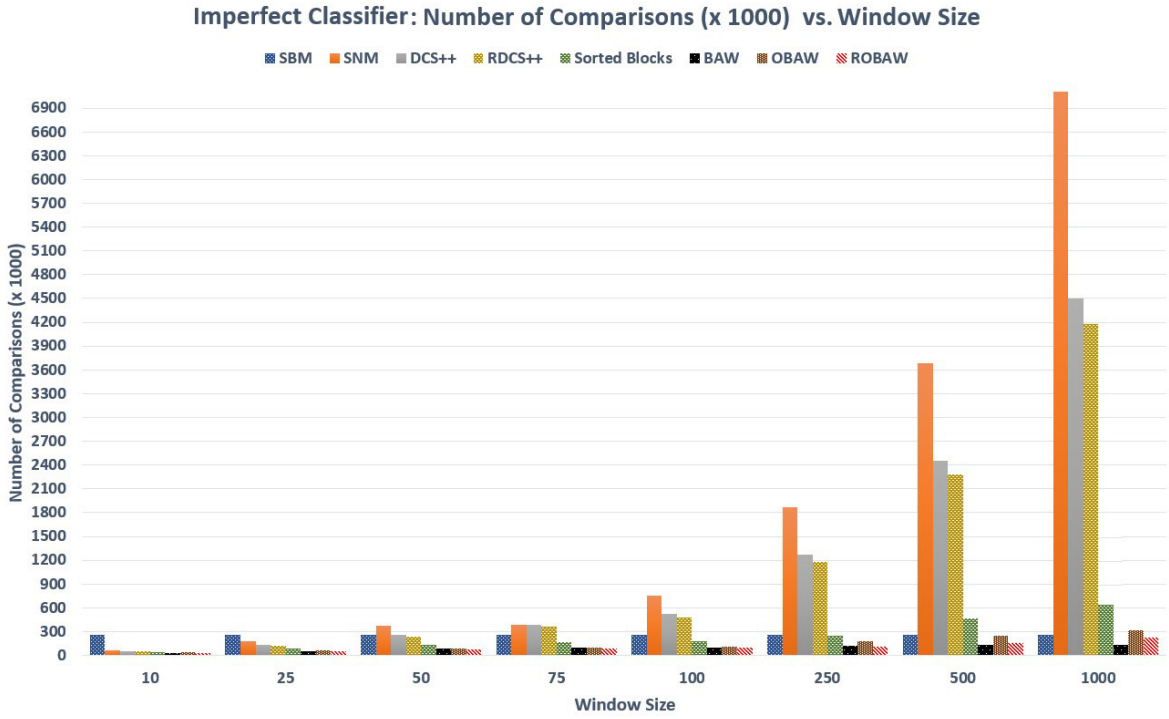

Figure 5.10: Number of comparisons performed for DS1 varying the window size.

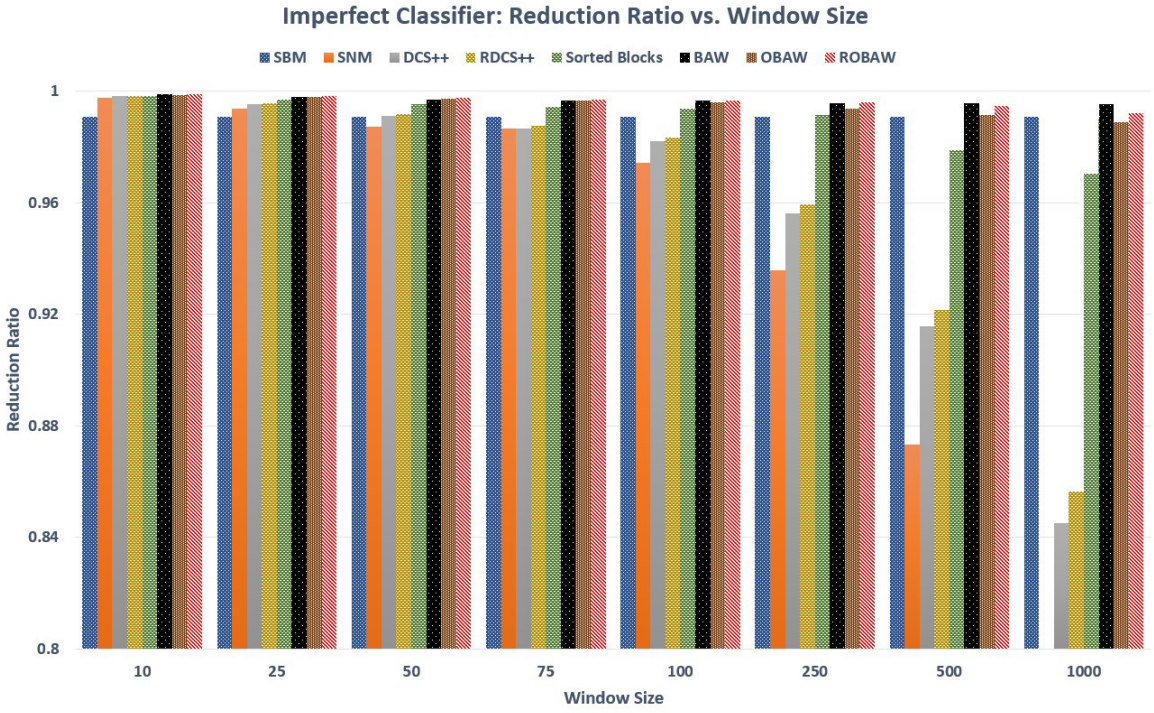

Figure 5.11: Reduction ratio for DS1 varying the initial window size.

Experiments with Multi-pass Variant using both Classifiers

As we have seen, even using a perfect classifier, it is not possible to achieve F-Measure values close to 1. The main cause of this limitation is the chosen blocking key. Since the first three letters of a publication title cannot be considered an ideal blocking key, the sorting phase is not capable of grouping together all similar entities. To overcome this limitation, as mentioned in Subsection 2.1.2, multiple blocking keys may be considered aiming to increase even more the duplicate detection rate. To evaluate the behavior of the methods in the context of multiple passes, we use DS1 (due the availability of a gold standard) and both similarity classifiers. Also, we perform two passes for each method. The first pass is based on the same blocking key utilized previously (the first three letters of the publication title) and the second one is based on the concatenation of the first three letters of the authors' name.

Figures 5.12, 5.13 and 5.14 depict the results of two passes using a perfect classifier. As we can see, the recall for all methods is increased by at least 10% in relation to the experiment with a perfect classifier shown previously (Figure 5.6). The root cause for this achievement is that the employment of multiple blocking keys and matching passes enables the methods

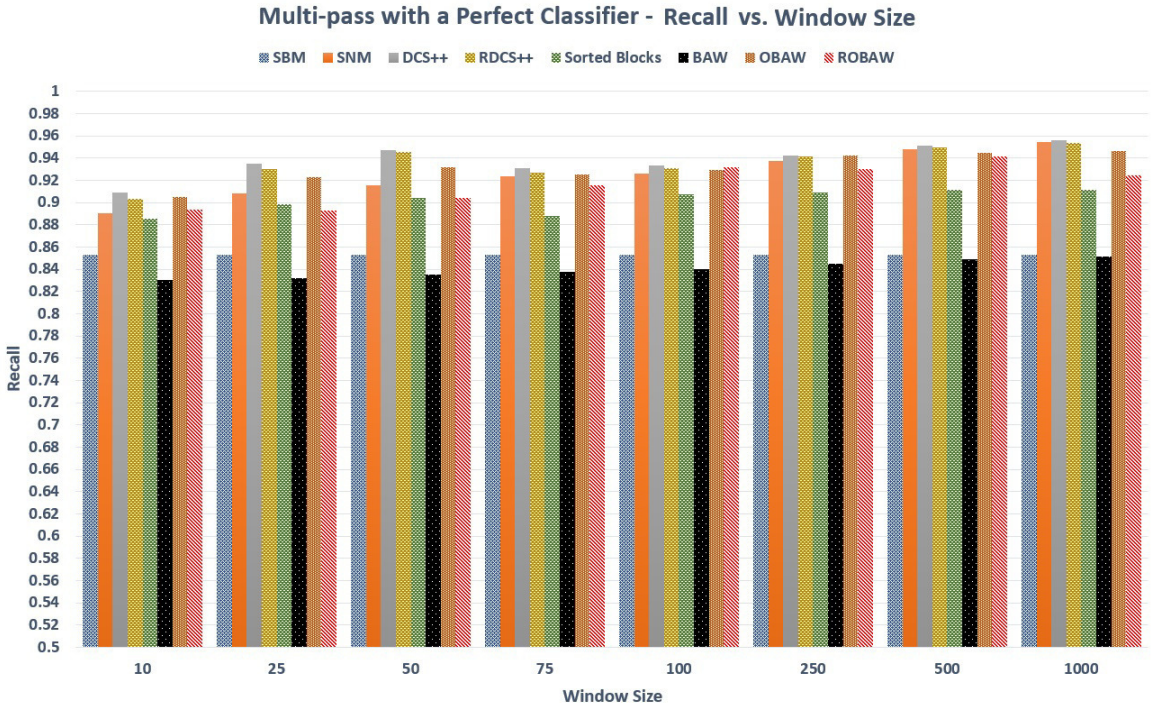

Figure 5.12: Execution results for two passes involving the F-Measure collected for DS1 and a perfect classifier varying the initial window size values.

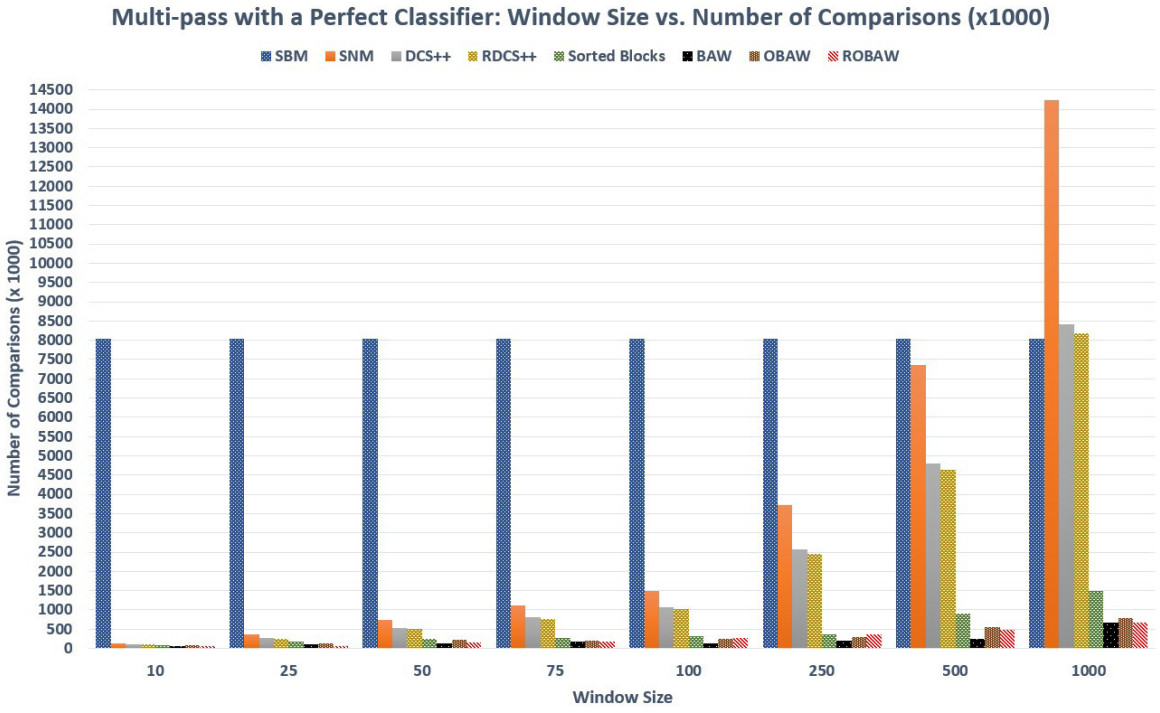

Figure 5.13: Execution results for two passes involving the number of comparisons performed for DS1 and a perfect classifier varying the initial window size values.

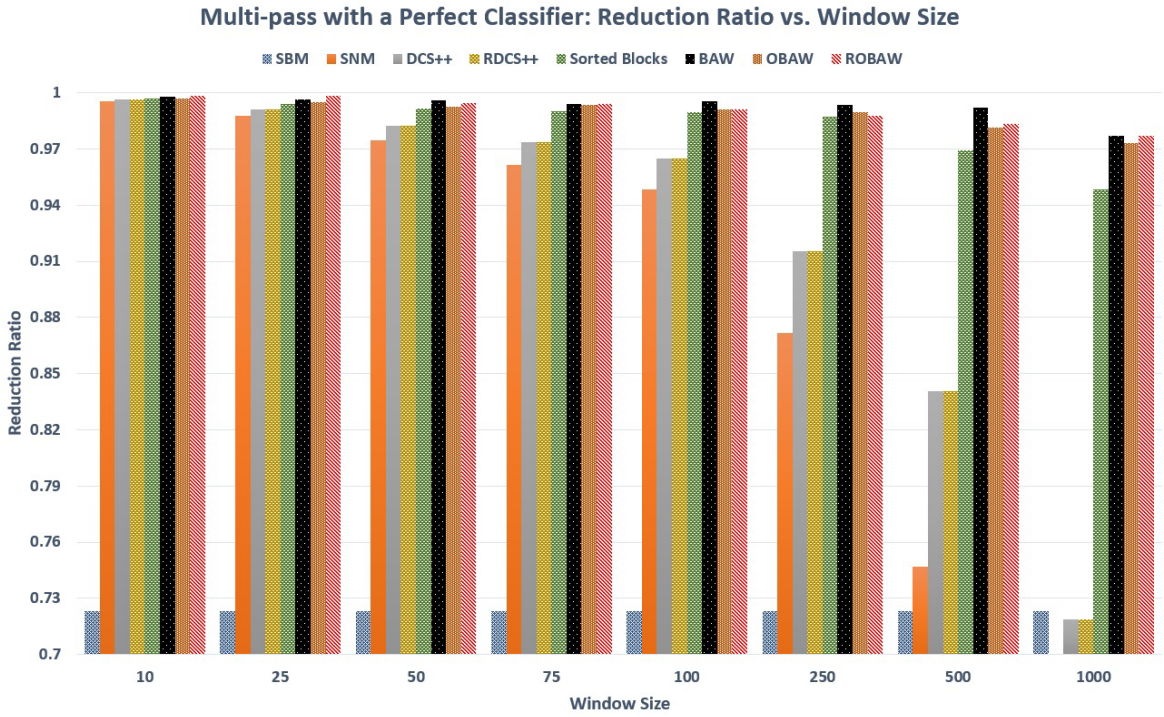

Figure 5.14: Execution results for two passes using the perfect classifier for the reduction ratio varying the initial window size values.

to explore new regions of entities with high probability of finding matching pairs. Another advantage of the multi-pass variant is that the individual passes can be done with relatively small window sizes and thus significantly improve both match effectiveness and efficiency. For instance, as shown in Figure 5.12, with a window size of 250, OBAW and ROBAW achieved one of the highest recall by performing less than 500,000 comparisons.

Due to the exploitation of new regions of entities generated in the second pass, even using a perfect classifier, the recall achieved by DCS++ and RDCS++ did not outperform the one achieved by ROBAW and ROBAW this time. Furthermore, comparing the experiment using window size value equals to 1,000 in Figure 5.12 with the one in Figure 5.6, we can note that the multi-pass variant of ROBAW not only achieved a recall 10% higher than the single-pass DCS++ but also performed a number of pair comparisons eight times smaller.

Figures 5.15, 5.16 and 5.17 depict the performing results of two passes using an imperfect classifier. The evidence found in the multi-pass experiments performed with a perfect classifier were also found here. However, it is important to highlight that in this experiment the F-Measure values of DCS++, RDCS++, OBAW and ROBAW were similar. The main

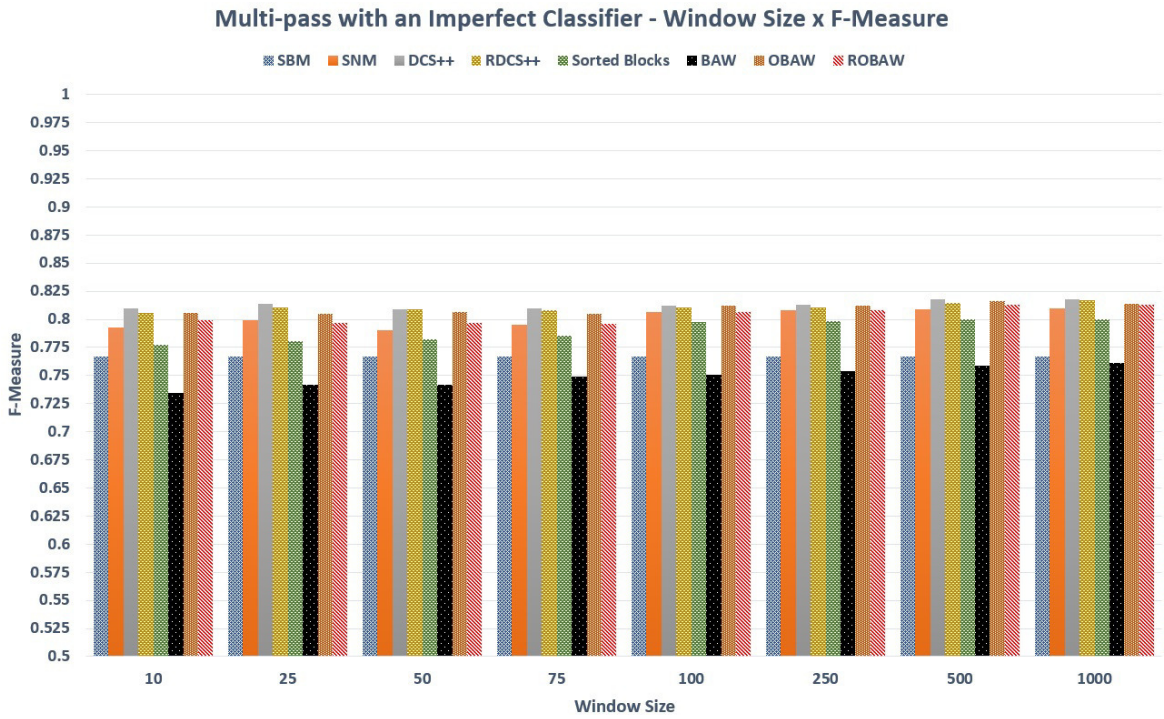

Figure 5.15: Execution results for two passes involving the F-Measure collected for DS1 and an imperfect classifier varying the initial window size values.

