ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : informatique

École doctorale Matisse

présentée par

Javier Olivares

préparée à l'unité de recherche Inria Rennes – Bretagne Atlantique
Institut National de Recherche en Informatique et en Automatique
Université de Rennes 1

Scaling out-of-core K-Nearest Neighbors Computation on Single Machines

Thèse soutenue à Rennes le 19 Décembre 2016

devant le jury composé de :

Pierre SENS
Professeur à l'Université de Paris 6 / *Rapporteur*

Sébastien MONNET
Professeur à l'Université Savoie Mont Blanc / *Rapporteur*

Erwan LE MERRER
Chercheur à Technicolor Rennes / *Examinateur*

François TAÏANI
Professeur à l'Université de Rennes 1 / *Examinateur*

Anne-Marie KERMARREC
Directrice de recherche, INRIA Rennes / *Directrice de Thèse*

A mi esposa Andrea, mi compañera de aventuras, mi apoyo y mi motivación.

This thesis was funded by **Beca Doctorado en el Extranjero** Folio 72140173,
CONICYT, Ministry of Education, Government of Chile
and **Google Focused Award** Web Alter-Ego.

Acknowledgments

I would like to thank my thesis director Anne-Marie Kermarrec for her support, guidance and confidence during my Ph.D. Thank you Anne-Marie for accepting me as a student, for advising my research, and for encouraging me to improve my skills and help me to strengthen my weaknesses.

I would also like to thank the members of the jury for having accepted to examine this thesis. Thank you both Professors Pierre Sens and Sébastien Monnet for being referees of my thesis, and Dr. Erwan Le Merrer and Prof. François Taïani for accepting to participate in the jury.

Thank you very much Pierre-Louis Roman and Simon Bouget for your useful help in revising the text in french of this work. Thanks to all members of ASAP team for these three years, you were an important part of my stay in France.

Gracias a Dios por guiar mis pasos hasta aquí. Gracias a mi esposa Andrea por su apoyo, por su compañía, por su fortaleza y por su amor. Gracias por acompañarme en esta aventura. Todo esto ha sido posible en gran parte gracias a su ayuda.

Gracias a mi familia por su apoyo a pesar de la distancia. Gracias a cada uno de nuestros amigos en Francia y en Chile por los momentos que hemos vivido juntos.

Contents

Acknowledgements **5**

I Introduction and State-of-the-Art

1 Introduction **17**
1.1 Contributions 20
1.1.1 *Pons* 21
1.1.2 *UpKNN* 21
1.2 Publications 23
1.3 Organization of the manuscript 23

2 Background and state-of-the-art **25**
2.1 Definition and notation 25
2.2 Applications 26
2.3 General approaches to compute KNN 28
2.3.1 Brute-force versus approximate approaches 29
2.3.2 Approximate approach: KNN as a graph 31
2.4 Distributed computation 35
2.4.1 Cluster-based approach 37
2.4.2 MapReduce approaches 38
2.4.3 P2P approaches 38
2.4.4 Challenges in KNN distributed computation 40
2.5 Out-of-core computation 42
2.5.1 Motivation 42
2.5.2 Scaling out-of-core algorithms 43
2.6 Updating profiles on KNN algorithms 46
2.6.1 Motivation and challenges 46

2.6.2 Current situation 48
2.7 Summary 49

II Out-of-core KNN Computation

3 Towards a scalable out-of-core KNN computation 53
3.1 Introduction 54
3.2 System design 54
3.2.1 Overview 55
3.2.2 KNN iteration 55
3.2.3 Heuristics for processing the Partition Interaction Graph . 58
3.3 Concluding remarks 58

4 Out-of-core KNN approach 61
4.1 Introduction 61
4.2 Preliminaries 63
4.2.1 In-memory approach 64
4.2.2 Disk-based approach 65
4.3 *Pons* 66
4.3.1 Overview 67
4.4 KNN iteration 70
4.4.1 Phase 1: Partitioning 72
4.4.2 Phase 2: In-edge partition files 73
4.4.3 Phase 3: Out-edge partition files 74
4.4.4 Phase 4: Profile partition files 75
4.4.5 Phase 5: Distance computation 75
4.5 Experimental setup 76
4.5.1 Machine 76
4.5.2 Datasets 76
4.5.3 Evaluation metrics 77
4.6 Evaluation 78
4.6.1 Performance 78
4.6.2 Multithreading performance 79
4.6.3 Performance on different memory availability 80
4.6.4 Evaluating the number of partitions 81
4.7 Conclusions 84

III Updating User Profiles

5 Updating profiles in KNN algorithms 87
5.1 Introduction 88
5.2 Background 90
5.2.1 Notations and assumptions 90
5.2.2 Problem definition 91
5.3 *UpKNN* algorithm 91
5.3.1 Classify-Merge phases 92
5.3.2 Implementing *UpKNN* on *Pons* 98
5.4 Experimental setup 99
5.4.1 Machine 99
5.4.2 Datasets 99
5.5 Evaluation 100
5.5.1 Performance 101
5.5.2 Evaluating usage of data in memory 107
5.5.3 Evaluating design decisions 108
5.6 Conclusions 111

IV Final Remarks

6 Conclusions 115
6.1 Thesis summary 115
6.2 Objectives of this thesis 116
6.3 Contributions 117
6.3.1 *Pons* [24] 117
6.3.2 *UpKNN* [53] 118
6.4 Perspectives 118

V Appendix

7 Résumé 123

Résumé 123
7.1 Contributions 125
7.1.1 *Pons* 125

7.1.2 *UpKNN* 126
7.2 Perspectives 126
7.2.1 *Pons* 126
7.2.2 *UpKNN* 127
7.3 Future Work 127

Bibliography 129

List of Figures

2.1 Example of HyRec's KNN candidate set 36

3.1 5 phases: input $G(t)$, 1) KNN graph partitioning, 2) Hash Table, 3) Partition Interaction Graph, 4) KNN computation, 5) Updating profiles 55

4.1 Pons executes 5 phases: (1) Partitioning, (2) In-Edge Partition Files, (3) Out-Edge Partition Files, (4) Profile Partition Files, and (5) Distance Computation 69
4.2 Example graph. A's out-neighbors and A's neighbors' neighbors . . . 70
4.3 In-edge partition files and out-edge partition files 71
4.4 A's neighbors' neighbors found using bridge vertices 71
4.5 Partitioning example 73
4.6 Impact of multithreading on *Pons* 80
4.7 Impact of the available memory on *Pons* 81
4.8 Runtime: The impact of the number of partitions M. ANN-SIFT 50M dataset 82
4.9 Runtime: The impact of the number of partitions M. Twitter dataset 82
4.10 Memory footprint: The impact of M. ANN-SIFT 50M dataset 83
4.11 Memory footprint: The impact of M. Twitter dataset 84

5.1 Classify phase. Reader threads in continuous lines, classifier threads in dashed lines 93
5.2 $T_r/T_c/C$ configuration. (T_r in continuous lines, T_c in dashed lines) . . 94
5.3 Classification example 95
5.4 Merge example 97
5.5 Scalability in terms of updates processed 104
5.6 Threads scalability 107
5.7 Heap improvement (*Movielens* Dataset) 109
5.8 Heap improvement (*Mediego* Dataset) 110

List of Tables

3.1 Number of load/unload operations using partition interaction graph 58

4.1 Datasets 78
4.2 Relative performance Pons/INM, and memory footprint 79

5.1 Datasets description in number of users, items and number of updates (80% of the dataset size) 100
5.2 UpKNN's runtime and speedup (with respect to the baseline) in updating entities' profiles 102
5.3 Number of updates processed by second on UpKNN 103
5.4 Number of updates processed in a KNN iteration 104
5.5 Disk operations. % of *UpKNN*'s operations with respect to those of the baseline 106
5.6 Worst case (1 update per user) vs average case (same number of updates but distributed among all users) 108
5.7 Disk operations heap/no heap 110

Part I

Introduction and State-of-the-Art

Chapter 1

Introduction

For several years we have witnessed an overwhelming growth of the data available around us. This could not have been possible without the worldwide expansion of the Internet, and its ease of access for a significant majority of the users, through both personal computer and mobile devices. This vast access and availability have generated a significant increase in the volume of data generated daily. According to IBM's report, about 2.5 quintillion bytes of data are created every day[†1]. To illustrate, every minute are uploaded hundreds of hours of videos on YouTube[†2]; an average of 350,000 tweets per minute on Twitter[†3]; and 300 millions of new photos are posted on Facebook every day[†4].

Though the full access to a wide range of data can be helpful for the users, this huge amount of data becomes useless whether it is not properly classified, filtered, or displayed. In this overwhelming scenario, as users, we need to receive the data in an appropriate manner, after a thorough processing capable of offering the valuable information to fulfill our needs. Particularly, we can collect rewarding information after answering the following questions: *how do I find similar data in this huge set of data?*, *how do I find similar things to those I like in a vast world as Internet?*, or *how can I find similar photos/music/books to those that I have seen/listened/read before?*

K-Nearest Neighbors (KNN) is the basis of numerous approaches able to answer these questions. In this thesis, we focus on KNN algorithms, which have proved to be an efficient technique to find similar data among a large set of it. To do so, in a nutshell, KNN searches through the set, it compares the data, and finally delivers a list of those elements which are similar.