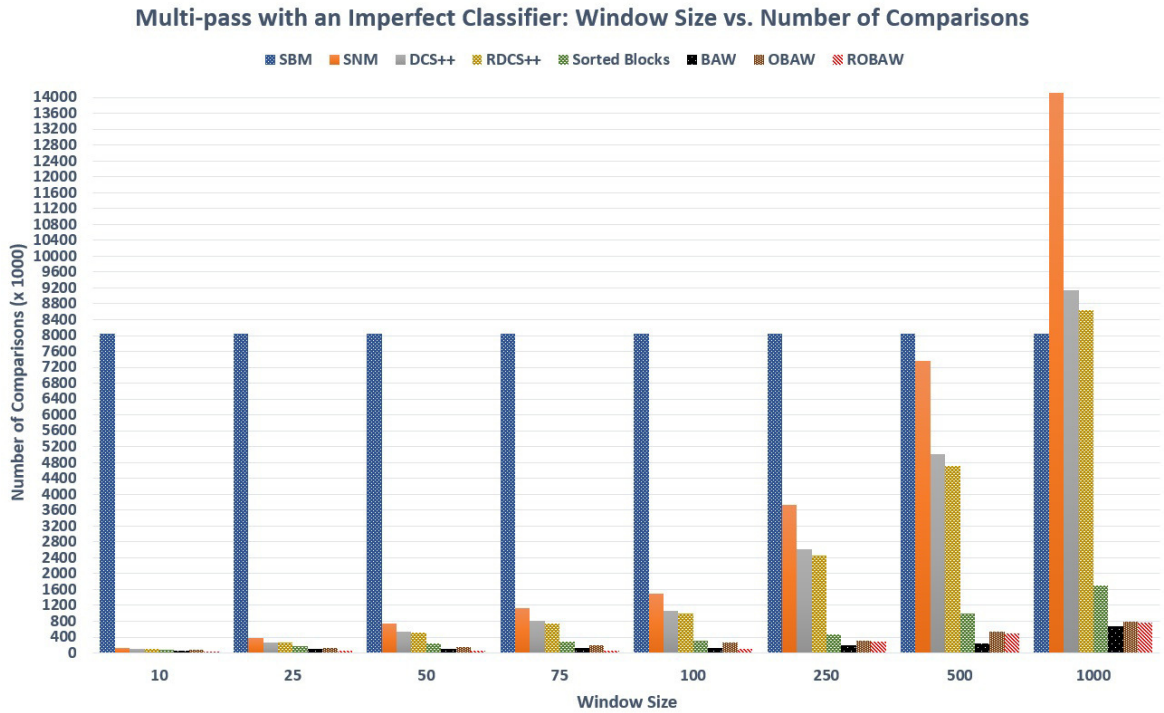

Figure 5.16: Execution results for two passes involving the number of comparisons performed for DS1 and an imperfect classifier varying the initial window size values.

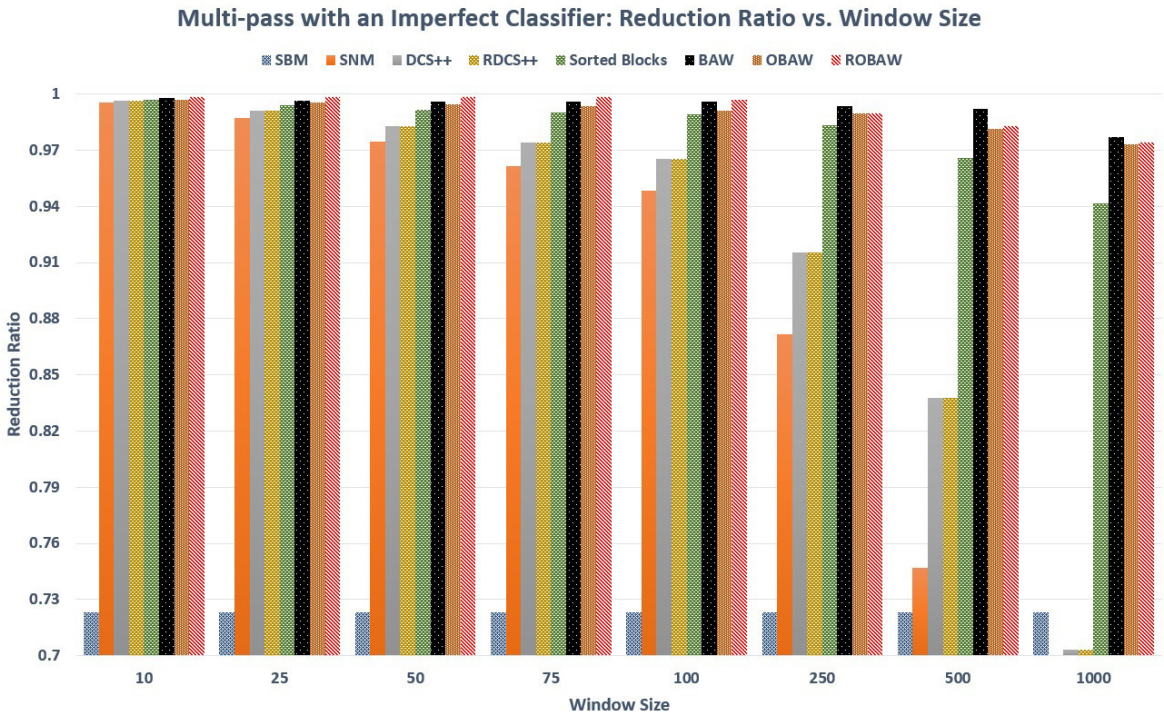

Figure 5.17: Execution results for two passes using the imperfect classifier for the reduction ratio varying the initial window size values.

reason for these results is that the usage of two blocking keys was sufficient to find most of the duplicates. The reduction ratio of each execution is shown in Figure 5.14 and corresponds to the number of comparisons performed by the methods shown in Figure 5.13.

5.3.2 Parallel Evaluation: S-BAW and its variants

In the following, we evaluate *S-BAW*, *S-OBAW* and *S-ROBAW* (the last two are the BAW variants) against the *MR-DCS++*, *S-SNM* (the Spark-based SNM that uses the slidingRDD from library MLlib) and *S-DCS++* approaches[9], regarding three performance critical factors: degree of skewness, the number of nodes available (n) in the cluster environment and the trade-off between the matching quality and execution time. In each experiment, we broadly evaluate the algorithms aiming to investigate their robustness against data skew, how they can scale with the number of available nodes and their robustness in maintaining the EM quality while their execution time decreases.

[9]The codes and data sources are available in https://sites.google.com/site/demetriomestre/activities

This time, we ran our experiments on a 20-node Intel(R) Xeon(R) cluster. Each node has one Intel Xeon processor 4 x 1.0GHz CPU, 4GB memory and runs the 64-bit Debian GNU/Linux OS with a 64-bit JVM, Apache Spark 1.4 and Hadoop 2.6. Each node runs at most two mappers and reducers in parallel. We utilized three real-world data sources. The first data source (DS2) is a sample of the Ask's database that contains about 214,000 question records. The second data source (DS3) is by two orders of magnitude larger and contains about 10.21 million DBLP publication records. The similarity between two entities was computed using the Jaro-Winkler distance (a low computational cost matcher) of the question for DS2 and TF-IDF metric of the authorship for DS3. Those pairs with a similarity $\Phi_{max} \geq 0.7$ were regarded as matches. The third data source is DS1. This third data source was utilized due to the absence of a gold standard for DS2 and DS3, which is necessary to evaluate the approaches' matching quality.

Robustness: Degree of skewness

In this experiment, we study the robustness of the load balancing approaches to handle data skew. In this case, data skew occurs when there is a region of high similarity that enables the increasing of the adaptive window. This window increasing generates new entity comparisons and thus is supposed to leverage the EM execution time during the window sliding on that region. Since the adaptive window increases only when a new duplicate entity is found, for this study, we control the degree of data skewness by modifying the number of duplicates in the data source by adding duplicate entities according to an input percentage. Given a fixed number of entities e in the data source, the number of duplicate entities d is equal to $d = e \cdot s$, such that $s \geq 0$ is the percentage that represents the degree of skewness (the degree of duplication). To exemplify, suppose we have 100 entities (e) and we set s = 0 (no skew), then there are no duplicate entities in the data source. In turn, if we want 10% of duplicate entities, then we set s = 0.1, and thus, $d = 100 * 0.1 = 10$. To compare the load balancing approaches for different data skews, we are interested in the execution time of the approaches when the data source presents regions with high similarity detection.

The execution time of the approaches for different data skews of DS2 (n = 20, m = 40, r = 40, w = 1000) is shown in Figure 5.18. As we can see, the execution time of *S-SNM* is similar in all scenarios. Since *S-SNM* employs a fixed window size, the workload balance is

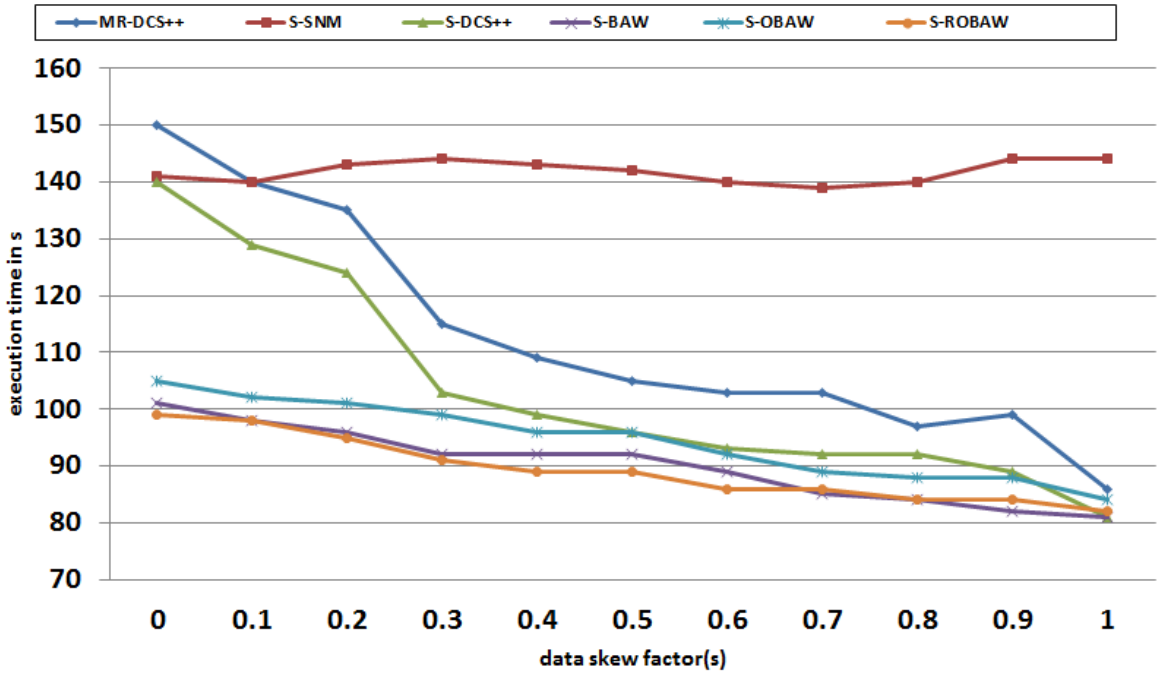

Figure 5.18: Execution times for different data skews using $w = 1000$ (n=20).

uniform. Also note that *MR-*$DCS++$, *S-*$DCS++$ and all Spark-based BAW approaches outperform *S-SNM* in the scenarios which there is duplicate detection. Only in the case that there are no duplicates in the data source, the execution time of *S-SNM* was similar to *S-*$DCS++$. This result confirms that even with the increasing of the window size in regions of high duplicate detection (which leads to more entity comparisons), the usage of the transitive closure mechanism works by minimizing the execution time. According to our experiment, the reason why the execution time is decreased as the degree of skewness grows is that the time saved by the acting of the transitive closure mechanism is higher than the time spent by the additional entity comparisons (generated due to the increasing of the window size).

Therefore, the results we have carried out indicate that there is no scenario in which *S-*$DCS++$ and *MR-*$DCS++$ outperform the Spark-based BAW approaches in terms of execution time due to unbalanced workloads. Only in the case that all data in the data source is duplicated ($s = 1$), the execution time of the *S-*$DCS++$ approach was similar to Spark-based BAW. Also note that *S-*$ROBAW$ outperforms considerably *S-*$DCS++$, especially in the scenarios where $s < 0.3$. Due to the blocked windowing, the Spark-based BAW approaches only need to compare about 12 to 20% of the number of comparisons

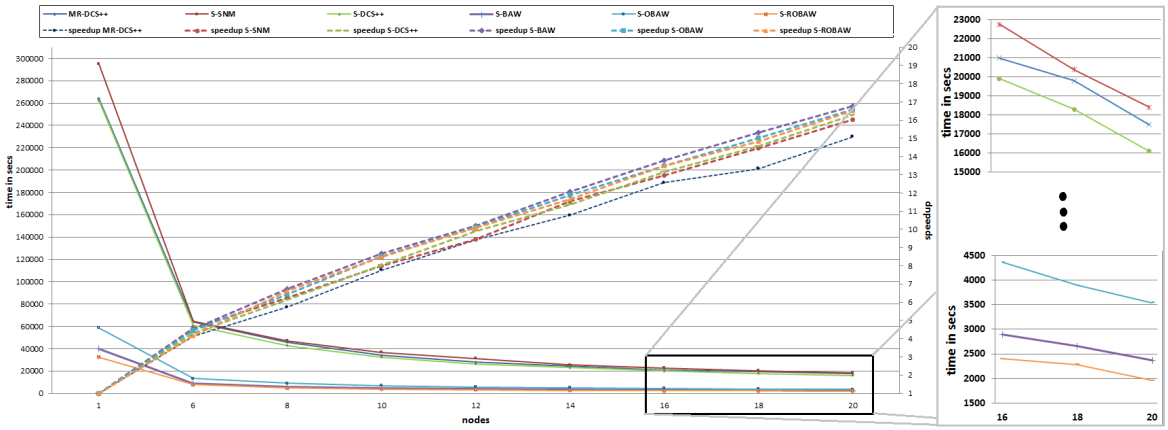

Figure 5.19: Execution times and speedup of the approaches using DS3 (w = 1000)

performed by *S-*$DCS++$ when $s < 0.3$. As long as the number of duplicates increases, the difference in the number of comparisons between the approaches decreases since the acting of the transitive closure mechanism works by skipping a large number of comparisons (due the presence of too many duplicate pairs), and thus, minimizing the execution time.

Scalability: Number of Nodes Available

To analyze the scalability of the approaches, we vary the number of nodes from 1 to 20. Following the Hadoop's wiki[10], for *n* nodes, the number of mappers is set to $m = 2 \cdot n$ and the number of reducers is set to $r = m$. The values of the execution times are shown in Figure 5.19 (DS3). Since the number of comparisons also grows with the window size (increasing the execution time), we defined the same window size for both approaches according to the magnitude of the data source size to avoid benefiting a specific approach. We utilized w = 1000 aiming to verify the scalability of the two approaches when performing a large ($w \geq 400$) window size [50].

As depicted in Figure 5.19, the approaches scale almost equally. These results show their ability to evenly distribute the workload across the workers. However, due to the difficulties presented by the traditional SNM (*S-SNM*) related to the fixed (and difficult to configure) window size, its performance is depreciated by the execution of many unnecessary comparisons. This problem is the main reason why the *S-SNM* approach always performs slower than the other Spark-based approaches and *MR-DCS++* even presenting by design a uniform

[10]https://wiki.apache.org/hadoop/HowManyMapsAndReduces.

Approaches												
	S-SNM		MR-DCS++		S-DCS++		S-BAW		S-OBAW		S-ROBAW	
Window size	w = 500	w = 1000	w = 500	w = 1,000	w = 500	w = 1,000	w = 500	w = 1000	w = 500	w = 1,000	w = 500	w = 1,000
#Comparisons	≈$3.7 \cdot 10^6$	≈$7.1 \cdot 10^6$	≈$2.4 \cdot 10^6$	≈$4.4 \cdot 10^6$	≈$2.4 \cdot 10^6$	≈$4.4 \cdot 10^6$	≈$1.6 \cdot 10^5$	≈$1.9 \cdot 10^5$	≈$2.9 \cdot 10^5$	≈$3.1 \cdot 10^5$	≈$1.6 \cdot 10^5$	≈$2.4 \cdot 10^5$
Execution time (in secs)	79	110	78	105	64	89	34	36	42	45	36	40
Matching Quality												
Precision	92.3%	91.5%	88.9%	88.2%	88.9%	88.2%	92.5%	91.1%	90.1%	89.3%	93.0%	92.4%
Recall	63.9%	64.6%	70.4%	71.0%	70.4%	71.0%	61.5%	62.4%	68.6%	68.4%	65.5%	65.5%
F-Measure	75.5%	75.7%	78.6%	78.7%	78.6%	78.7%	73.9%	74.1%	77.9%	77.5%	76.9%	76.7%

Figure 5.20: Comparison of quality and execution time for the approaches using different window sizes.

load balancing mechanism. Despite the better uniform load balancing mechanism of *S-SNM*, this result highlights the advantage of using Spark to perform EM tasks. Even with a higher number of comparisons, *S-SNM* takes advantage of the RDDs feature provided by Spark (which in turn caches the key-value pairs as RDDs to decrease CPU consumption (when processing text parsing) and overheads (for subsequent iterations). The usage of Spark's features is the main reason why speedup of all Spark-based approaches outperformed *MR-DCS++*.

These results also show that the *S-ROBAW* outperforms all approaches in terms of execution time, despite the slight gain of speedup in favor of *S-BAW*. This speedup gain achieved by *S-BAW* is justified by the usage of a single Spark cycle (one iteration of workers). Since *S-OBAW* and *S-ROBAW* have two Spark cycles, the execution of *S-BAW* presents a lower time spent with overheads, which justifies the high speedup achieved. However, *S-ROBAW* achieves a lower execution time than *S-BAW* and *S-OBAW* due to the usage of the retrenchment mechanism. The retrenchment mechanism is a strategy that can decrease the window in regions of low duplicate detection rate. This mechanism can save a high number of unnecessary comparisons, and thus, decrease the execution time.

Regarding the number of comparisons per window size of each method, the execution of *S-BAW* and its variants presented impressive results for DS3 (about 10 million entities). In the scenario with 20 nodes, the execution of *S-OBAW* (the worst execution time in comparison with *S-BAW* and *S-ROBAW*) decreased the EM task execution time in almost three hours in comparison with *S-DCS++*. Note that *S-ROBAW* outperforms *S-OBAW* execution time in almost 30 minutes. This achievement is justified by the decrease in the number of non-matching comparisons enabled by the approaches' mechanism that bounds the areas where

the adaptive window can act in order to avoid the unnecessary comparisons.

Matching Quality vs. Execution Time

Finally, we study the trade-off between the matching quality, in terms of F-Measure (*i.e.*, the harmonic mean of precision and recall [14]) and the execution time. The data sources used so far could not be applied for this evaluation due to the absence of a gold standard necessary for the match quality calculations. As mentioned earlier, we utilized another publication data source (based on [53]) that contains about 7,800 records and along with the data source there is the gold standard needed to compute the precision, recall and F-Measure metrics. Two window sizes $w = 500$ and $w = 1,000$ were utilized for the approaches. The similarity between two entities was computed using the q-gram similarity over their comparing attributes (*i.e.*, the publication title) and those candidate pairs with similarity value greater than 0.65 were regarded as matches. This time, we run our experiments on four nodes ($m = r = 8$) due to the small size of the data source. Otherwise, if a large number of nodes were utilized, the executions would be instantaneous and it would be difficult to highlight the differences between the execution times.

Figure 5.20 shows the observed execution times along with the number of comparisons and the precision, recall and F-measure collected values for the execution of the approaches with window sizes $w = 500$ and $w = 1,000$. Precision, recall and F-Measure are computed including all pairs that have been directly identified and those generated by transitive closure.

In all cases, Figure 5.20 shows that the number of comparisons performed by *S-SNM* is significantly higher than the amount performed by the other approaches. As an example, *S-SNM* for $w = 1,000$ performs about 2.7 million more comparisons than *S-DCS++* and 6.9 million more comparisons than *S-BAW*. In this case, the difference is also highlighted by the execution time since *S-BAW* (36 secs) outperforms *S-DCS++* (89 secs) by 40% and *S-SNM* (110 secs) by 68%. This indicates that all Spark-based BAW approaches perform efficiently the task distribution among the nodes since *S-SNM* by design has a robust load balancing mechanism (due to the fixed window size) and the execution time difference between the Spark-based BAW approaches and *S-SNM* approach is still significant given the gain with the saved comparisons.

In terms of matching quality, although the slight increase in the precision, due to the

avoidance of performing unnecessary comparisons, *S-DCS++* outperforms the Spark-based BAW approaches in all cases with respect to recall and F-Measure due to its ability to increase the window in regions of high similarity and perform large number of comparisons (see the precision of *S-DCS++*). Nevertheless, *S-OBAW* and *S-ROBAW* achieved high recall values considering the number of comparisons performed. The limitation to explore new candidate pairs out of the block bounds, even with larger window sizes, deteriorates slightly their recall value. However, note that the recall values of *S-OBAW* and *S-ROBAW* when using a window size of 1,000 (which generated about 240,000 comparisons) were approximated to the *S-DCS++* values when using a window size of 500 (which generated about 4,400,000 comparisons). These results suggest that the number of comparisons can still be drastically decreased even performing *S-OBAW* and *S-ROBAW* with larger window sizes.

The shown results confirm that Spark-based BAW approaches are able to achieve a high matching quality even with different window sizes and that they can outperform *S-DCS++* and *S-SNM* not only in matching quality but also in efficiency due to reduced window sizes and balanced distributed workload.

5.4 Final Considerations

In this chapter, we proposed a new generalized algorithm (BAW) and two variants (OBAW and ROBAW) that combines the search space reduction promoted by the Standard Blocking method with the adaptability efficiency of the Adaptive Windowing method (DCS++). We also presented a retrenchment strategy to improve the DCS++ method capable of decreasing significantly the number of irrelevant comparisons when performed with large window sizes. The BAW family methods provide a significant reduction in the number of comparisons enabling the EM task to perform more quickly without significant losses in the duplicate detection quality. Our evaluation showed that the BAW variants outperform the state-of-the-art methods by diminishing the overall number of comparisons and still maintaining the high values regarding duplicate detection rate.

Furthermore, we also proposed *S-BAW* (Spark-based BAW), a novel Spark-based approach for solving the problem of the adaptive SNM parallelization. The solution provides an efficient parallelization of the BAW method and its variants. The approach also addresses

the data skew problem with an automatic mechanism of data partitioning that enables a satisfactory load balancing across all available nodes. Our evaluation on a cluster environment using real-world data sources demonstrated that *S-BAW* scales with the number of nodes available. We compared our approach against two existing state-of-the-art algorithms (*i.e.*, *S-SNM* and *MR-DCS++*) and verified that *S-BAW* overcomes *S-SNM* and *MR-DCS++* in terms of performance (execution time). In the following chapter, we will study how to apply a Spark-based EM adaptive indexing approach for solving an EM problem in the context of urban mobility.

Chapter 6

Using Adaptive Indexing and Spark-based Parallelization to Streamline Bus Trajectories Matching

In this chapter, we discuss and treat a research problem raised during the EUBra-BIGSEA project. Several demands addressed during the project relied on the development of Entity Matching approaches to compose a service called Entity Matching as a Service (EMaaS). One of the problems treated by EMaaS refers to the identification of public transportation trajectories. The EMaaS approach enabled the provenance of high-quality integrated geospatio-temporal training data to support predictive machine learning algorithms used in the application layer of the project. Thus, this smart-city problem gave us the chance to apply the lessons learned during the studies of the main concepts exploited in this thesis. In other words, the solutions discussed in this chapter makes use of the (adaptive) indexing methods and distributed computing to leverage the aforementioned matching problem.

As discussed in Section 2.3, integrating AVL (Automatic Vehicle Location) and GTFS (General Transit Feed Specification) data demands identifying, for each vehicle in the AVL data, which of the trajectories in the informed route the vehicle has followed. More precisely, it is necessary to use (i) a vehicle trip specified as a sequence of coordinates reported through AVL, and (ii) a route id, to identify a bus trajectory among a set of candidates trajectories. A trajectory is specified as a sequence of bus stop coordinates.

Therefore, in this chapter, we investigate the task of identifying bus trajectories from the

sequences of noisy geospatial-temporal data sources. This task is known as a map-matching problem [8] and consists of performing the linkage between bus GPS trajectories and their corresponding road segments (shapes). The goal is to identify bus trajectories efficiently in batch or real-time modes in order to provide high-quality integrated geospatial-temporal training data to support predictive machine learning algorithms of ITS applications and services. By identifying bus trajectories efficiently we mean at proposing indexing-based approaches capable of performing the Map-Matching task in such a way to avoid the execution of unnecessary comparisons. To achieve this, we make the following contributions:

- We propose two novel unsupervised techniques denoted as *BULMA* (**BU**s **L**ine **MA**tching) and *BULMA-RT* (**BULMA R**eal-**T**ime) that are capable of matching a bus trajectory efficiently (in batch and real-time modes, respectively) with the correct shape in cases where there may exist multiple shapes for the same route. We also propose efficient Spark-based approaches for these new techniques;
- We evaluate *BULMA* and *BULMA-RT* against an adapted technique based on the Bag-of-Roads strategy (adapted from [76]). The Bag-of-Roads strategy consists in the usage of a sparse vector containing the number of road segments traversed by a bus b, where its i-th element denotes the frequency of bus b traversing the road segments. We denote this (baseline) **B**ag-**o**f-**R**oads **tech**nique as *BoR-tech*. We also show that *BULMA* outperforms *BoR-tech* by increasing considerably the number of correct bus trajectories classifications. The evaluation is performed using real-world data sources.

6.1 Problem Definition

In this section, we define the Map-Matching problem described in Subsection 2.3.2. For ease of explanation, we directly model the problem and notation to deal with the matching of bus trajectories. We consider a set B of buses such that for each bus $b \in B$ a sequence $g_b = (g_{<b,1>}, g_{<b,2>}, ..., g_{<b,p_b>})$ of measurement points is given, with $g_{<b,i>}$ consisting of the latitude, longitude, timestamp and route identification of i-th georeferenced point of bus b. The sequence is sorted in ascending order of the timestamps. A shape linestring of a predefined bus trajectory is a sequence $s_r = (s_{<r,1>}, s_{<r,2>}, ..., s_{<r,p_s>})$, where $s_{<r,i>}$ consists

of the latitude, longitude and route identification of the i-th shape point. In the context of bus trajectory identification, each bus is assigned to one or more routes and each route has one or more trajectories. Let R be a set of predefined routes such that $r_i = (r_{<i,1>}, ..., r_{<i,q_i>})$ represents the sequence of coordinates for each route $i \in R$. Note that there are ς predefined routes, *i.e.*, $\varsigma = |R|$.

Thus, the problem is to identify the shape line followed by each bus in which the predefined bus trajectories have different starting points and belong to the same route. In this case, we rely on the observation that there should be a group of buses serving to the same route but performing different shapes, as depicted in Figure 2.6. Therefore, the objective is to detect the correct shape of a bus by comparing its trajectory with the set of shapes (predefined bus trajectories geo-spatial representation). Formally, to map-match the trajectories and the GPS sequences of buses, we label l_γ for $b' \in B'$, where $B' \subset B$ is a subset of the buses and $l_\gamma \in S$ denotes the ID of the shape followed by bus b'. The task is to identify the label l_b for all $b \in B'$.

6.2 BULMA technique

Our BUs Line Matching technique, *BULMA*, is based on the intuition that all trips performed by a bus during a certain day can be treated analogously to a DNA chain (*i.e.*, Deoxyribonucleic acid studied in Biology) in the context of multiple sequence alignment. As each trip is associated to a single or multiple candidate trajectories, a DNA sequence to be aligned in the DNA chain can be seen as a shape [29] when arranging the sequences of DNA to identify regions of similarity sequences.

Multiple sequence alignment is an extension of pairwise alignment which incorporates more than two sequences at a time [29]. For example, as depicted in Figure 6.1, the GPS data of all trips performed by a bus during a certain day can be seen as a DNA chain whilst $shape1$ and $shape2$ as sequences of DNA to be matched within the DNA chain. In this example, $shape1$ fits better within the GPS sequence than $shape2$ because $shape1$ covers the largest portion of the GPS sequence. *BULMA* is build on this idea and makes use of map-matching strategies to solve this optimization task.

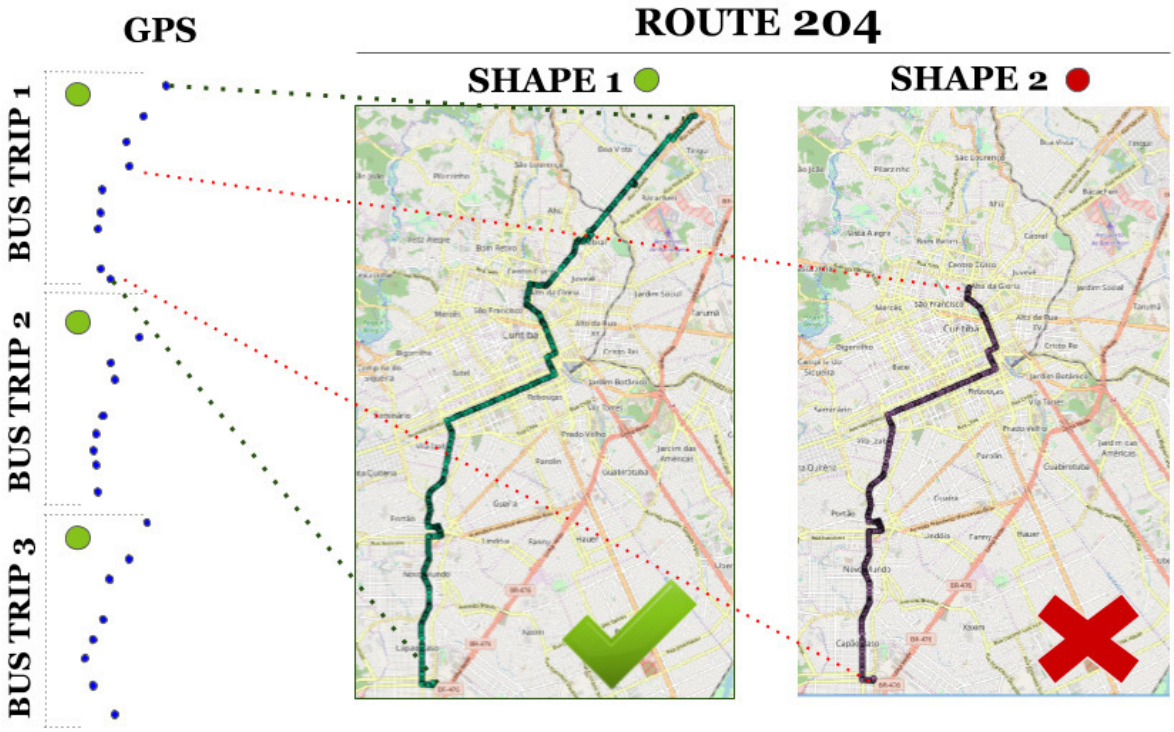

Figure 6.1: Sequence alignment of bus trajectories.

6.2.1 Blocking Strategies

The main strategy utilized by *BULMA* is blocking. Such strategy avoids applying the map-matching technique on the Cartesian product involving all input shapes and GPS data. There is an increasing trend of ITS applications and services being expected to deal with vasts amounts of data that usually do not fit in the main memory of a single machine. If the avoidance of a computational cost in the order of $O(n^2)$ is not guaranteed, it means that such applications and services are ineffective for large data sources. As mentioned in Subsection 2.1.1, blocking techniques reduce the search space and thus lessen the workload caused by the Cartesian product execution while maintaining the Map-Matching effectiveness. For instance, it is sufficient to compare bus trajectories and shapes that share the same route identifier[11].

Besides the usage of the route identifier as a blocking key, we also consider the starting and ending points of the bus trajectory together with a maximum distance radius threshold as a secondary blocking key. The use of this second blocking key is straightforward in the

[11]Route identifier is an attribute of both GPS and shape data.

sense that it provides robustness on blocking similar shapes without suffering a considerable influence of noisy, missing or sparse GPS data along the (GPS) bus trajectory. Applying this blocking strategy further reduces the workload caused by an excessive number of comparisons.

The maximum distance radius threshold indicates if the starting and ending points of a shape match the starting and ending points of the (GPS) bus trajectory, respectively. It is calculated as follows:

$$\phi_{max\_distance} = \frac{\Delta_{TD}}{\#shape\_points * \phi_{PSR}} \tag{6.1}$$

where Δ_{TD} is the traveled distance value calculated using the last shape point, $\#shape\_points$ is the number of points existing in a certain shape and ϕ_{PSR} is the percentage of shape reduction in which the distance between two consecutive points of a shape is still considerable for detecting the starting or ending points of a bus trajectory.

6.2.2 Finding the Correct Shape

The functioning of *BULMA* is detailed in Algorithm 7. As input, *BULMA* receives the content of three files. The first file contains the list of routes R (extracted from the GTFS), the second one contains the list of GPS records G (as described in Table A.2 of Appendix A) and the third one contains the list of shapes S (as described in Table A.1 of Appendix A). The algorithm consists of three steps: Candidate shape selection (step 1); Candidate shape election (step 2); and Trips labeling (step 3).

***Step 1*: Candidate shape selection**. In this step, *BULMA* iterates the list of GPS points of each bus b and selects the nearest starting and ending GPS points according to the (second blocking key) starting and ending points of the shapes that belong to the (first blocking key) route followed by b (lines 3 to 14 of Algorithm 7). Afterwards, the candidate shapes are submitted to a shape election process (line 15 of Algorithm 7).

***Step 2*: Candidate shape election**. In this step, *BULMA* first determines if the candidate shapes are complementary (*i.e.*, shapes that must join other shapes to form a complete route) or circular (*i.e.*, shapes that describe a complete route by itself). This is imperative to decide if multiple or single shapes must be elected. The decision is based on the largest distance

Algorithm 7 BULMA

1: **function** BULMA(R, G, S)
2: **for each** $r \in \mathcal{R}$ **do**
3: **for each** $g_b \in \mathcal{G}_r$ **do**
4: **for each** $s \in \mathcal{S}_r$ **do**
5: $listIndexStartPoints \leftarrow []$;
6: $listIndexEndPoints \leftarrow []$;
7: **for** $index = 1 \rightarrow |g_b.points|, index{+}{+}$ **do**
8: $p \leftarrow g_b.points[index]$;
9: **if** $p.isNearEnoughToStartShapePoint(\phi_{max\_distance})$ **then**
10: $listIndexStartPoints.add(index)$;
11: **else if** $p.isNearEnoughToEndShapePoint(\phi_{max\_distance})$ **then**
12: $listIndexEndPoints.add(index)$;
13: $g_b.addPossibleShapeMatch(s, listIndexStartPoints, listIndexEndPoints)$;
14: **for each** $s \in g_b.possibleShapes$ **do**
15: $listIndexes \leftarrow []$;
16: **if** $|s.listIndexStartPoints| = 0$ **then**
17: $s.listIndexes.add(s.listIndexEndPoints)$;
18: **else if** $|s.listListIndexEndPoints| = 0$ **then**
19: $s.listIndexes.add(s.listIndexStartPoints)$;
20: **else**
21: // This line indicates that the shape is complementary
22: **while** $|s.listIndexStartPoints| > 0$ **and** $|s.listIndex\text{-}EndPoints| > 0$ **do**
23: $s.populateList(listIndexes)$;
24: // Sorts the list of possible shapeLines in descending order
25: // according to the absolute value of first element of its list of Indexes
26: $g_b.findBestShapes()$;
27: $g_b.setUpTrips(g_b.bestShapes)$;
28: **for** $tripIndex = 1 \rightarrow |g_b.trips|, tripIndex{+}{+}$ **do**
29: **for each** $t \in g_b.trips[tripIndex]$ **do**
30: $s \leftarrow t.shapeLine$;
31: **for each** $p \in t.points$ **do**
32: $findClosestShapePoint(p, s)$

between two consecutive points of each shape. In other words, if the distance between the starting and ending points of a shape is larger than a largest distance threshold ld, the shape is marked as complementary; otherwise, it is marked as circular. Next, *BULMA* performs a "best fit" procedure to map-match the sequence of shapes that covers the largest portion of the GPS sequence (of all trips) performed by bus b during the day. If during this procedure one or more trips do not match with any of the candidate shapes, this trip is marked as problematic. A problematic trip occurs due to the presence of noisy, missing or sparse GPS data or the application of an insufficient $\phi_{max\_distance}$ to determine the starting and ending points. For instance, if the $\phi_{max\_distance}$ value is defined too small, starting and ending points may not be detected. On the other hand, if the $\phi_{max\_distance}$ value is defined too large, too many starting and ending points will be considered as candidates.

***Step 3*: Trips labeling**. The last step of the algorithm is concerned with labeling the trips with the selected trajectory(ies). Since the output of *BULMA* is a set of lines containing the GPS points referenced by its closest shape point, in this step, *BULMA* associates to each GPS point the closest point belonging to the selected shape (lines 34 to 41 of Algorithm 7).