[†1] https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html
[†2] http://expandedramblings.com/index.php/youtube-statistics/
[†3] http://www.internetlivestats.com/twitter-statistics/
[†4] https://zephoria.com/top-15-valuable-facebook-statistics/

Although KNN is not the only method available, it has certainly grown in popularity [117], which is mainly due to its capacity, simplicity and versatility. To illustrate its versatility, we can easily find KNN applications in many diverse fields such as business [27, 73, 115], medicine [2, 58, 61, 122], music [34, 36, 56, 105], urban planning [29, 75], or computer science (specially on recommender systems, image classification, and information retrieval), among others [3, 19, 121]. For instance, KNN can be used in an image classification context to find similar images in a large catalog. In a recommender system scenario, we can use KNN algorithm to trace those users with similar tastes, taking advantage of their similarity to recommend them new items to consume. Alternatively, assuming a business context, KNN can be a good tool to recognize similar patterns in the stock market and thus predict fluctuations in the stock price, for example. Furthermore, an extensive research has been done on multiple applications of KNN algorithms, demonstrating its adaptability to different contexts, and its capacity to achieve excellent results.

However, such an efficient and versatile algorithm, always comes at a cost. Such a cost can be exorbitant, particularly nowadays where the set of data available is continuously growing at unimaginable rates. This growth creates new challenges in the development of more efficient techniques to process data, more energy-efficient hardware, and also more powerful, reliable, and easy to use computing systems.

In a world where data changes continuously, performing efficient KNN computations on large datasets requires significant amounts of memory. Although the basic idea behind KNN does not change considerably, data represented and processed may vary significantly across the different applications. For instance, applying KNN to find similar images, means to handle large catalogs of images in memory, spanning gigabytes or beyond for each image, specially on satellite images, one of the most complex cases.

Besides, along with the large memory footprints generated for handling large sets of data, performing KNN queries turns to be a very time-consuming task by itself. In this regard, while searching similar data among large set of data impacts the time consumption, the cost is dominated by comparing this data [14, 32].

On account of the fact that KNN queries on large datasets are often a resource-greedy computation, many works have proposed a wide range of algorithms to efficiently leverage the resources of single machines or distributed systems. In the one hand, the use of single machines brings new challenges on the use of the often limited memory available in such a setup. It is a well-known fact that the most efficient algorithms leverage fast accesses delivered by modern RAM cards, instead of using virtual memory or disk. Those re-

sources' bandwidth -despite the progress in the development of faster devices- is still slower than that of a RAM. Unfortunately, storing the whole set of KNN data in main memory is not always affordable. Despite the fact that the main memory available in modern computers has grown considerably over the years, so it has the amount of data to process. On the other hand, notwithstanding that distributed systems overcome the limitation of the available memory in single machines, the design, implementation, and deployment of distributed algorithms still remain challenging. This is a consequence of the fact that KNN's data does not necessarily show a good extent of spatial locality. Data accessed during similarity comparisons could be stored anywhere, probably on different machines, consequently data access is costly, affecting considerably algorithms' performance. In other words, this lack of locality increases the communication among machines during the process of comparisons, thereby increasing the global runtime of the algorithm.

As a direct consequence of its high cost, we may observe on some the state-of-the-art works [13, 14, 32], specially those processing KNN queries on very large datasets, that the KNN computation is often performed offline. In other words, offline periods are those when the system is not working or is running under a reduced workload. Generally, KNN algorithms represent just a layer of major systems, hence they very seldom have access to the full set of computational resources at any time. For instance, in a recommender system application, KNN searches similar users to generate a set of recommendations based on users' similarity. KNN computation, in this context, represents only one phase of the system's processing, the remaining time is used on the recommendation tasks, and some system's maintenance works. As the KNN processing is a time/resource-consuming task, it is performed offline (at night generally), allocating available system's resources to prioritize online tasks. Although offline KNN computation decongest system's load to expedite online tasks, offline process runs over static or outdated data, which may be harmful for the quality of the KNN outcome. Computing KNN offline, on data obtained in long-time window periods, makes us lose valuable information arising from the inherent dynamism of data, particularly in contexts like social networks, where the data flow is enormous every second.

The high-cost of computing KNN not only leads to its offline computation, but also to a simplification regarding data dynamism. Current state-of-the-art works on KNN handle only static sets of data throughout the computation. This simplification aims to reduce the computational complexity and runtime of the algorithm. Unfortunately, despite it brings benefits, it also induces a downside. Computing on static data does not reflect appropriately its true dynamic nature, affecting the potential results. In these days, where data flows and changes rapidly, the processing of static data (or data updated daily

or even less frequently), makes us lose some valuable information that arises from the dynamism observed on a much smaller scale, at the level of minutes, seconds or even less.

It is reasonable to think that updating data during the computation adds new complexities to the algorithm, specially if we consider that this data is continuously accessed to perform comparisons. Furthermore, as data changes rapidly, updating data in an online fashion would force the algorithm to process, not only its own KNN task, but also to handle large streams of dynamic data simultaneously. Such a multiple processing brings new challenges in the design and implementation of algorithms capable of computing KNN on data that changes continuously and rapidly over time.

1.1 Contributions

The main goal of this thesis is to propose an efficient solution for scaling the computation of the K-Nearest Neighbors algorithm on single machines. A scalable solution must be capable of processing large current datasets within a reasonable time, considering the limitations imposed by a restricted set of computational resources. The motivation behind the use of single machines instead of more complex distributed systems, is the ease of access to this sort of computational resource and its lower cost with respect to that of distributed systems. In algorithms designed for running on single machines, synchronization, data consistency, and some others well-known issues in distributed systems, do not need to be addressed. Besides, single machines have shown good performance running well-designed algorithms, along with a good extent of simplicity in the design, coding, and deployment of complex algorithms.

In this work, we do not only aim to scale the KNN computation, but we also aim to build a lightweight approach, able to leverage the limited resources of a single commodity machine. Thus, becoming an inexpensive, but not less efficient, approach. A lightweight approach would lend itself as a solution for performing online KNN computation, mainly due to its capacity to run well using fewer resources. It is worth mentioning again that KNN algorithms are usually only a layer of more complex systems, therefore it is not appropriate to assume that all resources will be available for the KNN process. Therefore, an online KNN approach is a valuable solution whether, along with using fewer resources, it runs within reasonable times.

Along with scaling the KNN computation, we also aim to propose an efficient solution for processing updates on data during the KNN computation.

As we mentioned above, the dynamic nature of data has not been properly reflected and adequately handled on static algorithms, as current KNN state-of-the-art approaches do.

In this thesis, we fulfill these goals through two main contributions, which we describe in the following.

1.1.1 *Pons*

Our first contribution is *Pons* [24]: an out-of-core algorithm for computing KNN on large datasets that do not completely fit in memory. To do so, *Pons* leverages efficiently both disk and the available memory. Our approach is able to compute KNN incurring a minimal cost, by storing all data in hard disk, loading and processing this data from disk into a limited section of the main memory. The main rationale of our approach is to minimize random accesses to disk, and to favor, as much as possible, sequential reading of large blocks of data from disk.

The specific contributions of this work are as follow: We propose *Pons*, an out-of-core approach for computing KNN on large datasets, using at most the available memory, and not the total amount required for a fully in-memory approach. *Pons* has been designed to solve efficiently the non-trivial challenge of finding neighbors' neighbors of each entity during the KNN computation.

Our experiments performed on large-scale datasets show that *Pons* computes KNN in only around 7% of the time required by an in-memory computation. *Pons* shows to be also capable of computing online, using only a limited fraction of the system's memory, freeing up resources for other tasks if needed. *Pons*' performance relies on its ability to partition the data into smaller chunks such that the subsequent accesses to these data segments during the computation is highly efficient, while adhering to the limited memory constraint. Our evaluation shows *Pons*' capability for computing KNN on machines with memory constraints, being also a good solution for computing KNN online, devoting few resources to this specific task.

1.1.2 *UpKNN*

Our second contribution is *UpKNN* [53]: a scalable and memory efficient, thread-based approach for processing real-time updates in KNN algorithms. *UpKNN* processes large streams of updates, while it still computes KNN efficiently on large datasets.

By using a thread-based approach to access and partition the updates in real-time, *UpKNN* processes millions of updates online, on a single commodity PC. This is achieved by moving away from traditional random access approaches towards a more efficient partition-based idea. Instead of directing a stream of updates directly towards users, we propose to partition the updates, based on the existing partition-based KNN approach (such as in *Pons* [24]).

The specific contributions on *UpKNN* are as follow: We propose an efficient multithreading approach that addresses the challenge of performing real-time updates on KNN data. To achieve good performance, *UpKNN* greatly reduces the number of disk operations performed during the computation, favoring the reading and writing of large chunks of data from disk. Our carefully designed multithreading approach leverages the use of two-layer in-memory buffers to reduce synchronization between threads and concurrency issues in I/O operations.