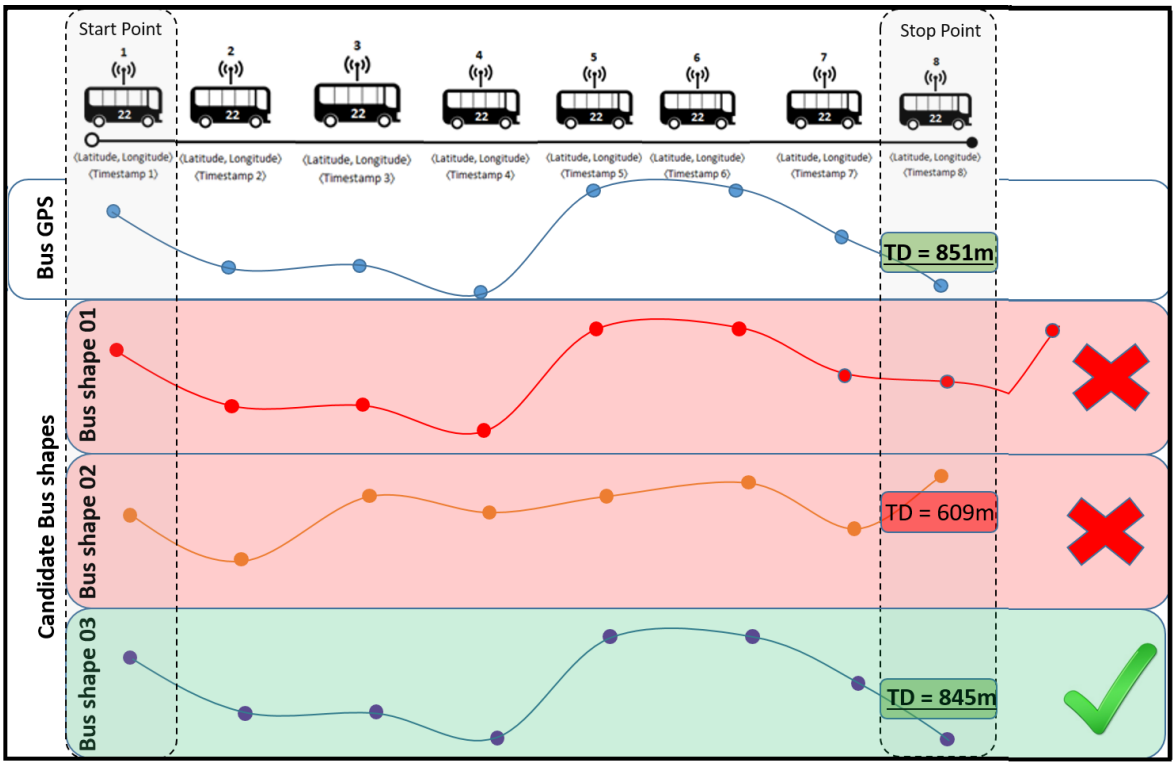

Figure 6.2: Example of BULMA execution for route 022 containing three shapes.

The example depicted in Figure 6.2 shows the strategies utilized by *BULMA*. We have GPS data produced by one bus for route 022, *i.e.*, the bus trajectory. Clearly, most of the comparisons between bus GPS trajectories and shapes will correspond to a non-match result. As mentioned in Section 6.2.1, to eliminate unnecessary comparisons, we use blocking techniques. This is the case of the comparison between *shape 01* and the bus GPS trajectory whose starting and ending points do not match. The threshold strategy (*i.e.*, $\phi_{max\_distance}$) can reduce the search space in this step. Another strategy utilized is to select the shape, among the remaining ones, with the closest traveled distance, extracted from attribute SHAPE_DIST_TRAVELED (see Table A.1) of the last point of the shape, related to the real distance (according to the GPS) traveled by the bus. Thus, since the traveled distance of *shape 03* is the closest to the real distance traveled by the bus, *shape 03* is marked as selected.

Therefore, the *BULMA* output data is provided in text format and records are delimited by newline characters (comma-separated values). Table 6.1 describes the fields available in the output file and provides an example of value for each attribute.

6.3 Parallelizing BULMA Technique

An interesting line of reasoning when we deal with Spark-based *BULMA* is to define an efficient approach (with load balancing handling) using a good partitioning strategy to properly split the incoming data and promote an even workload distribution among the nodes of the cluster infrastructure. Since the *BULMA* technique uses three blocking keys, *i.e.*, route identifier, bus code and starting and ending points, which are able to generate disjoint blocks (partitions) to be processed in parallel, these blocking key candidates must be considered to promote the parallelism of *BULMA*.

Therefore, the route identifier and the bus code were chosen due to the following strategic reasons. First, only GPS trajectories and shapes holding the same route identifier must be compared because the comparison between GPS trajectories and shapes holding different route identifiers would result in a non-match pair. Second, since each bus performs an approximate number of trips per day, each task generated according to the bus code (blocking key) partitioner will produce an approximate number of comparisons (thus leveraging the

Table 6.1: BULMA Output Data

FIELD NAME	***DESCRIPTION***	***EXAMPLE OF VALUE***
ROUTE	The route identifier	519
TRIP_NUM	The trip identifier	1
SHAPE_ID	The predefined trajectory identifier	1708
SHAPE_SEQUENCE	Sequence number of a predefined bus trajectory	3778
SHAPE_LON	The longitude of the closest shape geo-spatial point (to the GPS point)	-25.5180
SHAPE_LAT	The latitude of the closest shape geo-spatial point (to the GPS point)	-49.2797
DISTANCE_ TRAVELED_SHAPE	The traveled distance (in meters) from the beginning of the shape until the current geo-spatial point	6012.21
BUS_CODE	Bus line code	EA183
GPS_POINT_ID	The GPS identifier	1
GPS_LAT	The latitude sent by the GPS	-25.515291
GPS_LON	The longitude sent by the GPS	-49.230788
DISTANCE_ SHAPE_POINT	The distance (in meters) from the GPS geo-spatial point to the closest shape geo-spatial point	7.348101
TIMESTAMP	The interpolated timestamp of the shape point based on the timestamp of the nearest GPS point	13:07:27
STOP_POINT_ID	The bus stop identifier	40442
PROBLEM	Problem identifier (if exists) that determines trajectories deviations.	NO_PROBLEM

load balancing of the approach).

The starting and ending points were not chosen as a partitioner because a portion of the trajectories (grouped by the bus code) could be missed due to the noisy and missing data and, thus, the detection quality of the approach could be damaged. The starting and ending points were only considered in the comparison step (performed after the partitioning step). Thus, we propose a Spark-based *BULMA* for map-matching processing that uses a combination of multiple blocking keys and is performed using multiple Spark transformation steps. Each step is detailed in the following subsections.

6.3.1 Spark-based BULMA (S-BULMA)

To provide an efficient workload distribution among the workers and also enable the *BULMA* technique to be efficiently executed in a Spark workflow, we perform the Spark-based *BULMA* (*S-BULMA*). The map-matching process of *S-BULMA* employs three transformation steps as illustrated in Figure 6.3. The key idea is to improve load balancing by selecting and allocating, to the same worker input partition, the shapes and GPS trajectories that hold the same route identifier, and also to consider the bus code in the case of GPS trajectories. By doing this, the Cartesian product between the number of GPS trajectories and the number of shapes processed by each worker will be approximately the same, leading the worker to perform an approximate number of comparisons. This strategy addresses load balancing because the definition of the route identifier and bus code as the partitioner avoids the overload of a single worker.

Thus, the first step of the *S-BULMA* approach is a Spark mapper that receives as input from each worker a partition of the shape data source. Since the shape input data source consists of a sequence of predefined points sorted by the position of each point in the shape, the first step mounts shape entities and save them in a broadcast variable mapped by the route identifier (blocking key), *e.g.*, route 022 (see Figure 6.3).

In the second step, since the input GPS data source consists of an unsorted collection of GPS record, the mapper (worker) reads the blocking keys of each GPS record, *i.e.*, the route identifier and the bus code, and outputs a key-value RDD pair containing an output key = (the combination of the two blocking keys, *e.g.*, 022.AC104) and value = (GPS record) to enable the key-value RDD pair to be assigned to the same parallel map-matching task. During this

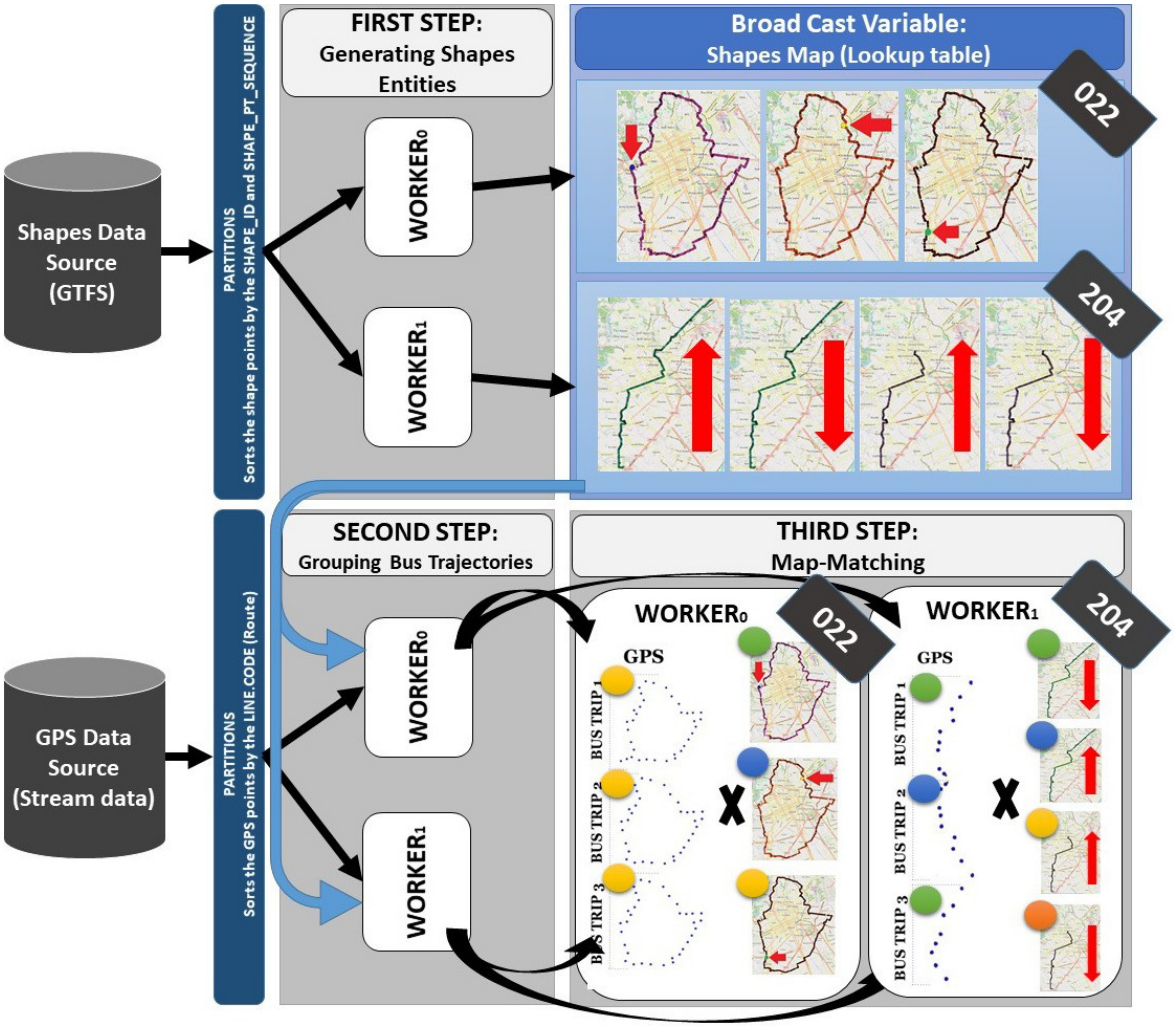

Figure 6.3: General workflow of S-BULMA.

mapper step, the shapes are also read from the broadcast variable and emitted as an output containing an output key = (the combination of the route identifier and the bus code) and value = (shape entity).

To assign a bus code to each shape entity, the shapes must be replicated for each bus performing that route identifier. For example, if three buses (which bus codes are AC104, BR149 and CR014) are performing route 022 (which has three shapes according to Figure 6.3), each one of these three shapes is replicated three times to enable the three resulting tasks (from the partitioning AC104, BR149 and CR014) to process the keys 022.AC104, 022.BR149 and 022.CR014. The aim of this phase is to avoid a data skew problem and promote load balancing by allocating an approximate information/workload size to each worker.

In the third step, the GPS output generated by the second step is sorted by the output key and the timestamp (attribute of the GPS record). Then, the reducer (worker) performs the *BULMA* steps described in Subsection 6.2.2 in parallel for each route and vehicle (route identifier and bus code). As shown in Figure 6.3, each shape is represented by a color (*i.e.*, green, blue and yellow) and each bus trip is colored according to the matched shape (*e.g.*, bus trip 1 is colored with yellow because it corresponds to the shape represented by the color yellow).

6.4 Real-Time BULMA (BULMA-RT)

The real-time *BULMA* challenge has been raised (in the context of the EUBra-BIGSEA project) to support the predictive machine learning algorithms that must be often trained in short periods of time (*e.g.*, every two hours). The objective is to answer the following question: can we train and update predictive models in real-time mode using updated bus trajectory information? Additionally, the objective can be extended to address other types of questions, such as: can we use the *BULMA* output to identify bus trajectories and thus monitor bus trajectories in real-time? Such questions have motivated the investigation of a new technique able to produce integrated geospatio-temporal data of bus trajectories by processing real-time streaming GPS data.

The main difference between the *BULMA* (batch) and real-time *BULMA* is the Map-

matching context. The input of *BULMA* (batch) contains all the GPS information regarding the bus trajectories performed during the entire day. This historical context of bus trips enables a broader analysis of the trajectories performed by the buses such as the definition of complementary or circular trajectories. Unlike this scenario, the input of real-time *BULMA* contains only a few GPS geo-spatial points and the algorithm must identify the correct shape considering these small pieces of trajectory information. This means that, this time, the usage of the ending point of the trips (as part of the blocking key) is no longer possible. Consequently, the context information of the bus trajectories is no longer available.

6.4.1 Blocking Strategies

The main strategy followed by *BULMA-RT* is adaptive blocking. Besides the usage of the route identifier and the bus code as a blocking key, we also consider the starting point of the bus trajectory together with the maximum distance radius threshold (discussed in Subsection 6.2.1) as a secondary blocking key. Since the usage of the ending point of the trips (as part of the blocking key) is no longer possible (due to the lack of such information), *BULMA-RT* is not able to decide if the bus is performing a circular or complementary shape. Thus, it must consider all the search space of shapes sharing the same route identifier. The adaptability of the search space relies on the detection of the ending point. If the ending point is detected (as new GPS data is processed), then *BULMA-RT* is able to decide if the bus is performing a circular or complementary shape and thus decrease the search space (of shapes) to consider complementary or circular shapes.

Another aspect of the search space adaptability is the mechanism developed to reevaluate the matched shape when *BULMA-RT* detects that an erroneous choice was made during the shape selection step. Since the input of *BULMA-RT* contains only a few GPS geo-spatial points, errors can occur in the shape selection step. To detect such error, *BULMA-RT* evaluates if the GPS geo-spatial points are following a trajectory different from the one specified by the initial matched shape. In order to decide if the bus trajectory is projecting far from the matched shape, *BULMA-RT* evaluates if the number of successive GPS geo-spatial points is exceeding a certain threshold (detailed in Subsection 6.4.2). If the number of successive GPS geo-spatial points is exceeding the threshold, it means that the chosen shape was not the right one and a revaluation of the matched shape must be conducted. As we

can see, the search space adaptability discussed in this section is not regarding a windowing algorithm (as discussed in chapters 4 and 5), but regarding a standard blocking algorithm.

6.4.2 Finding the Correct Shape in Real-time

To address the same bus trajectories matching problem in real-time (defined in Section 6.1), we propose the Real-Time BUS Line Matching (*BULMA-RT*) technique. *BULMA-RT* is able to match bus trajectories with the correct shape in real-time when there exists multiple shapes for the same route over streaming data sources. The *BULMA-RT* algorithm is detailed in Algorithm 8 and consists of four steps: Read the context of the GPS geo-spatial point p (step 1); Search for the first geo-spatial point of the trip (step 2); Search for the ending geo-spatial point of the trip (step 3); and Evaluate the matched shape (step 4).

Step 1*: Read the context of the GPS geo-spatial point *p. In this step, the context of the incoming GPS geo-spatial point p is recovered. This context is composed of the follow structure: a key that consists of the combination of the route identifier and bus code (line 2 of the Algorithm 8); the sequence of GPS geo-spatial points (previously processed) describing the current trip of a certain bus (line 3 of the Algorithm 8); the matched shape (if already found); the starting and ending GPS geo-spatial points of the bus trajectory (if already found); and the most similar shapes based on the route (line 4 of the algorithm 8), which are the shapes whose starting point matched with one of the geo-spatial points of the current bus trajectory (according to the threshold explained in Subsection. 6.2.1).

***Step 2*: Search for the first geo-spatial point of the trip**. The second step consists of searching a shape, among the most similar shapes (based on the bus route), in which the shape starting point matches with p. If the initial shape point matches with p, p is defined as the first point of the trip. This step is only necessary if the first point of the trip has not yet been defined (lines 12 and 13 of Algorithm 8). Then, p and the next points will be added to a list denoted as trip progress. The trip progress list contains the sequence of GPS geo-spatial points (which were processed earlier and matched with the shape points) that describes the current trip performed by a certain bus.

***Step 3*: Search for the finish geo-spatial point of the trip**. The third step consists of searching the finish geo-spatial point of the trip. To accomplish that, the trip progress list is kept in memory until the finish point is detected. When it occurs, no more GPS geo-spatial

Algorithm 8 Real-time BULMA

```
1: //p is a incoming GPS point
2: function BULMA-RT(p)
3:   key ← p.busCode + p.route;
4:   trip ← mapOfCurrentTrips.get(key);
5:   similarShapes ← findSimilarShapes(p);
6:   if similarShapes ≠ null then
7:     if trip.hasFoundInitialPoint() then
8:       if trip.isOutOfTheShape() and |similarShapes| >= 2 then
9:         findCorrectShape(similarShapes, trip);
10:      findEndPoint(p, trip, similarShapes);
11:    if trip.hasNotFoundInitialPoint() then
12:      findInitialPoint(p, trip, similarShapes);
```

points are added to the trip and the processing of the next trip (of the bus) is performed. This step is executed only if the first point has been detected (line 10 of Algorithm 8).

***Step 4*: Reevaluate the matched shape**. The last step consists of reevaluating the matched shape when *BULMA-RT* detects that an erroneous choice was made in the shape matching step. To detect such error, *BULMA-RT* evaluates if the GPS geo-spatial points are following a trajectory different from that specified by the initial matched shape (lines 7 and 8 of Algorithm 8). In order to know if the bus trajectory is projecting far from the matched shape, a list of successive GPS geo-spatial points which exceeded the distance threshold (related to the shape) is kept in memory and, if this list reaches a certain size, defined as a user parameter (*e.g.*, five geo-spatial points), it means that the shape chosen was not the right one and a revaluation of the matched shape must be conducted. This step is crucial to detect the bus trajectories deviations or shape changing during the trips.

6.5 Spark-based Real-Time BULMA

To provide an efficient workload distribution among the workers and also enable the *BULMA-RT* technique to be efficiently executed in a Spark workflow, we perform the *S-*

BULMA-RT map-matching process employing three transformation steps as illustrated in Figure 6.4. The key idea is to improve load balancing by selecting and allocating, to the same worker input partition, the shapes and GPS data that hold the same route identifier, and also consider the bus code in the case of the GPS data (following the same steps of the *S-BULMA* approach). By doing this, the Cartesian product between the number of GPS geo-spatial points and the number of shapes processed by each worker will be approximately the same, leading the worker to perform an approximate number of GPS geo-spatial points evaluation (addressing load balancing).

Thus, similarly to the *S-BULMA* design, the first step of the approach is a Spark mapper that receives as input from each worker a partition of the shape data source. Since the shape input data source consists of a sequence of predefined points sorted by the point position in the shape, the first step mounts shape entities and save them in a broadcast variable mapped by the route identifier (blocking key), *e.g.*, route 204 (see Figure 6.4).

In the second step, since the input GPS data streaming consists of an unsorted collection of GPS records, the mapper (worker) reads the blocking keys of each GPS record, *i.e.*, the route identifier and the bus code. The mapper outputs a key-value RDD pair containing an output key = (the combination of the two blocking keys, *e.g.*, 022.AC104) and value = (GPS record) to enable the key-value RDD pair to be assigned to the same parallel map-matching task. During this mapper step, the shapes are also read from the broadcast variable and emitted as an output containing an output key = (the combination of the route identifier and the bus code) and value = (shape entity). The remaining steps are similar to the *S-BULMA* strategy. The only difference is the size of GPS geo-spatial points processed. Note that, in Figure 6.4, only portions of the GPS data are processed by streaming incoming according to the arriving rate of the GPS data streaming.

6.6 Evaluation

In this section, we evaluate *BULMA* and *BULMA-RT* against the *BoR-tech* technique[12] (strategy utilized in [76]). *BoR-tech*, as discussed in Section 3.5, is a technique able to infer transit

[12]The codes and data sources are available in https://github.com/eubr-bigsea/EMaaS

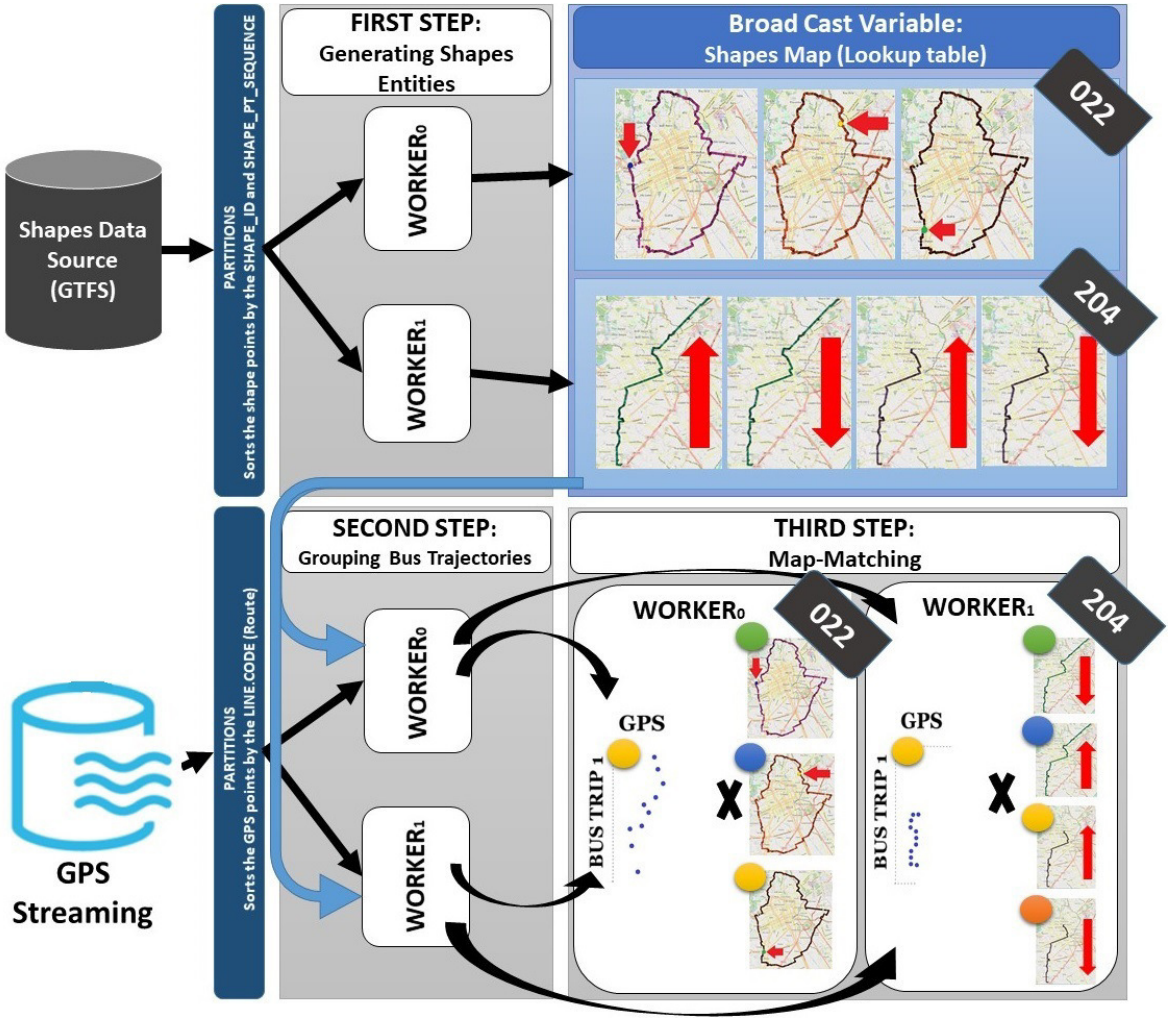

Figure 6.4: General workflow of Spark Streaming-based real-time BULMA.

Table 6.2: DS-GPS statistics

Shape types/Statistic	**Complementary**	**Circular**	**Overall**
Number of buses	9	6	15
Number of trips	147	68	215
Number of problematic trips	48	20	68

network topology from commonly available data feeds[13]. It makes use of a Bag-of-Roads strategy (which is a sparse vector containing the number of road segments traversed by a bus) to enable the correct identification of the shape performed by the bus. We evaluate the techniques regarding two performance critical factors: the trade-off (relation) between the map-matching effectiveness (quality) and the efficiency (execution time) of each technique; and the number of executors available (n) in the cluster environment.

6.6.1 Map-matching Quality vs. Execution Time

To evaluate the first performance critical factor, we ran our experiments on a commodity server machine that has one Intel I7 processor with four cores, 16GB of RAM and 1TB of hard disk. Among the software installed at the machine, Ubuntu 64-bit and JAVA 1.8 were utilized. Two data sources are used: DS-GPS contains one day of GPS information (2016-10-30) from the city of Curitiba (Brazil), and DS-shapes contains the GTFS specification from the same city. As a gold standard for the evaluation, ground-truth data was manually (visually) labeled by a human data specialist considering all the trips of nineteen routes performed by the buses of Curitiba at 2016-10-30. The statistics of this gold standard data source are shown in Table 6.2. It contains 215 trips performed by 15 buses. These trips comprise 147 trips performed by 9 buses on complementary shapes and 68 trips performed by 6 buses on circular (as described in the Subsection 2.3) shapes.

Before performing the map-matching effectiveness evaluation, we first needed to define the ϕ_{PSR} value, *i.e.*, the threshold used to decide if a GPS point is a starting or ending point, to be utilized as a parameter of *BULMA*. To define that, we performed a sensibility analysis

[13] A data feed is an ongoing stream of structured data that provides users with updates of current information from one or more data sources.

using six different thresholds (from 0.05 to 0.15) over the gold-standard data source, as depicted in Figure 6.5. We only show this threshold range because a threshold value under 0.05 or above 0.15 would result in worst values of F-measure. As we can see, the ϕ_{PSR} value chosen was 0.09 due to the higher map-matching effectiveness (in terms of F-measure) achieved. Thus, all experiments of this chapter utilized the ϕ_{PSR} value of 0.09.

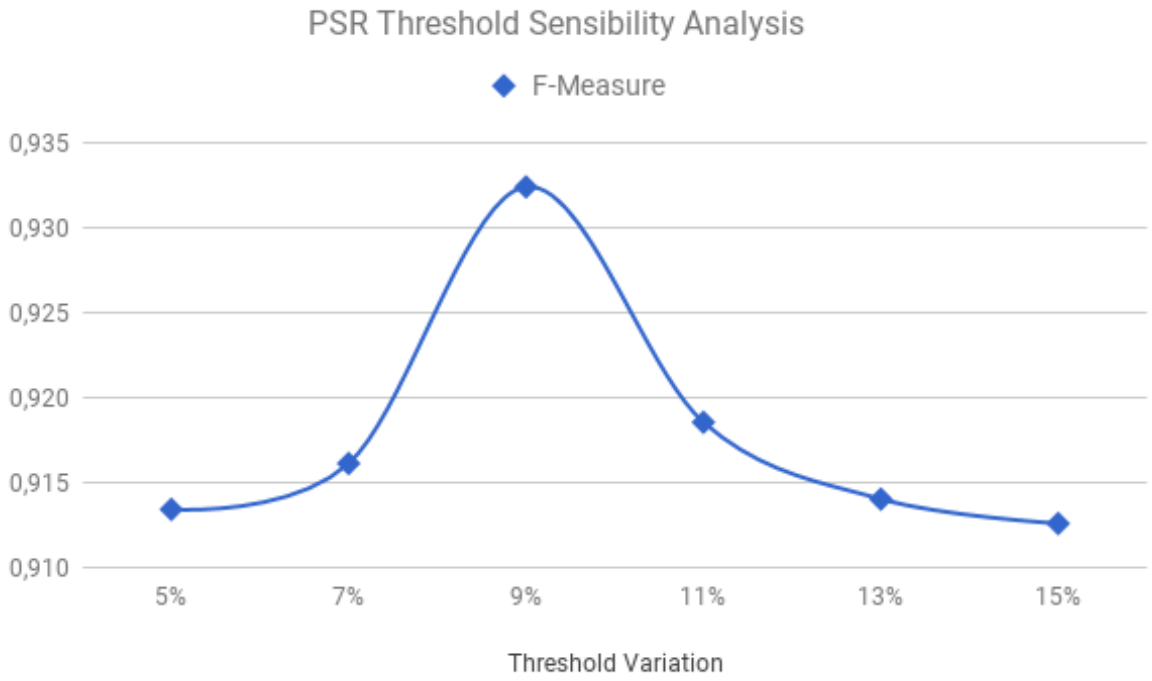

Figure 6.5: Sensibility analysis of ϕ_{PSR}.

Regarding the map-matching effectiveness of the two techniques, Figures 6.6 and 6.7 show the map-matching effectiveness results obtained for the three techniques. The evaluation is organized in two scenarios. The first one is related to the classification performed by the techniques over all the DS-GPS regarding the two types of shapes (described in the Section 2.3): complementary shapes (*i.e.*, shapes that must join other shapes to form a complete route) and circular shapes (*i.e.*, shapes that describe a complete route by itself). The second one is related to the classification performed by the techniques regarding only the problematic trips (see Table 6.2) existing in the DS-GPS data source. The status of problematic is defined based on the impossibility to detect the starting point (starting or ending points in the case of *BULMA*) properly. This impossibility can occur due to the presence of noisy, missing or sparse GPS data or the usage of an insufficient threshold (able to determine the staring and

ending points). Thus, all trajectories which present such impossibility of detection for both approaches are marked as problematic.

	Execution Time (s)			F-Measure		
Type of trajectory	Bor-Tech	BULMA	BULMA-RT	Bor-Tech	BULMA	BULMA-RT
Complementary	**13**	21	15	0.94	**0.98**	0.94
Circular	**13**	19	16	0.26	**0.87**	0.75
Overall	**17**	25	19	0.70	**0.94**	0.86

Figure 6.6: Comparative performance of all bus trips in DS-GPS

Regarding the first scenario, Figure 6.6 shows the observed execution times along with the F-measure values collected for the execution of the three techniques over common routes existing in the DS-GPS data source. As we can see, in all cases, none of the approaches achieve the (highest) F-measure of 1.0. The main reason for this result is related to the significant amount of sparse and missing GPS data within a few bus trips. Such situation leads the techniques to make erroneous decisions during the detection of starting and ending points.

In particular, there is a higher gain in effectiveness when using *BULMA* to perform the map-matching among circular shapes (as in the case exemplified in Figure 2.6). *BULMA* achieved a F-measure of 0.87 against 0.26 of *BoR-tech* for the circular routes. This result shows the lack of robustness of techniques that do not account for detecting the correct shape among multiple shapes that refer to the same route. *BULMA* (0.87) also outperformed *BULMA-RT* (0.75) in the case of circular shapes. This result occurs due to the robustness of *BULMA* in making a late decision about the matched shape based on the context of the trips performed by the bus during the day. Since *BULMA-RT* makes the decision about the matched shape in the beginning of the bus trip, it is prone to make wrong decisions (although rectifying them later).

Figure 6.6 also shows that *BoR-tech* has shorter execution times than *BULMA* and *BULMA-RT*, as it does not employ any computation to treat the problem of multiple trajectories in the same route. On the other hand, *BULMA* provides higher map-matching effectiveness than *BoR-tech* for all cases. Since *BULMA* selects the best sequence of shapes associated to the entire trajectory performed by a bus during a day, it is able to optimize the

"best fit" sequence of shapes according to the trajectory performed by the bus.

Figure 6.7 shows the observed execution times along with the number of buses and F-measure values collected from the execution of the three techniques over problematic routes existing in the DS-GPS data source. Figure 6.7 shows the results regarding the execution of both techniques over 68 problematic trips (which have noisy, missing or sparse GPS data). Although the techniques have marked these trajectories as problematic, *BULMA*, *BULMA-RT* and *BoR-tech* were capable of achieving high map-matching effectiveness results (in terms of F-measure), as shown in Figure 6.7. Considering that the three techniques have presented high map-matching effectiveness, *BoR-tech* and *BULMA-RT* outperformed slightly *BULMA* due to the lack of mechanisms of the former to detect ending points and, consequently, the context of the trips performed along the day.

	Execution Time (s)			F-Measure		
Type of trajectory	Bor-Tech	BULMA	BULMA-RT	Bor-Tech	BULMA	BULMA-RT
Problematic	**11**	17	37	**0.91**	0.89	0.91

Figure 6.7: Comparative performance considering only the noisy, missing and sparse bus trips in DS-GPS data source

Regarding the execution time analysis of the three techniques for one node, we utilized a second GPS data source (DS-GPS2) which contains five days of GPS information (from 2017-05-22 to 2017-05-26) collected from the public transportation agency of Curitiba. Thus, we vary the number of days processed from 1 up to 5. Since we were able to conclude about the scalability behavior of the techniques with few experiments due to the linear increasing fashion of the results, we utilized five days in this experiment. The resulting execution times values are shown in Figure 6.8. As we can see, *BoR-tech* outperforms *BULMA* and *BULMA-RT* in terms of execution time. The main reason for this result, as discussed earlier, is due to the lack of more complex mechanisms to treat the problem of multiple shapes that describe the same route. *BULMA* spends a significant time considering the context aspects of the trajectories performed by a bus, such as the analysis of all trips performed along the day. *BULMA-RT* spends a considerable time analyzing and "correcting" (if it is the case) the matched shape. On the other hand, *BoR-tech* is designed to collect

the most promising (similar) shapes according to a specific sequence of GPS data and does not consider context aspects of the overall bus trajectories. As the number of days increases, the number of executions involving more complex analysis of the contextual aspects of the trajectories increases considerably.

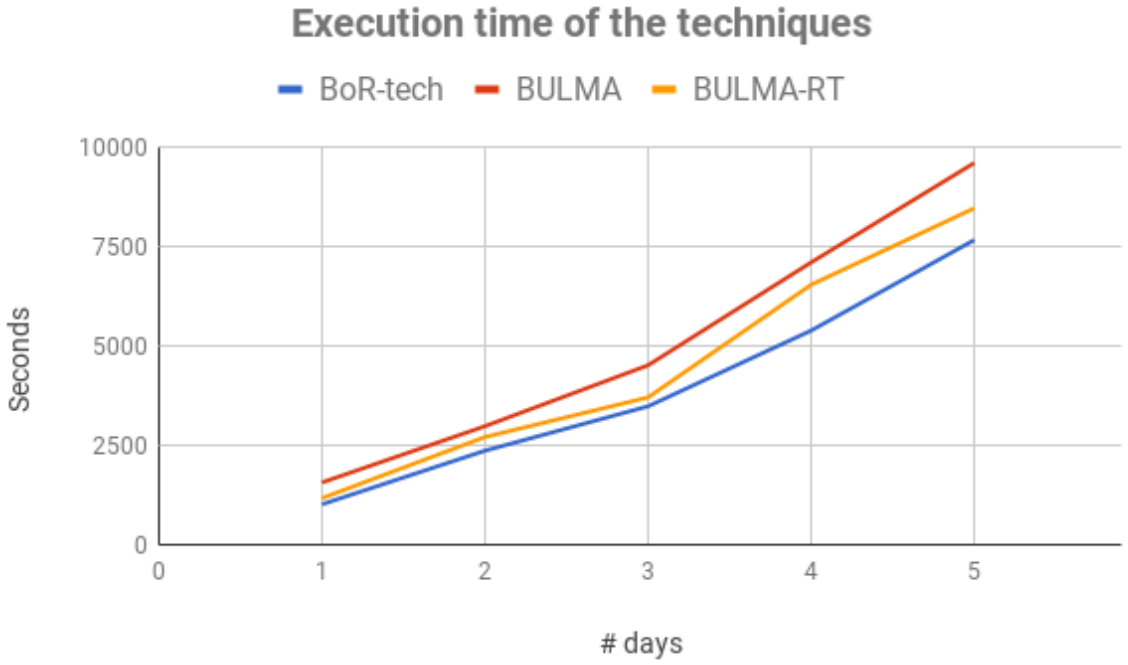

Figure 6.8: Execution time of each technique varying the number of days processed.

6.6.2 Scalability: Number of Executors Available

Based on the results shown in Figure 6.8, it is important to highlight the main motivation behind the Spark-based proposes. Note that, for one executor, *BULMA* takes almost 160 min to process the map-matching of five days of GPS data, showing thus the heaviness of this data-intensive task. Therefore, to analyze the scalability gain of the three Spark-based approaches, we evaluate *S-BULMA*, *S-BULMA-RT* and *S-BoR-tech* aiming to investigate how they scale with the increasing number of available executors (nodes) n. To achieve this, we ran our experiments on a cluster composed of eight virtual machines. Each one has four cores, 8GB of RAM and 500GB of hard disk. Among the products of software installed at the machine, Ubuntu 64-bit and JAVA 1.8 were utilized. For each round of the techniques execution, a new instance of JVM was created to disable information reuse (in memory). We

utilized DS-GPS2 and DS-shapes data sources for this evaluation.