We perform an extensive set of experiments to demonstrate *UpKNN*'s gain in performance, with respect to that of the baseline approach. The baseline applies the updates directly from a non-partitioned stream of data. As well as *UpKNN*, the baseline was implemented using the multithreading programming model. The set of experiments were performed on a single commodity machine using a well-known publicly available dataset and a large proprietary dataset.

The experimental results show *UpKNN*'s capability to update 100 millions of items in roughly 40 seconds, scaling both in the number of updates processed and the threads used in the computation. Thereby, *UpKNN* achieves speedups ranging from 13.64X to 49.5X in the processing of millions of updates, with respect to the performance of a non-partitioned baseline.

Various experiments prove that *UpKNN*'s performance is achieved by a the right combination between the reduction of the random disk operations and our efficient multithreading design. This design minimizes the need of thread synchronization, aiming to exploit full parallelism. Besides, we show that these results have been achieved by performing roughly 1% of the disk operations performed by the baseline. Experiments also show that *UpKNN* processes an average of 3.2 millions updates per second, making our approach a good solution for online KNN processing.

1.2 Publications

The contributions in this thesis are reflected in the following publications:

As the main author:[†5]

Chiluka Nitin, Kermarrec Anne-Marie, and **Olivares Javier**. *The Out-of-Core KNN Awakens: The light side of computation force on large datasets*. In the 4th International Conference on Networked Systems (NETYS), Marrakech, Morocco, 2016. Springer International Publishing. **Best paper award**. [24]

Kermarrec Anne-Marie, Mittal Nupur, and **Olivares Javier**. *Multithreading approach to process real-time updates in KNN algorithms. (In submission, notification 15th November 2016)* 25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), St. Petersburg, Russia, 2017. IEEE. [53]

Chiluka Nitin, Kermarrec Anne-Marie, and **Olivares Javier**. *Scaling KNN computation over large graphs on a PC*. In the ACM/IFIP/USENIX Middleware conference, Bordeaux, France, 2014. ACM. [23]

As one the main authors:

Rodas Jorge, **Olivares Javier**, Galindo José, and Benavides David. *Hacia el uso de sistemas de recomendación en sistemas de alta variabilidad*. In Congreso Español de Informática (CEDI), Salamanca, Spain, 2016. [89]

1.3 Organization of the manuscript

The remainder of this thesis is organized as follows:

Chapter §**2** describes and defines the *K-Nearest Neighbors* problem and its computational challenges. This chapter provides the general background to understand the contributions presented on this thesis. Additionally, this chapter presents the most important works belonging to the state-of-the-art on KNN computation.

Chapter §**3** presents our preliminary work on KNN computation [23]. This work is a first approach that led us to discover the main challenges of scaling the KNN computation on single machines. Thereby, [23] becomes an initial

[†5]The lists of authors are arranged alphabetically

version of our main contribution *Pons*, which successfully addresses the computational challenges and drawbacks discovered in [23].

Chapter §**4** describes in details our main contribution *Pons* [24], an out-of-core algorithm for computing KNN on large datasets respecting the computational limitations imposed by the use of single machines.

Chapter §**5** presents *UpKNN* [53], our second contribution, a multithreading approach for processing large streams of profiles' updates in KNN algorithms.

Chapter §**6** concludes this work and presents some perspectives and future work.

Chapter 2

Background and state-of-the-art

In this chapter, we describe in details the K-Nearest Neighbor method, its main computational challenges, and the fundamental concepts required to understand the main contributions of this work. Additionally, we present the most significant works on each topic addressed in this thesis.

In Section §2.1 we define the concepts that support the KNN computation. To understand its importance, and growth in popularity, in Section §2.2 we exhibit various applications that have made use of the KNN algorithm, demonstrating that it is a powerful tool for a wide variety of areas. Then, in Section §2.3 we present the two general approaches for computing the KNN. In Section §2.4 and §2.5 we describe a set of works that perform KNN computations implementing distributed or out-of-core algorithms, respectively. Finally, in Section §2.6 we motivate the need of computing KNN over data that changes continuously over time, and we present the most relevant works belonging to the state-of-the-art in this specific topic.

2.1 Definition and notation

A K-Nearest Neighbors algorithm (*KNN* onwards) is a method used to solve several data classification problems [26]. KNN, a specific case of the general problem of the *nearest neighbors (NN)* search, is formally defined as follow [8]. Given a set V of data points or entities ($|V| = N$), defined in a D-dimensional data space, find the K closer (more similar) points to each data point v in such a data space.

The process of searching the K-nearest neighbors of a data point is based on two fundamental elements: data points' profiles on which similarities are

computed, and a well-defined distance (similarity) function [17, 83]. On the one hand, each entity $v \in V$ is represented by a profile F_v, being F_v the set of data that describes, defines or represents v in the D-dimensional data space. On the other hand, the distance (similarity) function is used to determine the distance (closeness) or similarity between two data points in the data space. Consequently, two data points (entities) are similar if their profiles are similar, based on the comparison made using a similarity or distance metric such as cosine similarity or Jaccard coefficient [17, 33, 68].

While there are many different methods to address the problem, the general procedure to perform the search of v's similar entities remains fairly intuitive: v's profile is compared either against the whole set of data points V or a bounded fraction of it. After comparing the set of profiles, the list of the K most similar entities become B_v, the KNN of v. In Section §2.3, we describe in depth the general methods to perform the computation of KNN.

2.2 Applications

The use of the KNN algorithm has been widely spread to very different areas of application as medicine [2, 58, 61, 122], business [27, 73, 115], music [34, 36, 56, 105], as well as computer science [13, 30, 47, 89, 109, 112, 127], among others [3, 19, 121]. Specifically, it is in computer science where KNN has reached its maximum level of development, spanning years of research in several topics such as recommender systems [11–13], image classification [10, 78, 113], or information retrieval [30, 51, 116].

Regarding its application on recommender systems, KNN algorithm has been a fundamental tool in *content-based* [104, 130], *user-based* [11–13] and *item-based* [84, 92] *collaborative filtering (CF)*.

Firstly, KNN is used in content-based approaches to search similar items among a set of them. To clarify its use in content-based recommender systems, let us consider the following example. Let a list of books be the set of items in a recommender system. Each book is represented by a profile, which may consist of the book's genre, publication year, number of pages, author, etc. In this example, two books are consider as similar if their profiles exhibit a high degree of similarity, based on a given similarity metric. Therefore, the following two George R.R. Martin's books: *A Clash of Kings* and *A Feast for Crows*, may be considered as similar because they share the genre, author and number of pages. Once the algorithm finds similar items, it generates recommendations to users who have rated at least one of the items in the list

of similar items. This action is based on the assumption that users who have rated a type of items in the past are likely to rate/like items that are similar to those previously rated. In our example, a reader of *A Clash of Kings* is likely to be interested in reading *A Feast for Crows*, based on the high extent of similarity between the books.

Secondly, in a user-based recommender system, each user possesses a profile, which is made of the content consumed/purchased/liked for her. Consequently, KNN searches for similar users, based on the comparison of their profiles. With the result of the KNN search, the recommender system advices each user to consume/purchase some specific items that were consumed/liked for those users similar to her.

Similarly, in an item-based recommender system, each item has a profile, which is built from the set of users that have consumed/purchased/liked such an item. In this case, the KNN algorithm is used to find the most similar items among the set. Particularly, two items are similar if their profiles are similar, that is to say, if the sets of users consuming both items exhibit a high degree of likeness.

Although the different types of recommender systems have their specific strengths and weaknesses depending on the application served, there have been some works that try to unify user-based and item-based algorithms in one [109], using KNN algorithms in a similar fashion. The main goal of this attempt of unification is to merge the main strengths of both worlds in a more robust algorithm.

Research in databases has also witnessed the strengths of KNN in several works [9, 118, 125, 127]. In this regard, multiple database applications have implemented the KNN algorithm as a tool in the *Knowledge Discovery* process. Process which analyzes databases searching for new and valuable information that can be obtained from the similarity or closenesses of the stored data. In this sense, one of the main operations supported is the *KNN join*, which leverages KNN's capacity to find similar data points in large datasets. These sets are the result of a join operation of several databases. As well as a part of the knowledge discovery process, we hypothesize that KNN algorithm can also be implemented to find similar data in databases, aiming to optimize the process of data placement. Under the assumption that similar data is likely to be retrieved by the same query, storing similar data physically close, using proactive fetching or caching adequately can improve significantly databases' performance [129].

Beyond databases, KNN has been used in the process of image classification. In such a case, the algorithm of KNN searches similar images among

a catalog of them, based on some specific representation of the images. In this context, works as [21, 47, 112] have proposed various proficient methods to compute KNN on large images. Particularly, computing KNN in this case brings several computational challenges due to the fact that the images' profiles typically are represented by high-dimensional data, which may span thousands or millions of dimensions in some cases. Handling and processing large images' representations pose several computational challenges due to the large memory footprints and runtimes. The phase of profile comparison, which often dominates the runtime in KNN algorithms, is particularly costly in computing KNN on high-dimensional data. Besides, high-dimensional profiles may suffer the *curse of dimensionality* [52, 94], which occurs when an important fraction of the profile dimensions does not contain a value. Unfortunately, as a large portion of the stored data does not represent valuable information, much of the resources are misused.

Finally, research in text classification [30, 40, 102, 103, 123, 124] also harnesses KNN's features. In this subject, KNN algorithms have been implemented to organize, categorize, and classify information in large sets of text collections. Specifically, KNN can be adapted to classify new documents based on similar documents already classified into the text collections. Thus, making use of KNN's outcome, each new document is classified according to its neighbors' category. Thereby, as similar documents are associated to the same category, the subsequent search and access to related information is more efficient. Similarly, KNN has been implemented to find similar content within a document or a set of documents.

Based on these multiple applications, and many others not mentioned in this thesis, we can argue that KNN's popularity relies on some facts that are easily observed on the state-of-the-art: it is simple, thus easy to understand; it is very versatile, it easily adapts to various applications; and it has shown very good results in most of the applications. Yet, the implementation and use of KNN algorithms still remain challenging as data increases in volume and dynamism.

2.3 General approaches to compute KNN

In this section, we analyze the most important techniques for computing KNN algorithm, paying particular attention to how they deal with the challenges faced by such an algorithm, namely, data growth and scalability, high-dimensional data, and data's dynamism. First, in Subsection §2.3.1 we describe the two general approaches for computing KNN: the brute-force al-

gorithm and approximate algorithms. Subsection §2.3.2 describes how the graphs lend themselves particularly well to model and process data in the KNN computation. Specifically, in the last subsection, we describe the specific KNN algorithm implemented in our main contribution.

2.3.1 Brute-force versus approximate approaches

There have been presented several works that propose techniques to compute KNN efficiently in terms of computational performance, resource consumption and algorithm's precision. Based on all those works, KNN algorithms can be divided in two general approaches: *brute-force algorithms*, and *approximate algorithms*.

On the one hand, a *brute-force* approach computes the K-nearest neighbors in the following manner (Algorithm §1). For a set V of N entities ($|V| = N$), computing exact KNN of a given entity v in a *brute-force* fashion means that each entity v's profile F_v is compared with all other $N - 1$ entities' profiles (lines §3–§9), using the aforementioned distance/similarity function (line §7), namely cosine similarity or Jaccard [17, 68]. Then, the K closest entities are chosen as v's KNN (line §9). Unfortunately such an approach is time-efficient only when the number of entities to compare is small. Such an approach has a time complexity of $O(N^2)$, making it very inefficient for a large N.

Algorithm 1: Brute-force KNN pseudo-code

```
   Data: Set of entities V (|V| = N), Set of profiles F
   Result: Each entity v ∈ V finds its KNN B_v
1  begin
2      B ← ϕ                    /* KNN data structure of size N */
3      foreach v ∈ V do
4          TopK ← ϕ             /* TopK: heap of size K */
5          foreach u ∈ V do
6              if v ≠ u then
7                  SimValue ← SimFunction(F_v, F_u)
8                  Update(TopK, u, SimValue)
9          B_v ← TopK
```

On the other hand, the second type of approach for computing KNN are those in the category of *approximate* approaches. Unlike exact approaches, *approximate* KNN computation searches iteratively the nearest neighbors of

an entity, based on a rather local search of closer neighbors, improving results as the iterations of the algorithm progress. Although *approximate* approaches do not find exact results as brute-force approaches do, after some iterations they are able to gradually improve results, reaching in most cases, high-precision results. These good results come along with a significant reduction of the computational complexity, scaling better than the brute-force approaches, particularly running on very large datasets.

In spite of the high-cost of computing exact KNN using brute-force approaches, some works [37, 38, 60, 70, 71, 97] have taken advantage of the processing power of Graphics Processing Units (GPU) to overcome the challenges imposed by such an intensive computation. These works perform fully parallel operations of profile comparisons, based on the observation that these operations are completely independent. Although profiles are accessed concurrently in the process, there is no need of synchronization between threads, since all threads perform only read operations on the profiles. Such a parallel access to the memory delivers the best performance possible on GPU-based algorithms. Despite the good results observed on these works, they still fail to show good scalability on very large datasets.

As a result of the computational cost and lack of scalability of the brute-force approaches, a long list of *approximate KNN algorithms* have surged to overcome the drawbacks observed. The main advantage of the approximate algorithms is that they significantly decrease the computation complexity, while the results' quality remains high. Approximate KNN computation differs from brute-force in the fact that, instead of searching better neighbors among the whole set of entities, it performs iterative local searches of closer neighbors, limiting the search space. This optimization reduces computational complexity in terms of runtime and memory overhead, without affecting significantly algorithm's precision.

As we mentioned above in Section §2.2, one the greatest challenges in computing KNN is the high dimensionality of the profiles. Bearing this in mind, some approximate approaches have been proposed to perform KNN operations efficiently on high-dimensional data. One of such approaches are those based on Local Sensitive Hashing (LSH) [28, 39, 41, 81, 101]. LSH-based approaches use *locality sensitive* hashing functions to map similar data into the same hash bucket. Specifically, LSH has been used to perform approximate KNN computation on high-dimensional datasets due to its capacity to reduce significantly data dimensionality. A D-dimensional data point, using one or more carefully selected hash functions, is mapped into a 1-dimensional space, and assigned to a specific hash bucket. The use of multiple hash functions aims to cope the collision problem observed on hash-based applications. Due

to the locality ensured by these functions, the set of data mapped in a bucket shows a high probability of being a good candidate for the KNN of an entity v assigned to the same bucket. As a consequence, to perform the KNN query for a data point v, other data points assigned to v's hash bucket are compared, selecting the K most similar.

Unfortunately, despite the results showed in these works, the efficiency and performance of LSH-based algorithms remains highly dependent on the right choice of the hash functions.

Alternatively, Yu et al. in [126] proposed *iDistance*, a KNN method for processing high-dimensional data. In this work, they presented a three-phase approach to perform both efficient KNN search and similarity range search. First, high-dimensional data is divided in m partitions. Second, for each partition, a reference point is identified. Third, based on the reference points, each high-dimensional point is mapped to a single dimension space based on its distance to the nearest reference point. Once high-dimensional points are mapped to a single dimension space, they are indexed using a B^+-tree. This data structure supports efficient one-dimensional search. While *iDistance* [126] shows good results, it is strongly dependent on the partitioning used and the reference points selected.

With a similar goal in mind, Chen et al. in [20] proposed an iterative *divide and conquer* procedure to compute KNN on multidimensional data. This *divide and conquer* approach uses a Lanczos procedure to divide recursively the data into two subsets, computing KNN for these smaller subsets. A second stage of the algorithm, the *conquer* phase, gathers the results into the final KNN graph. Aiming to improve performance through the reduction of comparison operations, this work uses a hash table to store the comparison already performed, avoiding duplicated operations.

2.3.2 Approximate approach: KNN as a graph

In recent years, graph processing has become a hot topic in computer science. This growing interest arises from the huge amount of data that can be naturally represented as a graph. In simple terms, a graph G is way of encoding pairwise relationships among a set of entities. Formally, a graph G consists of a collection of $|V|$ vertices (or nodes) and $|E|$ edges, each of which connects two vertices [55]. To illustrate, we can represent in a graph the data of events, road networks, neuronal networks, large-scale computer networks, or social networks, among others.

The main advantage of representing data as a graph is the existence of well-studied graph algorithms. They allow us to extract valuable information from the topology of the graph, its degree of connection/disconnection, or its evolution over time. As well as useful, extracting data from graphs is also extremely challenging, mainly because these graphs nowadays are massive, hence, processing them is highly computationally expensive. Today, the size of real graphs is very large, for instance, *Facebook* graph has millions of vertices (users) and billions of edges (friendship between two users). Another current example is the *Twitter* graph, which represents millions of users and billions of edges (follow relationships). Along with the size of the data, graph processing is also challenging since the supported operations are often complex. Consequently, designing and coding efficient graph algorithms requires a significant amount of time and a non-negligible computational power [108]. Particularly regarding the latter, a scalable processing on large graphs requires efficient usage of disk and main memory, along with high-performance data structures.

The computation on large-scale graphs and its computational challenges concern to KNN as follows. Approximate K-Nearest Neighbors computation is often seen as the iterative construction of a graph $G^{(t)} = (V, E)$, where V is the set of entities, and E the set of edges that connect each entity with its K nearest neighbors at the iteration t.

In this matter, Dong et al. [32] have presented *NN-Descent*, a scalable algorithm for constructing iteratively the KNN graph using an arbitrary similarity metric. *NN-Descent* starts from a fully random graph, and builds iteratively a more precise KNN graph, based on the principle that *a neighbor of a neighbor is also likely to be a neighbor* [32]. In fact, *NN-Descent* demonstrates the high probability of finding closer neighbors among the list of neighbors' neighbors of a vertex in the KNN graph.