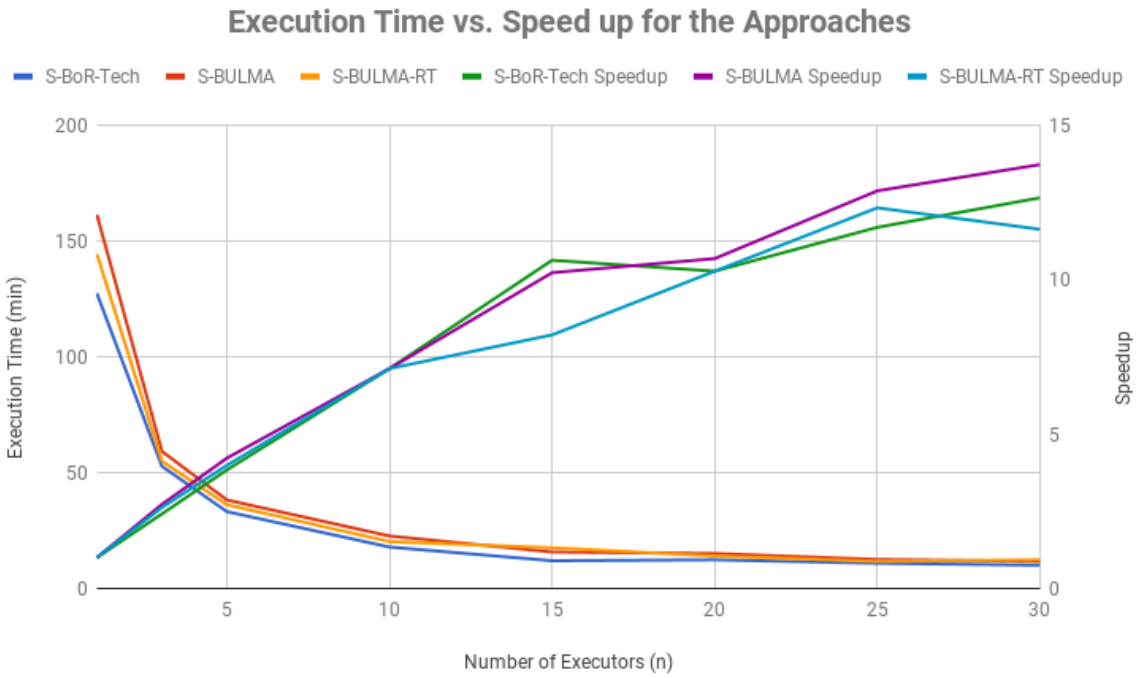

Figure 6.9: Execution time and speedup for BULMA.

As shown in Figure 6.9, we can note that the approaches scale almost linearly up to five executors showing their ability to evenly distribute the workload across the workers. The load balancing was properly treated when the blocking key based on the combination of the route identifier and bus code was utilized as a partition splitter. The usage of the route identifier promotes a significant reduction in the number of shapes to be compared (at most six shapes). Such strategy increases the granularity of parallel tasks favoring thus load balancing. However, with about seven workers, the parallelism was compromised due to the presence of a much larger amount of executors needed to process all workload generated. Note that the execution time stabilizes when more than six workers were utilized. Also note that the execution time (of *S-BULMA*) decreases from almost 160 min (for one worker) to about 18 min (for 15 workers).

6.6.3 Threats to validity

Aiming to full disclosure the validation process of *BULMA* and *BULMA-RT* techniques, it is important to highlight two limitations regarding this validation. First, the threshold value

of ϕ_{PSR}, used (in the map-matching effectiveness evaluation) to decide whether (or not) a GPS point is a starting or ending point, is only applicable to the DS-GPS data source. This means that a sensibility analysis of ϕ_{PSR} must be conducted aiming to define a proper ϕ_{PSR} value to other GPS data sources. Second, the *BULMA-RT* technique was designed to be executed over streaming data sources. Since we focused on the evaluation of the *BULMA-RT* effectiveness over static data sources (to compare it with a state-of-the-art technique), *BULMA-RT* was not properly evaluated in an appropriate streaming scenario.

6.7 Final Considerations

In this chapter, we propose *BULMA* and *BULMA-RT*, two novel map-matching techniques for solving the problem of identifying bus trajectories referenced by multiple shapes in batch or real-time modes, respectively. We also propose the efficient parallel map-matching approach for both techniques. The solutions are capable of identifying bus trajectories from sequences of noisy geospatio-temporal voluminous data sources by using the route identifier and the starting and ending points (of the bus trajectory) together with a dynamic radius threshold as blocking keys to minimize the workload caused by too many comparisons. The technique also considers the context aspects of the trajectories performed by a bus, such as the analysis of all trips performed by that bus along the day, to make better choices about the correct shape followed by the bus.

Our evaluation using real-world data showed that *BULMA* and *BULMA-RT* achieve high levels of map-matching effectiveness. We compared *BULMA* and *BULMA-RT* techniques against an adapted technique based on the Bag-of-Roads strategy, *BoR-tech* (adapted from [76]), and verified that although *BULMA* and *BULMA-RT* perform slower than *BoR-tech* (in terms of execution time), *BULMA* and *BULMA-RT* overcome *BoR-tech* in map-matching effectiveness terms.

Chapter 7

Conclusions and Future Work

This chapter presents the conclusions of the study and the main perspectives for the future work.

7.1 Conclusions

Considering that the efficient parallelization of the EM adaptive indexing methods and real-time EM approaches are open research questions in literature, the general hypothesis of this thesis was to evaluate if the proposition of new approaches for adaptive indexing of the EM task and real-time EM in parallel are able to reduce significantly the execution time of the EM task on a large scale maintaining with small losses the quality levels of EM results. Thus, the main objective of this work is the proposal of MapReduce and Spark-based EM adaptive indexing approaches, as well as a Spark streaming-based real-time EM approach. In this context, different approaches to reduce the execution time of the parallel EM approaches executed in cluster environments are investigated. Therefore, it was proposed: i) an efficient EM approach using MapReduce based on a well-known state-or-the-art adaptive indexing method; ii) an EM adaptive indexing method that combines the traditional blocking method with the adaptive indexing method, as well as the approach for the efficient Spark-based parallelization of these new methods; iii) an EM indexing method, in the context of bus trajectories matching in batch mode, as well as the Spark approach for the efficient parallelization of this new method; iv) an EM adaptive indexing method, in the context of real-time bus trajectories matching, as well as the Spark streaming approach for the efficient

parallelization of this new method.

In addition, experimental evaluations were carried out to validate the efficacy and efficiency of the approaches and methods proposed in the work. Such evaluations indicate, in short, that the new approaches present solutions that significantly decrease the execution time of the EM task and achieve efficacy results equivalent to those achieved by the competing methods. Thus, the evaluation results have given support to the general hypothesis of the work.

Regarding the adaptive SNM parallelization problem, we proposed a novel Multi MR-based approach, known as multi-pass *MR-DCS++*. The solution provides an efficient parallelization of the $DCS++$ method [28] by using multiple MR jobs and applying a tailored data replication during data redistribution to allow the resizing of the adaptive window. The approach also addresses the data skewness problem with an automatic mechanism of data partitioning that can be combined with *MR-DCS++* to ensure a satisfactory load balancing across multiple available nodes. Our evaluation on a real cluster environment using real-world data demonstrated that *MultiMR-DCS++* scale with the number of available nodes. We compared our approach against an existing one (*RepSN*) for single and multi-pass and verified that *MultiMR-DCS++* in both cases overcomes *RepSN* in performance (execution time and matching quality) terms.

Aiming to improve the EM performance gain of the adaptive methods, we proposed a new generalized algorithm (BAW) and two variants (OBAW and ROBAW) that combine the search space presented by the Standard Blocking (SBM) method with the efficiency of the Adaptive Windowing method (DCS++). We also presented a retrenchment strategy to improve the DCS++ method which is capable of decreasing significantly the number of irrelevant comparisons when performed with large window sizes. The BAW family methods provide a significant reduction in the number of comparisons enabling the EM task to perform more quickly without significant losses in the duplicate detection quality. Our evaluation showed that the BAW variants outperform the state-of-the-art methods by diminishing the overall number of comparisons and still maintaining the high values regarding duplicate detection rate.

In addition, we also proposed *S-BAW* (Spark-based BAW), a novel Spark-based approach for solving the problem of the adaptive SNM parallelization. The solution provides an effi-

cient parallelization of the BAW method and its variants. The approach also addresses the data skew problem with an automatic mechanism of data partitioning that enables a satisfactory load balancing across all available nodes. Our evaluation on a cluster environment using real-world data sources demonstrated that *S-BAW* scales with the number of nodes available. We compared our approach against two existing state-of-the-art algorithms (*i.e.*, *S-SNM* and *MR-DCS++*) and verified that *S-BAW* overcomes *S-SNM* and *MR-DCS++* in terms of performance (execution time).

Finally, to solve the problem of identifying bus trajectories referenced by multiple shapes, we proposed *BULMA* and *BULMA-RT*, two novel map-matching unsupervised techniques for solving the problem of identifying bus trajectories referenced by multiple shapes in batch and real-time modes, respectively. *BULMA* is capable of identifying bus trajectories from sequences of noisy geospatial-temporal data sources by using the route identifier, the bus code and the starting and ending points (of the bus trajectory) together with a dynamic radius threshold as blocking keys to minimize the workload caused by too many comparisons. The technique also considers the context aspects of the trajectories performed by a bus, such as the analysis of all trips performed along the day by that bus, to make better choices about the correct shape followed by the bus.

In turn, *BULMA-RT* is also capable of matching bus trajectories with the correct shape in real-time when there exists multiple shapes for the same route. Since the real-time streaming input contains only a few GPS geo-spatial points and *BULMA* must identify the correct shape considering these small pieces of trajectory information, *BULMA-RT* is designed to adapt its search space when erroneous choices are made in the shape matching step. We also propose efficient Spark-based map-matching approaches for both techniques. Our evaluation using real-world data showed that *BULMA* and *BULMA-RT* achieve high levels of map-matching effectiveness. We compared our techniques against an adapted technique based on the Bag-of-Roads strategy, *BoR-tech* (adapted from [76]), and verified that although *BULMA* and *BULMA-RT* perform slower than *BoR-tech* (in terms of execution time), *BULMA* and *BULMA-RT* overcome *BoR-tech* in map-matching effectiveness terms.

7.2 Future Work

In this section, several thematic themes of extension perspectives and future work are presented in the context of the work developed.

Improvement and applicability of the *MultiMR-DCS++* approach. Regarding the adaptive SNM parallelization solution, *i.e.*, *MultiMR-DCS++*, an interesting future work is to investigate how to improve even more the load balancing of this solution. It is known that the usage of some strategies, such as entity replications and fixing the number of entities per input partition, is generally indispensable to promote the uniformity of the workload distribution. However, it is important to investigate if *MultiMR-DCS++* can still evenly distribute the workload across reduce tasks and nodes without the usage of entity replications. In addition, it is important to investigate other blocking and matching techniques for cluster-based Entity Matching and compare them with *MultiMR-DCS++*. Another opportunity is to verify how *MultiMR-DCS++* can be adapted to address other MR-based adaptive techniques for different kinds of data-intensive tasks.

Improvement and applicability of the BAW and its variants methods. As future work of the new generalized algorithm (BAW) and its two variants (OBAW and ROBAW), it is important to investigate how to improve even more the efficiency of BAW variants with better strategies of overlapping blocks and retrenchment of window sizes. Another research direction is to verify how *S-BAW* can be adapted to address other Spark-based adaptive techniques for different kinds of data-intensive tasks.

GPU-based processing. In addition to parallelization with Spark, the usage of Graphical Processing Units (GPUs) on each cluster node can be used to accelerate the computationally intensive similarity functions. Freely available frameworks, such as OpenCL[14] or CUDA[15], allow the use of massively parallel graphics processors for computations in various application areas. Cloud service providers, such as Amazon EC2, have responded to growing demand by offering virtual machines with exclusive access to powerful GPUs. In order to

[14]OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs).

[15]CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing.

maximize the utilization of each node's resources, it is useful to combine Spark-based parallelization of EM jobs with the acceleration of similarity functions by GPUs. Spark would play the role of a coordination and persistence layer. According to the authors of [33], the use of GPUs is promising, especially in the area of privacy-preserving EM, since the uniform representation of data sources (by bit vectors of the same size) can be easily mapped to the data structures supported by GPUs and the similarity computations over simple bit operations can be traced back. Furthermore, it is also important to compare the MapReduce and Spark-based approaches proposed in this thesis against other parallel technologies, such as SNM on GPUs. Another important remark regarding the GPU processing is that the development of the approaches is done by a graphical API and it demands an intense learning curve to use it properly.

Streaming-based parallel incremental EM approaches. The scenario of data-intensive batch processing (over stationary data) can be prohibitively expensive for big data sources that receive (streaming) data updates frequently (dynamic data). For example, the Web evolves at an enormous rate with new Web pages, content and links added daily [32]. This means that it is impractical to process the whole EM process every time a new increment arrives. Thus, several approaches to perform a fast update of the EM result, as soon as a new increment arrives, have been proposed. These approaches are known in the literature as incremental EM or incremental Linkage [37]. Their purpose is to reduce considerably the EM execution time spent by an EM batch processing and still achieve similar EM results to those ones reached when an EM batch processing is applied.

However, even using incremental EM approaches, the EM remains a data-intensive task when applied over large volumes of data [37, 50]. In the particular case of incremental EM, if the arriving increments have a large size, the task processing can still be computationally costly. Thus, in order to mitigate these difficulties, distributed computing can be considered to enable a fast execution of data-intensive tasks over Big Data sources [50]. In this context, studies involving the usage of the programming models MapReduce (MR) and Spark for the parallelization of data-intensive incremental EM tasks must be conducted.

Conclusion and extension of the toolkit for parallel entity matching. Since the toolkit for parallel entity matching described in the appendix B is a work in progress, as future work, it makes sense to implement and integrate other types of parallel EM approaches. Parallel ap-

proaches for incremental EM, machine (and active) learning-based EM, privacy-preserving EM, among others, would enrich the baseline of approaches available in the toolkit. Furthermore, the full development of the unfinished toolkit components, such as the post-processing module, must be conducted.

Development of new strategies to improve the *BULMA* and *BULMA-RT* map-matching effectiveness. As discussed in Section 6.6, despite the gain in effectiveness of *BULMA* (compared to *BoR-tech*) to perform the map-matching over circular shapes (*BULMA* achieved a F-measure of 0.87 against 0.26 of *BoR-tech*), the F-measure result of *BULMA* (0.87) indicates that *BULMA* must be more robust in the presence of high amounts of noisy and missing data. Thus, an interesting research direction is to investigate new strategies to improve even more its map-matching effectiveness.

A framework development for monitoring the bus fleet based on open data sources. The purpose of this research direction is to get benefit from the high quality and integrated data generated by the execution of *BULMA* and *BULMA-RT*. Due to the generation of an enriched data source, the monitoring and analysis of the bus fleet trajectories can be performed. Thus, several research opportunities (to build the framework functionalities) can be developed aiming to improve urban mobility. Such research opportunities can be listed as follows: i) identification of bus routes that present constant delays and the possible problems for these delays; ii) automatic detection of route deviations and analysis of possible causes for such deviations; iii) automatic recommendation of shapes adjusts according to the frequency of deviation of the buses in a certain route, without the need for a human operator to make the adjustment; among others.

Bibliography

[1] Essam Algizawy, Tetsuji Ogawa, and Ahmed El-Mahdy. Real-time large-scale map matching using mobile phone data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):52, 2017.

[2] Yasser Altowim, Dmitri V. Kalashnikov, and Sharad Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, July 2014.

[3] Hotham Altwaijry. *Analysis-aware approach to entity resolution*. University of California, Irvine, 2015.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD '03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, 2003.

[6] Omar Benjelloun, Hector Garcia-molina, Heng Gong, Hideki Kawai, Tait E. Larson, David Menestrina, and Sutthipong Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *Distributed Computing Systems, International Conference on, 0:37*, 2007.

[7] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[8] James Biagioni, Tomas Gerlich, Timothy Merrifield, and Jakob Eriksson. Easytracker: Automatic transit tracking, mapping, and arrival time prediction using smartphones. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 68–81, New York, NY, USA, 2011. ACM.

[9] Tarciso Braz de Oliveira Filho, Matheus Maciel, Demetrio Gomes Mestre, Nazareno Andrade, Carlos Eduardo Pires, Andreza Raquel Queiroz, and Veruska Santos. Estimating inefficiency in bus trip choices from a user perspective with schedule, positioning and ticketing data. *IEEE Transactions on Intelligent Transportation Systems*, 1:1 – 10, 2018.

[10] Douglas Burdick, Lucian Popa, and Rajasekar Krishnamurthy. Towards high-precision and reusable entity resolution algorithms over sparse financial datasets. In *Proceedings of the Second International Workshop on Data Science for Macro-Modeling*, page 18. ACM, 2016.

[11] Pedro V Camargo and Sarah V Hernandez. Vehicle route reconstruction from gps data: a map-matching algorithm harnessing open source software. In *ARRB Conference, 27th, 2016, Melbourne, Victoria, Australia*, 2016.

[12] Surajit Chaudhuri, Venkatesh Ganti, and Dong Xin. Mining document collections to facilitate accurate approximate entity matching. *Proc. VLDB Endow.*, 2(1):395–406, August 2009.

[13] Zhaoqi Chen, Dmitri V. Kalashnikov, and Sharad Mehrotra. Exploiting context analysis for combining multiple entity resolution systems. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 207–218, New York, NY, USA, 2009. ACM.

[14] Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012.

[15] Peter Christen. A survey of indexing techniques for scalable record linkage and dedu-

plication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, September 2012.

[16] Peter Christen, Ross Gayler, and David Hawking. Similarity-aware indexing for real-time entity resolution. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1565–1568, New York, NY, USA, 2009. ACM.

[17] Peter Christen and Karl Goiser. Quality and complexity measures for data linkage and deduplication. In *Quality Measures in Data Mining*, pages 127–151. Springer, 2007.

[18] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. Entity resolution in the web of data. *Synthesis Lectures on the Semantic Web*, 5(3):1–122, 2015.

[19] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proc. VLDB Endow.*, 9(11):864–875, July 2016.

[20] William Cohen, Pradeep Ravikumar, and Stephen Fienberg. A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, volume 3, pages 73–78, 2003.

[21] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, August 2003.

[22] Georgina Cosma and Mike Joy. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Trans. Comput.*, 61(3):379–394, March 2012.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[24] D. DeWitt and M. StoneBraker. Mapreduce: A major step backwards, 2008, http://www.cs.washington.edu/homes/billhowe/mapreduce_a_major_step_backwards.html, June 2013.

[25] Wen Dong and Alex Pentland. A network analysis of road traffic with vehicle tracking data. In *AAAI Spring Symposium: Human Behavior Modeling*, pages 7–12, 2009.

[26] U. Draisbach and F. Naumann. A generalization of blocking and windowing algorithms for duplicate detection. In *Proceedings of the 2011 International Conference on Data and Knowledge Engineering (ICDKE)*, pages 18–24, Sept 2011.

[27] Uwe Draisbach and Felix Naumann. Dude: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, volume 100000, page 10000000, 2010.

[28] Uwe Draisbach, Felix Naumann, Sascha Szott, and Oliver Wonneberg. Adaptive windows for duplicate detection. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 1073–1083, Washington, DC, USA, 2012. IEEE Computer Society.