As we show in Algorithm §2, to improve the graph iteratively, *NN-Descent* works as follows. In a set of vertices V, let B_v be v's KNN, and $R_v = \{u \in V \mid v \in B_u\}$ be the reverse KNN of v. Then $\overline{B}[v] = B_v \cup R_v$ is the general set of v's neighbors. Based on $\overline{B}[v]$'s elements, the algorithm reaches all v's neighbors' neighbors to perform the profile similarity comparisons, and selects the v's K-closer vertices in the current iteration. Such an iterative search stops when the number of updates in vertices' neighborhoods is less than δKN, being δ a precision parameter.

Algorithm 2: *NN-Descent* pseudo-code

Data: Entities set V ($|V| = N$), Profile set F, similarity metric σ, δ
Result: Each entity $v \in V$ finds a good KNN approximation B_v

```
1  begin
2      foreach v ∈ V do
3          B_v ← RndSample(V, K)        /* K random entities from V */
4      while c > δKN do
5          c ← 0
6          foreach v ∈ V do
7              R_v ← {u ∈ V | v ∈ B_u}
8              B̄[v] ← B_v ∪ R_v
9          foreach v ∈ V do
10             foreach u_1 ∈ B̄[v], u_2 ∈ B̄[u_1] do
11                 SimValue ← σ(F_v, F_u2)
12                 Update(B_v, u_2, SimValue)    /* B_v heap of size K */
13                 if B_v has changed then
14                     c ← c + 1
```

Several other works developed approaches for building the KNN graph, using the graph as a fundamental data structure to support various operations [25, 35, 42, 45, 82, 83], such as nearest neighbor search and range similarity search. In all these works, KNN graph's popularity relies on its capacity to help in reducing costly distance computation operations. Similarly, Hajebi et al. [42] developed a nearest neighbor search algorithm based on a *hill-climbing* process. At first, this work builds an offline KNN graph. Later, to find the nearest neighbors of a new point Q, it starts from a random node of the KNN graph and moves forward employing a greedy hill-climbing search, selecting closer neighbors based on a distance measure. The hill-climbing search stops after a fixed number of moves.

Additionally, Paredes et al. [82, 83] also build and use a KNN graph to perform proximity search, more specifically in finding similar data in large databases. In these works, the KNN graph indexes the database, allowing fast retrieval and reduction of the number of distance computations performed for searching similar data.

In turn *HyRec* [13], a decentralized recommender system, employs the construction of the KNN graph to find similar users in the system, recommending new items to the users based on their similarity. Due to its importance in this thesis, in the following Subsection §2.3.2.1 we will describe this

approach in more depth.

Similarly, *Gossple* [7] a fully decentralized social network, leverages *nearest neighbors* graph construction to build an implicit social network based on users' similarity. Using this similarity-based network, the system improves users' navigation experience, personalizing the data that users receive, guided by similar users' preferences. Consequently, users only receive information of their interest, avoiding the flood of information that exists nowadays.

Although the list of works building an approximate KNN graph exhibit outstanding results in terms of the approximation's quality, and runtime, they were all implemented several years ago, using the resources available of that time, and they processed graph much more smaller than those found these days. As we already mentioned, processing massive graphs brings a wider range of new computational challenges.

2.3.2.1 HyRec

In this section we describe the KNN algorithm implemented in *HyRec* [13], a recommender system running on a hybrid architecture. As a matter of fact, *HyRec*'s KNN algorithm is a decentralized version of *Gossple*'s KNN algorithm [7]. In *HyRec*, the KNN algorithm identifies similar users in the system, according to a given similarity metric. Afterwards, the recommender system, based on users' similarity, recommends new items to them. The simplicity and excellent results showed by *HyRec*'s KNN approach leads us to adopt this algorithm in our work, scaling its computation on single machines, as we detail in Chapter §4.

As we observe in Algorithm §3, *HyRec*'s KNN algorithm works iteratively as follows. At each iteration t, each entity's v current K-nearest neighbors B_v are selected from the candidate set S_v. The candidate set S_v (line §5) contains the current set of v's neighbors, v's neighbors' neighbors (or two-hops neighbors), and K random entities (random entities prevents the search from getting stuck into a local optimum). To select v's KNN from S_v, the system compares v's profile with that of each user in S_v (line §9) using the cosine similarity metric [33], and selects the list of the K most similar to v (line §11). Although *HyRec*'s KNN algorithm exhibits some resemblance with Dong's *NN-Descent* approach, it yields a major difference in the elements of the candidate set. *HyRec*'s KNN algorithm does not consider reverse KNN as *NN-Descent* does. While this optimization in *HyRec* reduces computational complexity, it does not significantly impact KNN's graph quality. Additionally, *Hyrec*'s main contribution is the fact that the computation of the KNN is distributed,

NN-Descent, meanwhile, works in a centralized fashion.

Algorithm 3: *HyRec*'s KNN algorithm pseudo-code

Data: Set of entities V ($|V| = N$), Set of profiles F
Result: Each entity $v \in V$ finds a good KNN approximation B_v

```
1  begin
2      foreach v ∈ V do
3          B_v ← RndSample(V, K)   /* KNN data structure of size N */
4      foreach Iteration t do
5          S_v ← B_v ∪ (⋃_{u∈B_v} B_u) ∪ Rnd(K)
6          foreach v ∈ V do
7              TopK ← ϕ            /* TopK: heap of size K */
8              foreach c ∈ S_v do
9                  SimValue ← SimFunction(F_v, F_c)
10                 Update(TopK, c, SimValue)
11             B_v ← TopK
```

In the following example in Figure §2.1, we observe the candidate set for the user *Cliff*, highlighted in a blue circle. *Cliff*'s candidate set consists of the list of his neighbors (connected through blue lines) *Kirk*, *Alice*, and *Jason*; the list of his neighbors' neighbors (connected through green lines) *John*, *James*, and *Janis*; and K (3 in this example) random neighbors (highlighted in an orange circle) *Lars*, *Dave* and *Robert*. From this candidate set, the algorithm selects the K-most similar users as the new *Cliff*'s KNN.

2.4 Distributed computation

Within the last years, the volume of data to process has reached limits that makes its computation a very challenging task. Current large-scale datasets demand an important computational power, spanning terabytes of data to be stored and processed appropriately. As we can observe, computing terabytes of data efficiently requires large amounts of main memory. Unfortunately, the cost of the main memory has not decreased as the cost of hard disks or SSDs does, making it unfordable in many cases.

Regarding what concerns us, running KNN algorithms on large current datasets requires large amount of resources. KNN algorithms not only maintain data structures handling the list of current K-nearest neighbors of each

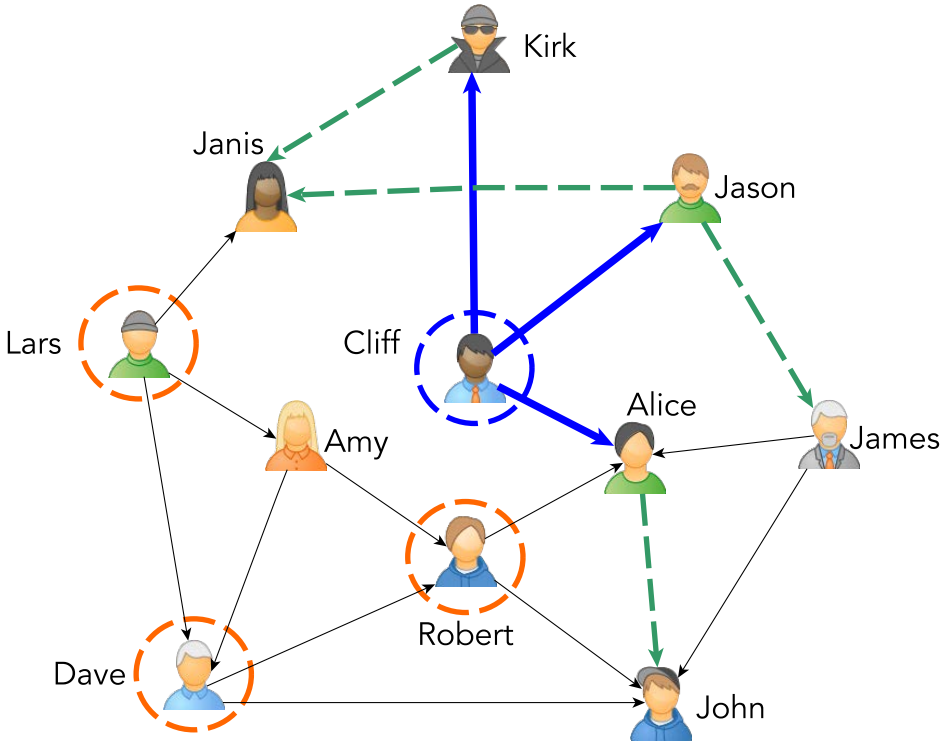


Figure 2.1: *Example of HyRec's KNN candidate set.*

entity, but also the large set of entities' profiles. Each entity in a KNN computation is represented by a profile, which varies depending on the application using the KNN algorithm. In a recommender systems case, each entity represents a user of the system, and profiles constitute users' tastes, books read, purchased products, movies seen, etc. Considering an information retrieval application, entities become data in a database, and profiles store some data's metadata that defines or characterizes it. With the growth of data volume over the years, handling large entities' profiles becomes a greater challenge.

Looking for a solution to address these challenges, we may find in distributed systems an efficient technique to handle large volumes of data. To do so, distributed systems gather the computational resources of a set of machines, making them available for the user as a whole. The set of machines behind a distributed system communicates through a network, creating a single global vision of the available resources. As we mentioned above, computing large amount of data in commodity single machines requires a large amount of resources, which are not always available. Rather, distributed systems allow us to gather the resources of a set of machines, and devote the whole to

the computation, allowing the user to have access to larger sets of resources, enough to process the increasingly larger volumes of data that we can observe nowadays.

Due to the growth of data volume mentioned above, research in KNN computation has also made use of distributed systems [6, 41, 86, 98, 128] to achieve much greater computing power in terms of available memory and storage capacity. Besides computing power itself, the use of distributed systems to run KNN algorithms call for designs able to face the challenges posed by characteristics of the KNN graph. Firstly, a graph that shows a near-random access to data. Secondly, a graph that often generates load unbalance due to some highly-connected entities, a common fact in scale-free networks [4].

While processing larger datasets is possible on distributed systems, a good algorithm's design may lead to better performance, surpassing performance achieved even on highly optimized algorithms running on single machines or clusters. But, this remarkable performance always comes at a high cost in terms of design, coding, and deployment effort.

In the following, we describe some approaches that address the problem of computing KNN in a distributed fashion, along with the major challenges arising from this type of computation.

2.4.1 Cluster-based approach

Plaku and Kavraki [86] proposed a distributed framework for computing KNN algorithm on a cluster of processors through a message-passing approach. They divide the computation in two tasks: firstly, each processor handles a partition of the set of entities, computing a local KNN query. Second, each processor receives messages from other processors, querying its own data to answer external requests. Finally, it sends the results of the queries to the other processors. Communication between processors is performed through a cache, which is filled with external queries. Aiming to obtain the best possible performance, each processor answers external queries in the cache as soon as possible, reducing others processors' idle time. Each request in the cache receives a weight, which is directly proportional to its waiting time in the cache, and inversely proportional to the number of requests in the cache owned by the same processor. Local processing of each processor is delayed until all requests in cache have been issued.

2.4.2 MapReduce approaches

Zhang et al. [128] address the challenge of performing distributed KNN join operations on large datasets. For doing so, they proposed a *MapReduce* parallel approach that performs both the join operation and the KNN search. The authors proposed in a prior work, a similar approach implementing a *block nested loop join*, which unfortunately did not show good scalability on large and multidimensional data. These results led the authors to propose a more efficient approach, which maps multidimensional data into one-dimension data using *space-filling curves*. As a consequence, KNN join operation becomes a much more simple one-dimensional range search, significantly improving system's performance.

Following a similar deployment in a *MapReduce* infrastructure, Stupar et al. proposed *RankReduce* [100], an approach that uses Locality Sensitive Hashing (LSH) to assign similar data closer in the distributed file system, improving in that way, nearest neighbors query's performance.

Although MapReduce paradigm has shown good results running complex algorithms, it sadly does not exhibit such a performance on algorithms as KNN or some other graph-based algorithms [22, 74, 77]. These dissimilar results are mainly explained by the near-random access pattern needed to process many graphs, specially those composed of highly-connected components. Consequently, these near-random accesses degrade performance due to the increase in the data exchanged between machines during the profile comparisons.

2.4.3 P2P approaches

Leveraging the benefits of this well-known type of distributed system, some works as [6, 41] have implemented a fully decentralized approach (P2P) to compute KNN queries. They take advantage of the resources provided by a set of machines, being able to process in parallel larger sets of query objects. In this regard, Batko et al. [6] proposed an extension of their previous work [5], a distributed metric structure named *GHT** to support nearest neighbors queries. To do so, each peer in the P2P network stores data objects in buckets, which are retrieved when the nearest neighbor query is performed. *GHT**, using a *Address Search Tree* (AST), navigates through the set of distributed buckets when is needed. Consequently, the nearest neighbor search occurs in the following manner. Each new object q is inserted using a function ψ that traverses the AST to find the right bucket for q. Later, a nearest

neighbor search for q starts comparing q with all the other objects inserted in its bucket. From the list of its K most similar, the K-th object is used as a radius for a range search of nearest neighbors, in other words, the current K-th nearest neighbor is an upper bound for similarity. Similarly, the range search finds all neighbors with similarity greater or equal to its K-th object in its KNN list. Using this approach, nearest neighbor query for q returns the exact answer.

In the same context of P2P systems, Haghani et al. in [41] investigate the use of Locality Sensitive Hashing (LSH) functions to assign similar data into the same peer. Such an assignation improves nearest neighbors search queries in terms of network messages sent over the network. In this work, a *p-stable* LSH is used to fulfill two major requirements: assignment of similar data in the same peer, and good load balance. Based on this *p-stable* LSH schema, the system assigns similar data in the same bucket, and similar buckets in the same peer. This assignation increases the probability of answering nearest neighbors search using only peer-local data. Consequently, it reduces significantly the communication through the network. A nearest neighbors search query for some data q starts on the peer that stores q, then continues in the predecessor and successor peers (in a Chord-based P2P network [99]). Whether among the buckets of these peers, it does not find more similar neighbors, the search stops. Otherwise, it continues on the next peers in the ring.

In the same line we find *WhatsUp* [11], a news recommender system, which deploys a P2P system to scale the computation and deal with the possible privacy issues faced in systems with a central authority having control over users' profiles. This P2P-based news recommender maintains a dynamic social network based on links between nodes in the system. These nodes are linked based on their similar interests. Although *WhatsUp* is not primarily computing KNN as such, it is, roughly speaking, a *nearest neighbors*-wise recommender system.

WhatsUp uses this similarity-based network to disseminate content among similar users, relying on the assumption that users with similar interests in the past are likely to share tastes in the future. In order to maintain this similar users linked, *WhatsUp* samples, for each user, a fraction of the network looking for some more similar users. This sampling process compares users' profiles using a specific similarity metric, which favors the connection between users that share more common items in their profiles.

Finally, *Gossple* [7] is a decentralized approach to enhance web navigation based on an anonymous and implicit social network. *Gossple* uses the *nearest neighbors* concept to create a network of anonymous *acquaintances* for each user in the system. This implicit *acquaintance* network is created based on the similarity between users' profiles. The main difference with an explicit social

network as *Facebook* is that *Gossple* links users based on their preferences, without revealing other users' profiles, preserving anonymity and privacy.

In *Gossple*, each user has a personalized view of the network, which leverages to improve her navigation. This view is composed for a set of similar users found across the whole network using a *Random Peer Sampling protocol (RPS)* [49] and a multi-interest clustering protocol called *GNet*. From this set of samples, for each user in the system, it selects the most similar users, comparing their profiles using a specific similarity metric.

Gossple leverages the features of P2P to ensure scalability and privacy, while it reduces the bandwidth consumption. The latter is accomplish by an efficient exchange protocol during profile comparisons. Initially, this protocol does not exchange the entire profile but only a compressed version of them obtained through a *Bloom Filter*. Then, the filtered profiles are compared, revealing if these profiles may be similar. In this point, only if the compressed profiles seem to be similar, the system exchanges the real profiles, consuming network bandwidth only when it is needed.

2.4.4 Challenges in KNN distributed computation

One of the major challenges in distributed computation is the finding of an efficient partitioning algorithm [46]. As data cannot be stored and processed on a single machine, it should be partitioned across a set of machines. Thereby, each machine handles a fraction of the data, performs the corresponding computation, and communicates with other machines through the network. This communication is performed to deliver partial or global results to other machines.

Although many research has been done on this issue, specially applied to large graphs [1, 54, 87, 106, 107, 114, 120, 131], partitioning is still a challenging and open area. This operation is particularly complex nowadays due to the growth of the volume of data to partition, and the variety of applications using partitioned data. One of such application is the partitioning of large KNN graphs across several machines. In this sense, to support efficient distributed KNN computation, an effective partitioning algorithm has to exhibit at least the following features: i) Minimal edge-cut of neighbors and neighbors' neighbors, ii) edge/vertex balance across partitions, and iii) lightweight processing.

Firstly, a distributed KNN partitioning algorithm must reduce edge-cut of edges between neighbors and neighbors' neighbors. In a KNN context,

edge-cut means that an entity and its neighbors or neighbors' neighbors are partitioned across different machines. Thus, during the profile comparison process the distributed algorithm incurs in inter-machine operations, therefore the performance of this process depends on the network bandwidth. In contrast, when there is no edge-cut, an entity and its neighbors and neighbors' neighbors share the same machine, hence the process of profile comparison entails only intra-machine operations. Consequently, operation's performance is bounded only for the main memory bandwidth, which generally shows higher bandwidth compared to that of the network.