[29] Robert C Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[30] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. Simplifying entity resolution on web data with schema-agnostic, non-iterative matching. In *34th IEEE International Conference on Data Engineering (ICDE)*, 2018.

[31] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, January 2007.

[32] Leonidas Fegaras. Incremental query processing on big data streams. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2998–3012, 2016.

[33] Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. Duplicate detection on gpus. *HPI Future SOC Lab: proceedings 2011*, 70:59, 2013.

[34] Lise Getoor and Ashwin Machanavajjhala. Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.

[35] Phan H Giang. A machine learning approach to create blocking criteria for record linkage. *Health care management science*, 18(1):93–105, 2015.

[36] Amir Globerson, Nevena Lazic, Soumen Chakrabarti, Amarnag Subramanya, Michael Ringaard, and Fernando Pereira. Collective entity resolution with multi-focal attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 621–631, 2016.

[37] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. Incremental record linkage. *Proceedings of the VLDB Endowment*, 7(9):697–708, 2014.

[38] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 127–138, New York, NY, USA, 1995. ACM.

[39] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9–37, January 1998.

[40] Sue-Chen Hsueh, Ming-Yen Lin, and Yi-Chun Chiu. A load-balanced mapreduce algorithm for blocking-based entity-resolution with multiple keys. *Parallel and Distributed Computing 2014*, page 3, 2014.

[41] Gang Hu, Jie Shao, Fenglin Liu, Yuan Wang, and Heng Tao Shen. If-matching: Towards accurate map-matching with information fusion. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):114–127, 2017.

[42] Britta Hummel. Map matching for vehicle guidance (draft). *Dynamic and Mobile GIS: Investigating Space and Time*, 2006.

[43] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing. *Journal of Network and Computer Applications*, 88(C):50–71, 2017.

[44] George R Jagadeesh and Thambipillai Srikanthan. Online map-matching of noisy and sparse location data with hidden markov and route choice models. *IEEE Transactions on Intelligent Transportation Systems*, 18(9):2423–2434, 2017.

[45] Cheqing Jin, Jie Chen, and Huiping Liu. Mapreduce-based entity matching with multiple blocking functions. *Frontiers of Computer Science*, 11(5):895–911, 2017.

[46] Hung-sik Kim and Dongwon Lee. Parallel linkage. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 283–292, New York, NY, USA, 2007. ACM.

[47] Toralf Kirsten, Lars Kolb, Michael Hartung, Anika Gross, Hanna Kopcke, and Erhard Rahm. Data Partitioning for Parallel Entity Matching. In *8th International Workshop on Quality in Databases*, 2010.

[48] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[49] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. In *Proceedings of the 28th International Conference on Data Engineering*, ICDE '12, pages 618–629, Washington, DC, USA, 2012. IEEE Computer Society.

[50] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Comput. Sci.*, 27(1):45–63, February 2012.

[51] Hannes Koller, Peter Widhalm, Melitta Dragaschnig, and Anita Graser. Fast hidden markov model map-matching for sparse and noisy trajectories. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 2557–2561. IEEE, 2015.

[52] Hanna Kopcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, February 2010.

[53] Hanna Kopcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.*, 3(1-2):484–493, September 2010.

[54] Veenita Kunwar, Neha Agarwal, Ajay Rana, and JP Pandey. Load balancing in cloud—a systematic review. In *Big Data Analytics*, pages 583–593. Springer, 2018.

[55] Sara Landset, Taghi M Khoshgoftaar, Aaron N Richter, and Tawfiq Hasanin. A survey of open source tools for machine learning with big data in the hadoop ecosystem. *Journal of Big Data*, 2(1):1, 2015.

[56] Yang W. Lee, Leo L. Pipino, James D. Funk, and Richard Y. Wang. *Journey to Data Quality*. The MIT Press, 2006.

[57] Jimmy Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[58] Kun Ma, Fusen Dong, and Bo Yang. Large-scale schema-free data deduplication approach with adaptive sliding window using mapreduce. *The Computer Journal*, page bxv052, 2015.

[59] David Menestrina, Steven Euijong Whang, and Hector Garcia-Molina. Evaluating entity resolution results. *Proceedings of the VLDB Endowment*, 3(1-2):208–219, 2010.

[60] Demetrio Gomes Mestre and Carlos Eduardo Pires. Efficient entity matching over multiple data sources with mapreduce. In *Proceedings of the 28th Brazilian Symposium on Databases*, SBBD'13, pages 19–24. SBC, 2013.

[61] Demetrio Gomes Mestre and Carlos Eduardo Pires. Improving load balancing for mapreduce-based entity matching. In *Proceedings of the Eighteenth IEEE Symposium on Computers and Communications*, ISCC '13. IEEE Computer Society, 2013.

[62] Demetrio Gomes Mestre, Carlos Eduardo Pires, and Dimas C. Nascimento. Adaptive sorted neighborhood blocking for entity matching with mapreduce. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 981–987, New York, NY, USA, 2015. ACM.

[63] Demetrio Gomes Mestre, Carlos Eduardo Santos Pires, and Dimas Cassimiro Nascimento. Towards the efficient parallelization of multi-pass adaptive blocking for entity matching. *Journal of Parallel and Distributed Computing*, 101:27 – 40, 2017.

[64] Demetrio Gomes Mestre, Carlos Eduardo Santos Pires, Dimas Cassimiro Nascimento, Andreza Raquel Monteiro de Queiroz, Veruska Borges Santos, and Tiago Brasileiro Araujo. An efficient spark-based adaptive windowing for entity matching. *Journal of Systems and Software*, 128:1 – 10, 2017.

[65] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, April 2012.

[66] Dimas C. Nascimento, Carlos Eduardo Pires, and Demetrio Gomes Mestre. A data quality-aware cloud service based on metaheuristic and machine learning provisioning algorithms. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1696–1703, New York, NY, USA, 2015. ACM.

[67] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[68] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[69] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.

[70] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. Comparative analysis of approximate blocking techniques for entity resolution. *Proceedings of the VLDB Endowment*, 9(9):684–695, 2016.

[71] Jong-Sun Pyo, Dong-Ho Shin, and Tae-Kyung Sung. Development of a map matching method using the multiple hypothesis technique. In *Intelligent Transportation Systems, 2001. Proceedings. 2001 IEEE*, pages 23–27. IEEE, 2001.

[72] Mohammed Quddus and Simon Washington. Shortest path and vehicle trajectory aided map-matching for low frequency gps data. *Transportation Research Part C: Emerging Technologies*, 55:328–339, 2015.

[73] Mohammed A. Quddus, Washington Y. Ochieng, and Robert B. Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15(5):312 – 328, 2007.

[74] Hossein Rahmani, Bijan Ranjbar-Sahraei, Gerhard Weiss, and Karl Tuyls. Entity resolution in disjoint graphs: an application on genealogical data. *Intelligent Data Analysis*, 20(2):455–475, 2016.

[75] Banda Ramadan, Peter Christen, Huizhi Liang, and Ross W. Gayler. Dynamic sorted neighborhood indexing for real-time entity resolution. *J. Data and Information Quality*, 6(4):15:1–15:29, October 2015.

[76] Rudy Raymond and Takashi Imamichi. Bus trajectory identification by map-matching. In *Pattern Recognition (ICPR), 2016 23rd International Conference on*, pages 1618–1623. IEEE, 2016.

[77] Chuitian Rong, Wei Lu, Xiaoyong Du, and Xiao Zhang. Efficient and exact duplicate detection on cloud. *Concurrency and Computation: Practice and Experience*, 25(15):2187–2206, 2013.

[78] Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills. *Advanced analytics with spark: patterns for learning from data at scale*. " O'Reilly Media, Inc.", 2017.

[79] Alieh Saeedi, Eric Peukert, and Erhard Rahm. Comparative evaluation of distributed clustering schemes for multi-source entity resolution. In *Advances in Databases and Information Systems*, pages 278–293. Springer, 2017.

[80] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 110–121, New York, NY, USA, 1989. ACM.

[81] James Shanahan and Liang Dai. Large scale distributed data science from scratch using apache spark 2.0. In *Proceedings of the 26th International Conference on World*

Wide Web Companion, pages 955–957. International World Wide Web Conferences Steering Committee, 2017.

[82] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Ozcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, September 2015.

[83] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. Blast: A loosely schema-aware meta-blocking approach for entity resolution. *Proc. VLDB Endow.*, 9(12):1173–1184, August 2016.

[84] Veda C Storey and Il-Yeol Song. Big data technologies and management: What conceptual modeling can do. *Data & Knowledge Engineering*, 108:50–67, 2017.

[85] Piotr Szwed and Kamil Pekala. An incremental map-matching algorithm based on hidden markov model. In *International Conference on Artificial Intelligence and Soft Computing*, pages 579–590. Springer, 2014.

[86] Guang Tan, Mingming Lu, Fangsheng Jiang, Kongyang Chen, Xiaoxia Huang, and Jie Wu. Bumping: A bump-aided inertial navigation method for indoor vehicles using smartphones. *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1670–1680, 2014.

[87] Yufei Tao. Massively parallel entity matching with linear classification in low dimensional space. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 98. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[88] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 85–98. ACM, 2009.

[89] Nagendra R Velaga, Mohammed A Quddus, and Abigail L Bristow. Developing an enhanced weight-based topological map-matching algorithm for intelligent transport systems. *Transportation Research Part C: Emerging Technologies*, 17(6):672–683, 2009.

[90] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[91] Tobias Vogel, Arvid Heise, Uwe Draisbach, Dustin Lange, and Felix Naumann. Reach for gold: An annealing standard to evaluate duplicate detection results. *Journal of Data and Information Quality (JDIQ)*, 5(1-2):5, 2014.

[92] Chaokun Wang, Jianmin Wang, Xuemin Lin, Wei Wang, Haixun Wang, Hongsong Li, Wanpeng Tian, Jun Xu, and Rui Li. Mapdupreducer: detecting near duplicates over massive datasets. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD'10, pages 1119–1122, New York, NY, USA, 2010. ACM.

[93] Guiping Wang, Shuyu Chen, Mingwei Lin, and Xiaowei Liu. Sbbs: A sliding blocking algorithm with backtracking sub-blocks for duplicate data detection. *Expert Systems with Applications*, 41(5):2415–2423, April 2014.

[94] Qing Wang, Mingyuan Cui, and Huizhi Liang. Semantic-aware blocking for entity resolution. *IEEE Trans. on Knowl. and Data Eng.*, 28(1):166–180, January 2016.

[95] Qing Wang, Jingyi Gao, and Peter Christen. A clustering-based framework for incrementally repairing entity resolution. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 283–295. Springer, 2016.

[96] Steven Euijong Whang and Hector Garcia-Molina. Developments in generic entity resolution. *IEEE Data Engineering Bulletin*, 2011.

[97] Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1111–1124, 2013.

[98] Niklas Wilcke. Ddup-towards a deduplication framework utilising apache spark. In *BTW Workshops*, pages 253–262, 2015.

[99] D. Wu, T. Zhu, W. Lv, and X. Gao. A heuristic map-matching algorithm by using vector-based recognition. In *Computing in the Global Information Technology, 2007. ICCGI 2007. International Multi-Conference on*, pages 18–18, March 2007.

[100] Su Yan, Dongwon Lee, Min-Yen Kan, and Lee C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '07, pages 185–194, New York, NY, USA, 2007. ACM.

[101] Wei Yan, Yuan Xue, and B. Malin. Scalable and robust key group size estimation for reducer load balancing in mapreduce. In *Big Data, 2013 IEEE International Conference on*, pages 156–162, Oct 2013.

[102] Can Yang and Gyoso Gidofalvi. Fast map matching, an algorithm integrating hidden markov model with precomputation. *International Journal of Geographical Information Science*, 32(3):547–570, 2018.

[103] Jae-seok Yang, Seung-pil Kang, and Kyung-soo Chon. The map matching algorithm of gps data with relatively long polling time intervals. *Journal of the Eastern Asia Society for Transportation Studies*, 6:2561–2573, 2005.

[104] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[105] Dongzhan Zhang, Chengfa Liao, Wenjing Yan, Ran Tao, and Wei Zheng. Data deduplication based on hadoop. In *Advanced Cloud and Big Data (CBD), 2017 Fifth International Conference on*, pages 147–152. IEEE, 2017.

[106] Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29, 2015.

[107] Linhong Zhu, Majid Ghasemi-Gol, Pedro Szekely, Aram Galstyan, and Craig A

Knoblock. Unsupervised entity resolution on multi-type graphs. In *International Semantic Web Conference*, pages 649–667. Springer, 2016.

Appendix A

Input Data Sources for the Map-matching Bus Trajectories Approaches

This appendix describes the schema of the GTFS and AVL data sources utilized as input to the Map-matching bus trajectories approaches presented in this thesis.

A.1 GTFS shape files

The data are provided also in text format and records are delimited by newline characters. Table A.1 describes the fields available in the file and provides an example for each value.

A.2 Bus Position Data (Vehicle Location)

The data are provided in text format and records are delimited by newline characters. Table A.2 describes the fields available in the file and provides an example for each value.

Table A.1: GTFS shape files

FIELD NAME	***DESCRIPTION***	***EXAMPLE OF VALUE***
ROUTE_ID	The route identifier	519
SHAPE_ID	The predefined trajectory identifier	1708
SHAPE_PT_LAT	The latitude of the geo-spatial point	-25.4139
SHAPE_PT_LON	The longitude of the geo-spatial point	-49.2797
SHAPE_PT_SEQUENCE	The geo-spatial point sequence of the predefined trajectory	5859
SHAPE_DIST_TRAVELED	The traveled distance (in meters) from the beginning of the shape until the current geo-spatial point	57.462

Table A.2: Bus Position Data

FIELD NAME	***DESCRIPTION***	***EXAMPLE OF VALUE***
BUS.CODE	Bus line code	EA183
LATITUDE	The latitude sent by the GPS	-25.515291
LONGITUDE	The longitude sent by the GPS	-49.230788
TIMESTAMP	The time when the GPS sends the data	00:06:29
LINE.CODE	The bus line (usually the route)	519
GPS.ID	The GPS identifier	1

Appendix B

Toolkit for Parallel Entity Matching

In the context of frameworks and tools for parallel batch EM using MR and Spark, there are two works [48, 98]. The first one proposes a tool, known as Dedoop, to solve the EM problem over stationary data sources using Hadoop MR. The second one proposes a strategy to implement a tool for Entity Resolution using a predefined Spark workflow. However, these tools neither deal with incremental EM nor present the combined use of Hadoop and Spark. They also do not allow the comparison of Spark-based approaches with MR-based approaches. In addition, Dedoop has been tightly implemented over a deprecated Hadoop version (0.20), which does not benefit from the new Hadoop features, such as YARN (Yet Another Resource Negotiator), a distributed resource manager. Thus, this chapter describes the proposition of a more complete and robust tool for the EM task large-scale processing.

B.1 Use-Case Overview

The tool provides a Web interface in which the user can specify and configure EM tasks in parallel and submit them to a Hadoop or Spark cluster. The main use cases are described as follows:

- Initially, the files or data sources containing the data that must be deduplicated by the distributed EM algorithms are submitted to the cluster by the user. The EM can be executed over a single or multiple data sources;
- The user can choose any of the approaches proposed in the Chapters 4 and 5 or submit

a particular MapReduce or Spark-based EM approach to be executed by the tool;

- Users can use various types of similarity metrics to perform comparisons between entities, both from libraries already consolidated such as Second-String [20], and libraries which can be submitted to the tool;
- The user can, after setting the configuration of a given EM task, submit the EM task to executed by the Hadoop or Spark and monitor the execution of the submitted task;
- A REST API is available to handle any request from EM task submission through the HTTP protocol (both from the user or any service).

B.2 Proposed Architecture

To provide the above set of functionalities, the tool was architected as shown in Figure B.1. The tool has been developed using Java technology, which provides the development of an object-oriented system, which makes it easier to extend the functionalities. Thus, the parts that compose the architecture are described as follows.

B.2.1 Web Application Tool for Parallel EM

Since it has been projected to be a web-accessible tool, the user interface has been developed using Java Server Faces (JSF), i.e., a JAVA specification for building component-based user interfaces for web applications. Business logic has been developed in Java using the Java Persistence API (JPA), i.e., a standard JAVA language API that describes a common interface for data persistence frameworks. This business logic is used to manage authentication and authorization information to enable the user (client) to access the distributed infrastructure resources, execute workflows of EM tasks and collect other types of information.

Thus, a user (e.g., researchers or data specialists) interested in performing distributed EM tasks will access the application through a Web interface. The service event handling methods, defined in this Web interface, submit requests to the business logic module using the HTTPS communication protocol. In turn, these requests are handled by the business logic module hosted on an Apache Tomcat application server, that is, a Java-based open

source web container that was created to execute Web applications that use Servlets and JSP technologies.

B.2.2 REST API for the Parallel EM Tool

The web application is connected to a REpresentational State Transfer (REST) API, i.e., a web service architecture style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements within a distributed hypermedia system. This service is, in fact, responsible for the requests made to the Hadoop ecosystem. Through the REST API interface, operations to be executed in the Hadoop ecosystem, such as creation, renaming, deletion, opening, reading and writing of files and directories in the HDFS as well as the submission of distributed EM tasks are specified. Note that, in Figure B.1, the REST service is built on top of the Kerberos protocol, i.e., a computer network authentication protocol that allows secure and identified individual communications on an insecure network. The use of Kerberos is a security mechanism to enable information exchange between the web application (or other services access to distributed infrastructure resources) and the Hadoop ecosystem.

B.2.3 Distributed Processing Ecosystem

Regarding the distributed service itself, the Hadoop ecosystem is a "software library" that gives users the opportunity to process large volumes of data on clusters of commodity computers through the use of simple distributed programming models. In other words, it promotes the possibility to collect, store, and analyze large volumes of data in parallel. The main components of the Hadoop project are the Hadoop Commons (a set of common utilities to support the rest of Hadoop modules), Hadoop Distributed File System (HDFS) and Hadoop MapReduce. As illustrated in Figure B.1, both Hadoop YARN and Hadoop HDFS are provided with REST-type interfaces, whose operations is submitted through the communication established with the REST API service of the EM tool.

As additional parts of the Hadoop ecosystem, Apache Spark and Apache HBase (a key-value NoSQL system that runs on the basis of HDFS) are used to promote the rest of the functionality available through the EM tool. Apache Spark, as mentioned earlier, is, such

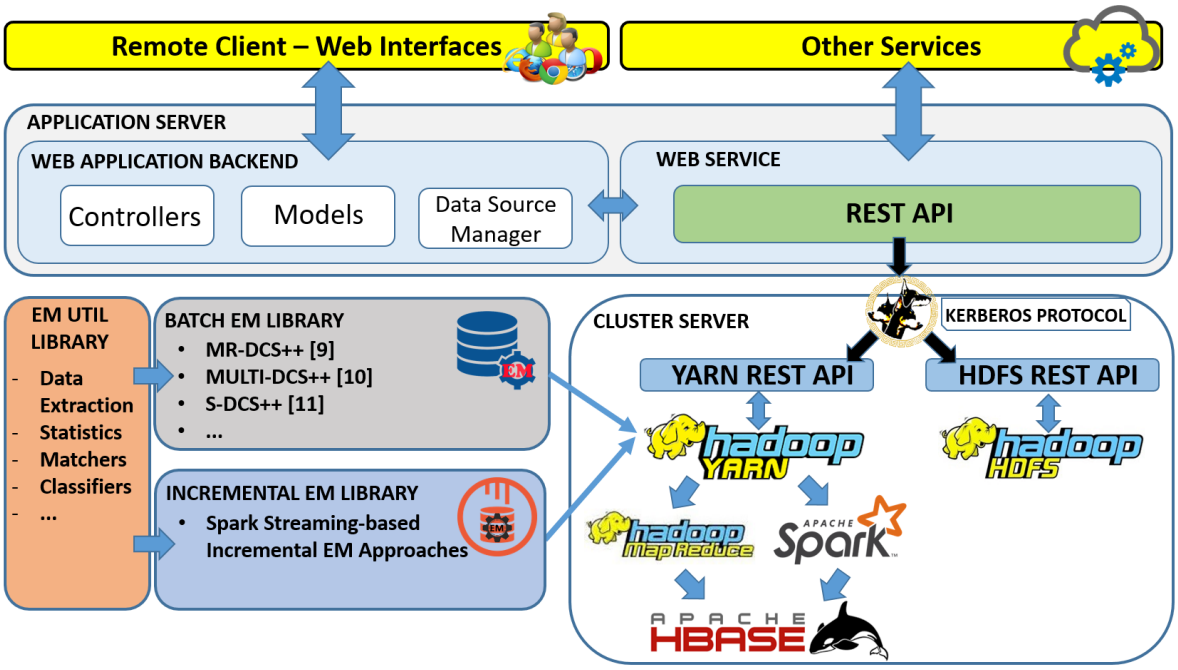

Figure B.1: Parallel EM toolkit architecture.

as the Hadoop MapReduce, a distributed computing device that promotes large-scale data processing and has been used to execute many of the approaches proposed in Chapter 5. Apache HBase is used as a distributed DBMS to support the execution of Hadoop MR and Spark tasks.

B.2.4 EM Service

All modules described in the Hadoop ecosystem and their additional parts are installed as their latest versions specify, without modification. However, the two EM services (EM for static data sources or batch and incremental and real-time EM for dynamic data sources) that is provided by the tool, is installed in the ecosystem Hadoop as code source that runs on the top of MR and Spark.

The EM service for static databases is composed of a set of algorithms, whose baseline was described in Chapters 4 and 5 of this thesis, that enables the execution of EM following different configurations. In other words, the service generates requests of MR or Spark-based EM job execution workflows according to the parameters specified by the user. These parameters can be related to the definition (by the client) of the indexing approach that will be applied, the classifier (similarity function) to be used in the entity comparisons, threshold

values needed to execute the distributed approaches, among others.

B.2.5 Structures and Functions

In the following, we describe the main features (components and functionalities) provided by the parallel EM toolkit[5]. Figure B.1 depicts the system architecture to support these features.

Data extraction This component allows several data sources to be read in parallel. Currently, it is capable of extracting records/objects from relational databases (i.e., Oracle, DB2, MySQL and PostgreSQL), CSV/XML text files and NoSQL databases (i.e., HBase). During the extraction, the component maps the records/objects into entities represented in a JSON format. For each data extractor (related to each type of data source), an entity identifier, consisting in one or more attributes (specified by a data specialist), can be defined and linked to a global identifier that is generated for each data extractor. This global identifier is stored together with the extracted records/objects. As a result, the comparison between entities from different data sources can be done without any mixing of identifiers.

Preprocessing This component is utilized by the data specialist to collect statistical information about the data sources during the data extraction phase, e.g., the number of (distinct) records or values. Once the statistical information is extracted, the parallel approaches can read it for some specific purpose. For example, the number of entities per index is important for load balancing strategies, since the approaches can calculate previously the number of comparisons that must be performed by each reducer/worker.

Parallel batch EM approaches The parallel batch EM approaches developed in this work is available in the toolkit. The toolkit persists the data structures (that enable the increments processing) on a HBase database (see Figure B.1). It is also be possible to submit and run external MR or Spark-based batch and incremental EM approaches to the toolkit.

Matchers and classifiers Various types of matchers and classifiers is available to be used as similarity comparison and decision functions. Matchers are typically used to compute

[5]https://sites.google.com/site/demetriomestre/projects/em_toolkit

the similarity value between entities. Thus, the toolkit provides two types of matchers: attribute value matchers (i.e., comparisons considering corresponding attributes) and context matchers (i.e., comparisons considering context or semantic relationships). For classification (decision) purposes, three kinds of classifiers is provided: machine learning, threshold, and rule-based.

Post Processing This component (not yet developed) will provide a statistical background capable of collecting and processing information about the execution of the approaches. Both the quality metrics as well as the EM performance in terms of execution time will be collected. By this, for instance, it will be easier to compare the performance of competing EM approaches. Among the metrics that can be collected, we can mention the number of entities comparisons generated and the number of entities classified as matches. If a Gold Standard file is provided for the data sources involved, the metrics precision, recall, and F-Measure can be calculated. This component will also collect and store the statistical information about the execution of the parallel for future analysis.

EM Output The output file generated, after the MR and Spark-based approaches execution, consists of a list of entity pairs associated to their similarity value and the classification result. All output is stored at the HDFS/HBase database to facilitate the output access through web interfaces.

Regarding the system architecture defined for the toolkit (Figure B.1), the main workflow works as follows. As we can see, the cluster server receives requests from the Application REST API or any other application/system interested in submitting EM tasks to the cluster environment. For this, the web service establishes a remote connection with the cluster server to enable the execution of the necessary operations such as submitting artifacts to the HDFS or starting the execution of the MR and Spark jobs. Also note that the cluster server is built on Kerberos (i.e., a computer network authentication protocol) to provide a secure communication between the web service and the REST APIs of the Ecosystem.

B.2.6 System Graphic Unit Interfaces

Figures B.2, B.3, B.4, B.5, B.6 show the main system workflow. The workflow sequence can be described as: i) listing and uploading files and libraries to the HDFS (Figures B.2,B.3); ii) submitting an execution of the Spark-based EM program (Figure B.4); iii) running the Spark-based EM program (Figure B.5); and iv) showing the output of the EM task (Figure B.6).

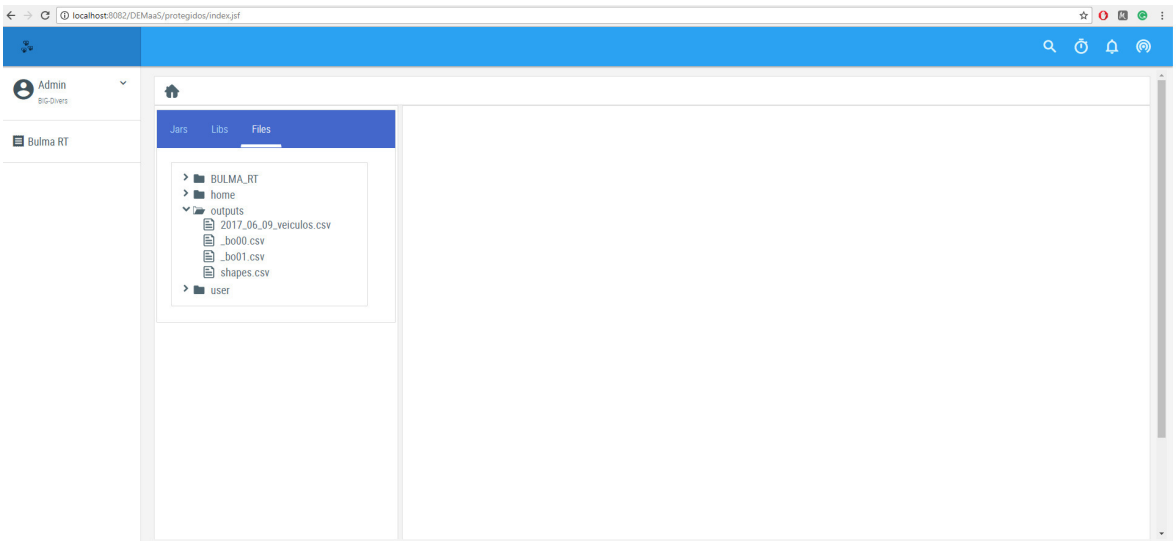

Figure B.2: Listing the files of the HDFS.

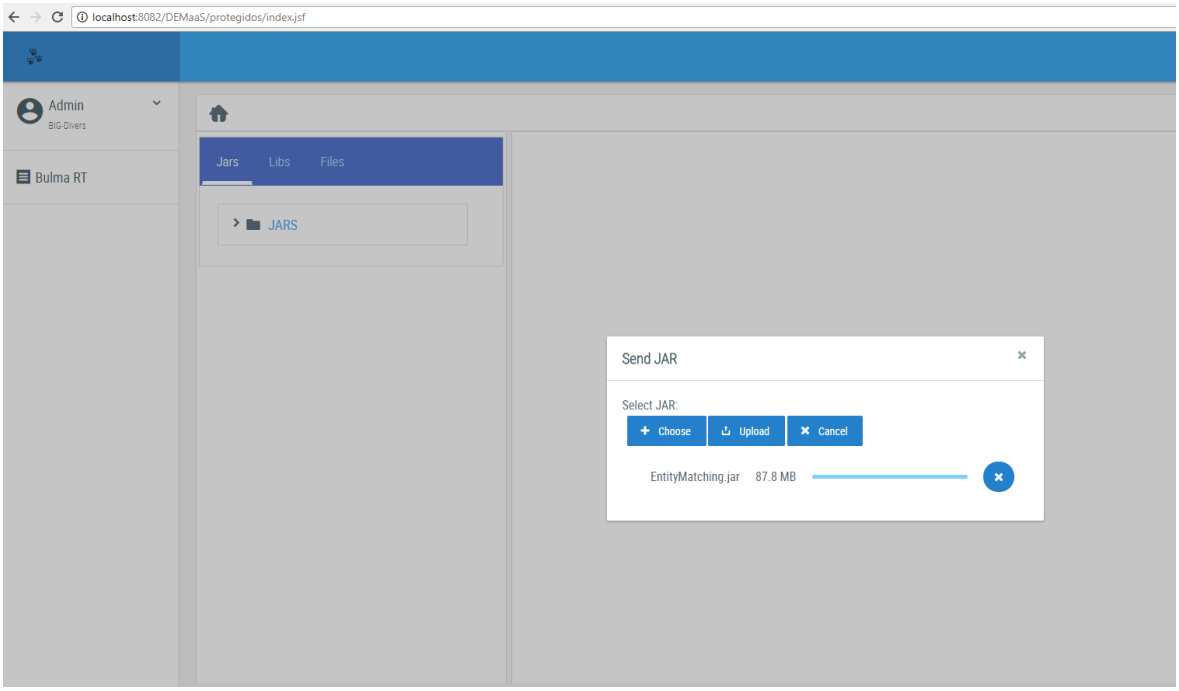

Figure B.3: Uploading a JAR library containing a Spark-based EM program.

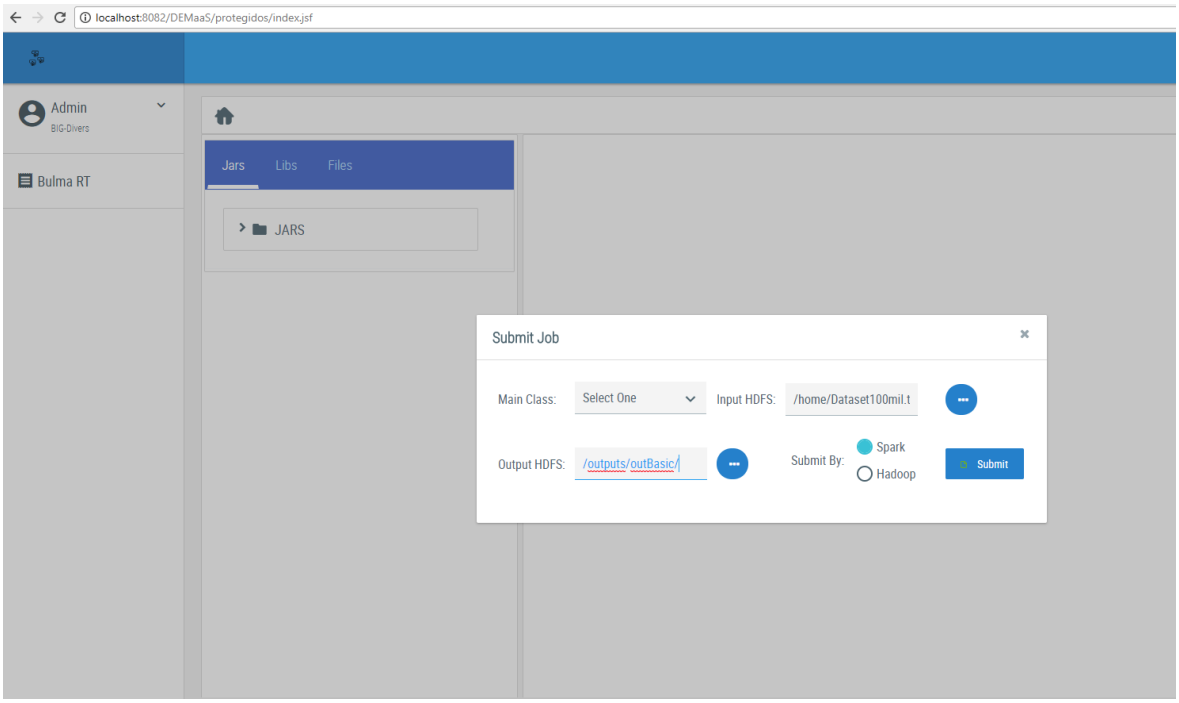

Figure B.4: Submitting the execution of the Spark-based EM program.

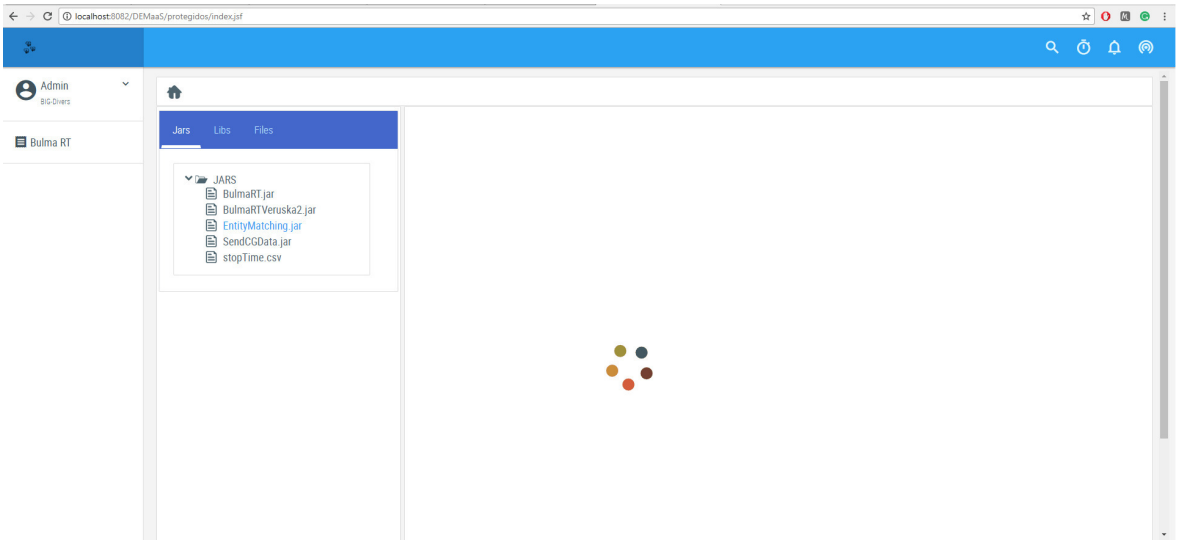

Figure B.5: Running the Spark-based EM program.

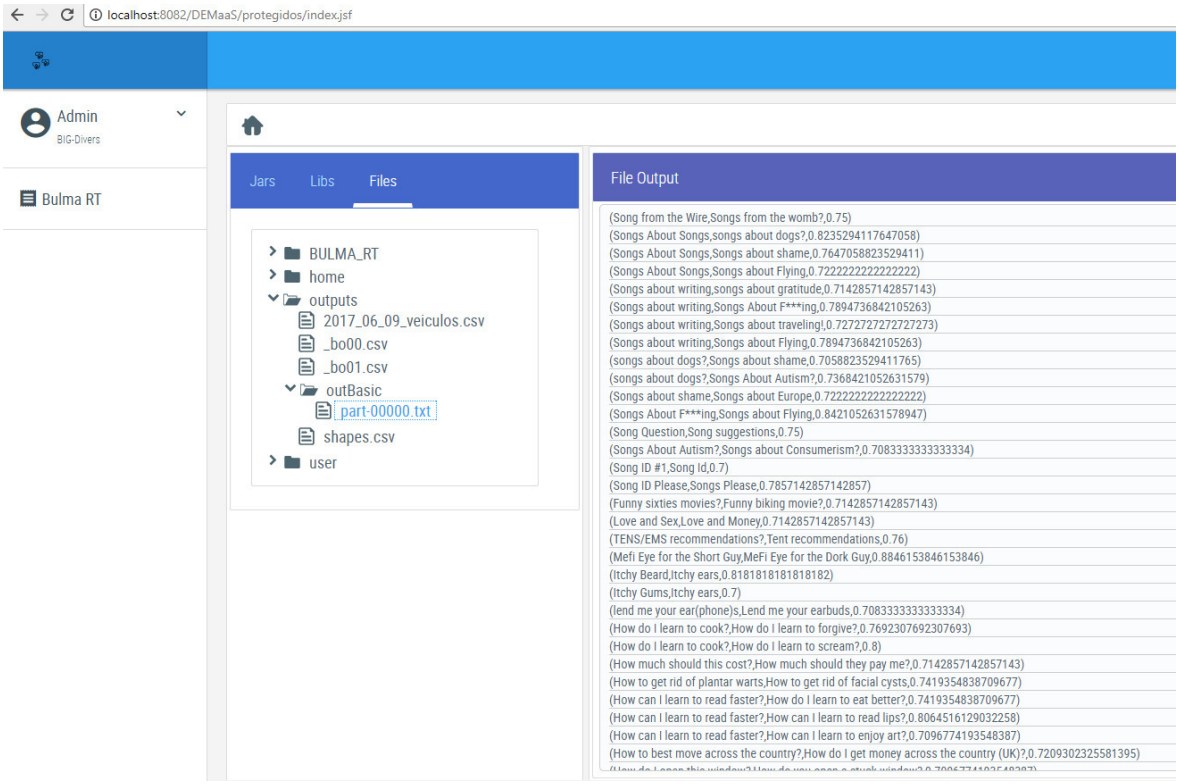

Figure B.6: Showing the output of the Spark-based EM program.