Secondly, an efficient distributed KNN partitioning algorithm must exhibit a good balance in the number of edges/vertices assigned to each partition. An unbalanced partitioning algorithm leads to partitions that handle more data than others, therefore, some machines are overloaded, either by their own computations or by the communication with other machines. In other words, a machine handling a larger set of vertices of the graph, computes KNN over a larger set of vertices, having more communication tasks to perform. Such an unbalance, considerably affects performance, because algorithm's completion is bounded by the performance of the most loaded machine.

Thirdly, a partitioning KNN algorithm must be lightweight. KNN graph changes over time, each entity in the graph has K out-edges to its K most similar neighbors at the current iteration. Therefore as iterations progress, each entity changes its neighbors. At the beginning of the computation, neighborhoods evolve fast, after some iterations, they change very little [66]. As a consequence of the changes in entities' neighborhoods, edge-cut and partitions balance are modified, affecting overall algorithm's performance.

In order to maintain edge-cut and partitions balance optimized, data must be re-partitioned at each iteration, adding an extra overhead to the computation. Bearing this overhead in mind, the partitioning algorithm must be as lightweight as possible. While we could say that partitioning data only at the beginning of the computation eliminates such an overhead, unfortunately edge-cut and balance change continuously over iterations. Therefore, as a result of a static partitioning, the system suffers a degradation, performing poorly. Accordingly, we observe a trade-off between re-partitioning overhead and system's performance. Thus, a good approach to face this trade-off is the implementation of a lightweight algorithm that re-partitions the data at each iteration, requiring little time to complete the task.

2.5 Out-of-core computation

In this section, we describe the concepts behind the out-of-core algorithms. This approach has shown to be an efficient way to scale the processing of large datasets running on single machines.

2.5.1 Motivation

Even though using distributed systems seems to be a good option to address the problem of computing KNN on large datasets, the processes of coding, debugging, optimizing and testing efficient distributed algorithms, involve a great effort. Moreover, the deployment and subsequent performance of the distributed algorithms are limited by the availability and capacity of the machines that compose the system. It should be mentioned that the use of large set of machines is not always possible, either by the lack of resources or higher demand from users.

Considering the above, the efficient use of commodity single machines brings new opportunities in the development of more efficient and lower cost algorithms. As we can observe, nowadays the access to cheaper personal computers is possible for almost everyone. Thus, the development of scalable algorithms running on single machines opens a new range of possibilities, available to a wider range of users.

Despite its advantage in terms of cost, processing KNN on large datasets using single commodity machines brings several computational issues. The main challenge arises from the fact that the set of entities' profiles, along with the graph structure, may have larger memory footprints than those that a single machine can handle. Consequently, data cannot be completely loaded in memory, resulting in algorithms with poor performance caused by the use of virtual memory [31]. Although useful as an abstraction, virtual memory is much slower than the real memory. In this context, aiming to get better performance, appear the *out-of-core* algorithms[†1].

Out-of-core algorithms are those that use the disk as an extension of the main memory, in cases where the data cannot be completely loaded in memory [111]. This sort of algorithm loads in memory the data required for the current computation, while the remaining is stored in disk, and retrieved only when needed. Because of this usual retrieval, the main challenge of using out-of-core algorithms emerges from the non-contiguous accesses to data, one of

[†1]Some authors use the terms *External memory algorithms* or *I/O algorithms* instead

the most common cases in graph algorithms [76]. Regular algorithm's operation accesses blocks of data in disk, and load them to main memory, therefore, if such a disk accesses are random, I/O operations become a major bottleneck. Additionally, the lack of data locality adversely affects performance in out-of-core algorithms [85].

In such a scenario, a smart design of out-of-core algorithms is fundamental for exploiting data locality and for reducing I/O costs. For such a purpose, it is important to make optimal use of disk read operations. In other words, reading at once as large and sequential blocks of data as possible. Reading sequentially large blocks of data, instead of randomly, reduces I/O cost considerably [110]. Accordingly, a smart data placement is highly beneficial for creating data locality, and to favor sequential disk accesses rather than random. Along with leveraging locality and making good use of read operations, it is important that out-of-core algorithms use main memory in an efficient way, avoiding the use of virtual memory, because it does not perform similarly.

2.5.2 Scaling out-of-core algorithms

An important amount of research has been done during the last years on efficient out-of-core algorithms. In this regard, large graph processing has received most of the attention [43, 63, 64, 90, 91, 119, 132]. This growing interest has led to algorithms capable of processing graphs of billions of edges or beyond, in the most recent works. In the effort of processing increasingly larger graphs, researchers have designed highly efficient algorithms running both on single commodity machines [43, 63, 64, 91, 119], as well as on clusters [90, 132].

Motivated by the fact that the design, implementation, and deployment of distributed system is extremely time-consuming and expensive, Kyrola et al. proposed *GraphChi* [63], an out-of-core graph processing system, which is capable of executing well-known graph algorithms [62], and database operations [64], on graphs with billions of edges on a single machine. The evaluation of this work shows that *GraphChi* handles large graphs in reasonable time, running on a commodity PC, therefore the cost and effort of development is less than that of distributed cases.

GraphChi's performance relies on an efficient method to store data sequentially in disk, improving performance during the process of data loading from disk. To perform the computation on edges, these are retrieved from disk using a *parallel sliding window*, which reduces the non-sequential disk accesses.

Based on a similar motivation, *X-Stream* [91] proposes a single machine approach to perform several graph computations. *X-Stream* is a edge-centric approach based on a scatter-gather programming model, which iterates over the list of edges of the graph to perform the computation. *X-Stream* outperforms *GraphChi* [63] by avoiding edge pre-processing time. In this sense, *X-Stream* does not sort edges as *GraphChi* does, conversely, *X-Stream* streams sequentially the unsorted set of edges from disk, based on the observation that sequential accesses to Solid State Drives (SSD), Hard Disks (HDD) or RAM deliver better bandwidth than random accesses.

Moreover, aiming to go further, Han et al. proposed *TurboGraph* [43], a graph engine that runs graph algorithms on billion-edge graphs. To do so, *TurboGraph* is implemented as a multithreading approach to run on multicore machines. *TurboGraph*'s design improves two performance issues observed in *GraphChi* [63]. First, they noticed that *GraphChi* does not exhibit full parallelism throughout the computation. Second, *GraphChi* performs two separate phases for I/O and computation. It is important to highlight that the processing of large graph implies a significant amount of I/O operations on out-of-core algorithms, therefore, overlapping I/O and computation is a key optimization to improve system's performance.

TurboGraph implements a *pin-and-slide* model, which divides vertices in pages. Later, to process a vertex of the graph, it pins vertex's list of pages in memory, it applies some kind of computation on them, and unpins these pages when they are no longer needed. The set of vertices is processed in parallel, overlapping I/O requests (for reading vertex's pages from disk) and the actual computation on vertices' pages already pinned to memory. Along with the overlapping of I/O and computation, *TurboGraph* also exploits internal parallelism in modern SSDs, achieving good performance. As a result of its careful design, *TurboGraph* outperforms *Graphchi* [63] by 4 orders of magnitude.

Thinking about a much simpler yet efficient design, *MMap* [72] introduces the concept of *Memory Mapping* on graph processing approaches. Memory mapping is a OS mechanism that maps data on disk to the virtual memory space, giving us the impression that data is already in memory. The inclusion of memory mapping on out-of-core algorithms simplifies the design and implementation of disk-based approaches, assigning the tasks of accessing memory or disk and code optimization to the operating system.

The works presented above have shown to be extremely efficient out-of-core approaches, running on graphs whose structure remains static during the entire computation. Unfortunately, they do not show same efficiency when the graph changes over time, as the case of the KNN computation. Besides, to

the best of our knowledge, there are no recent works scaling KNN on single machines through an out-of-core algorithm.

Improving I/O operation's performance

As it was mentioned above, I/O operations on out-of-core algorithms consume a significant fraction of the time. With this in mind, Zhu et al. designed *GridGraph* [119], a graph processing system that proposes a two-level partitioning method to process vertices efficiently, in such a way that I/O operations are reduced. *GridGraph* creates 1-dimensional chunks of vertices, and a 2-dimensional edge blocks. Such a grid-based approach, supports the streaming of large blocks of edges from disk using a sliding window, reducing the I/O operations. Besides, *GridGraph* implements a selective scheduling of the blocks of edges to process, avoiding I/O requests for loading unnecessary data.

Aiming to increase throughput and process larger graphs with respect to a single machine, *FlashGraph* [132] uses a cluster of machines to reach an aggregated memory larger than the size of the graphs. This approach leverages parallelism and Input/Output Operations Per Second (IOPS) of an array of SSDs to perform graph computations. As well as *TurboGraph* [43], *FlashGraph* hides latency by overlapping I/O requests and computation. Additionally, *FlashGraph* improves I/O throughput by merging I/O requests.

With a similar goal in mind, *Chaos* [90], an out-of-core graph processing system, exploits aggregated bandwidth, and storage capacity of a cluster of machines. *Chaos* is based on the streaming partition idea of *X-Stream* [91] to improve sequential accesses to the storage units, while the data is processed in parallel. *Chaos*' good performance steams from the decreasing of pre-processing times, along with an efficient method to partition the graph uniformly across the servers. Such a partitioning aims to reach a good load balance, maximize parallelism, and minimize network communication overhead. Additionally, *Chaos* implements a work-stealing method to improve load balance among servers. The careful design and efficient implementation of *Chaos*, makes the system able to handle a graph of a trillion of edges, a milestone in the area of the graph processing systems.

The main concept that we can extract from this section is the overlapping of I/O requests and actual computation. This optimization hides the high latency of I/O operations to improve algorithm's performance. We observe even better performances if the overlap leverages the internal parallelism of modern SSDs. This feature implies that multiple I/O requests are served in

parallel, exhibiting better throughput than HDDs. Multiple I/O requests in parallel, along with parallel computing, have shown excellent results in previous works, specially improving the performance on I/O intensive algorithms.

Consequently, in Chapter §5 we use the idea of overlapping I/O request and computation to improve the performance on our I/O-intensive computation.

2.6 Updating profiles on KNN algorithms

In this section, we describe the motivation behind updating profiles on KNN algorithms and the main challenges posed by such a computation. Besides, we present the very few works that address these challenges. Finally, we describe the benefits arising from the profile update process, specifically those concerning the recommender systems.

2.6.1 Motivation and challenges

We have mentioned in Chapter §1 that KNN computation is mainly based on entities similarity (or proximity) comparisons. Two entities are neighbors if they exhibit an extent of similarity, which is computed comparing entities' profiles. Consequently, profiles are a fundamental component of the KNN computation: they define or represent the set of entities in the system. For instance, in a social network, profiles represent the knowledge about the things users liked, their actions in the network, social relationships with other users, etc. By comparing users' profiles in the social network, a KNN algorithm is capable to point out these users as neighbors, if their profiles are similar according to a given similarity metric. In recommender systems, profiles store valuable information about users' actions, items purchased, books read, movies rated, etc. Based on these profiles, the system maintains useful information, which is used to generate the recommendation of unrated or unknown items.

Moreover, it is a well-known fact that available data changes fast over time [57, 59]. Millions of new *tweets* are posted every day, millions of photographies posted in *Facebook*, new books, new movies, new music, new TV show episodes are released every day. Such an overwhelming amount of new data leads users to change their profiles in social networks [59]. It should be mentioned that users change their preferences not only in the short term (minutes, hours or days) but also in the long term, users' preferences in their youth, probably will not be the same in their adulthood [96].

Although the computation on dynamic profiles obviously impacts system's performance, it also brings several benefits on KNN-based applications. Specifically, the recommender systems are one of the applications that have taken advantage of the KNN algorithms [11, 13, 14]. In recommender systems' contexts, users change their profiles quite often, hence the system have to take into account those profile updates as soon as possible, and as fast as possible. Several works [16, 18, 50, 65, 67] have shown the importance of updating users' profiles over time to generate better recommendations. In this sense, Koren [57] presents a methodology to model users' preferences over time. This work analyzes the time as a key factor in recommendations' quality, having in mind that users change continuously their preference over time. Finally, the author concludes that the incorporation of temporal information improves considerably recommendation's quality.

Besides the temporal aspect, profile updating over time favors diversity in profiles and users' neighborhoods. Diversity in recommendation systems has proved to be a good way to improve recommendation quality [67, 88]. As it is claimed in [50], users like to have new and better recommendations, process whose main support is the incorporation of new ratings or items into their profiles throughout the computation.

Regarding the computational challenges, besides the high-cost of computing KNN, processing online profile updates is a computational-intensive task, specially when the set of new items' profiles to process is large. Initially, this set must be read and processed accordingly, which can be extremely costly and complex. This is specially true whether the stream is composed by millions or billions of new items, as can be the case in an online social network as *Facebook* or *Twitter*.

The cost does not come only from the reading of the stream, but also from the fact that updates have to be incorporated or merged with existing versions of the profiles. Due to the growth of data stored on entities' profiles, it is difficult to say that all data can be handled in memory. Hence, updating profiles operations incurs in multiple reading of profiles from data storage devices. Modern data storage devices, even SSDs with high bandwidth, deliver lower bandwidth than RAM. As a consequence, I/O operations performed during the process of profile update considerably affect performance, especially when the stream of updates spans millions of new items or beyond. Thinking of a distributed environment, this processing cost is also increased by network communication.

Despite the large amount of research on KNN algorithms, none of the previous works have studied in depth some of the performance issues in computing KNN on large datasets for dynamic profiles, except for [79]. Nasraoui

et al. studied performance concerns in recommender systems running on machines with limited memory. To the best of our knowledge, they were the first work that does not assume unlimited memory for KNN computation on recommender systems.

2.6.2 Current situation

Unfortunately, many KNN algorithms, specially those applied to recommender systems purposes [11, 13, 14, 57, 92, 109], do not consider profile updating in their processing. In general, those algorithms rely on profiles that remain static throughout the computation or on profiles that change after a long time window [67]. In both cases, we can observe the same drawback: computation is performed on profiles that do not represent exactly users' behavior or dynamism in users' preferences.

In the case of more generic KNN applications, the situation is not so different [66], profiles remain static over the computation, and they are updated in batches, not frequently, and in an offline manner. KNN computation over large datasets poses big challenges, both for obtaining good performance and good results in terms of classification quality (in this work we focus mainly in the former). KNN operations are generally very time-consuming and resource-greedy, specially running on large datasets. Such a high cost only worsens in cases where entities' profiles change over time.

Additionally, due to the high computational cost, KNN algorithms are mainly performed offline [9, 25, 30, 32, 35, 38, 41, 42, 47, 60, 70, 71, 81, 83, 118, 125, 126]. That is to say, KNN computation and profile updating process are performed during periods where the system is offline or under periods of lower traffic. Although it reduces cost, offline KNN computation does not consider short-term updated profiles, losing valuable information that emerges from the dynamism of user preferences.

Notwithstanding that updating profiles is a great challenge, some recent works [95, 96] have considered somehow the evolution of users' preferences over time. New items for users' profiles are modeled as a stream of data, allowing online streaming clustering. However, users' preferences are not only those in the stream, but also those already observed. Siddiqui et al. in [96], present *xStreams*, which address this concern. They use a *multi-relational stream clustering*, computing users' similarity based on profiles composed by preferences associated to them in different moments in time. To do so, *xStreams* processes the stream of new items, and gathers these items with previously obtained users' profiles.

In order to consider time-evolving preferences, *xStreams* combines two factors in the computation of users similarity: a similarity based on their past preferences, and a similarity based on their new items on the stream. The process of maintaining users' profiles up to date is performed by a back-end process.

The authors of *xStreams* properly assess the system in terms of the quality of the results obtained, but they do not consider in their evaluation the performance of the system in terms of memory footprint, runtime and scalability. In this sense, even though the system is a good tool for computing KNN on dynamic data, it does not considered the performance of the system, a key factor to support fast online computations.

Yu et al. [127] highlight the existence of well-designed KNN algorithm for static datasets, but unable to handle updates efficiently. Addressing such a current situation, they proposed kNNJoin$^+$, an incremental approach that supports KNN join operations on high-dimensional databases, computing on profiles that change over time. To do so, they incorporate a *KNN join table*, which can be updated efficiently when updates appear. One of the most interesting contributions of this work is the incremental join process, which joins new updates on the existing results, avoiding the re-computation of the KNN join. This work shows excellent results in terms of quality and runtime with respect to the baseline, unfortunately, with the results showed in [127] is not possible to draw conclusions regarding system's scalability in terms of updates processed.

2.7 Summary

In this chapter, we have presented the *K-Nearest Neighbors (KNN)* method, an efficient tool to find similar data in large datasets. Although efficient, KNN is also a very challenging and resource-greedy computation, particularly in large scale scenarios. The efficient processing of very large datasets requires large amounts of memory, and significant computation times. In this regard, *approximate* KNN algorithms have grown in popularity due to their capacity to achieve a good trade-off between computational complexity and quality of the results.

Additionally, we have studied a set of novel approaches computing KNN in a distributed fashion. The main benefit of these works is the fact that they leverage distributed systems' power to scale the computation. These approaches are able to scale the computation on very large datasets by dividing

the work among multiple machines. Unfortunately, distributed systems have a main drawback: high cost. Designing, implementing, and deploying efficient distributed algorithms is expensive, both in terms of time and money.

Overcoming this high cost, we highlight the out-of-core algorithms, which explode disk and memory to scale complex computations running on single machines. To do so, we have to pay special attention to I/O requests, a critical point in KNN algorithms, and by far the most costly operation in out-of-core approaches.

Finally, we reviewed the state-of-the-art in KNN algorithms regarding their capacity to face a big challenge posed by data nowadays: it changes continuously and rapidly.

In brief, these are the challenges we address in the following. Scaling the KNN computation on large datasets, considering their large memory footprints. To do so, we aim to use only single machines, as a less expensive, yet efficient, way to deal with large datasets. Additionally, we also address the computational challenges of handling data that changes rapidly over time while the KNN computation is performed.

Part II

Out-of-core KNN Computation

Chapter 3

Towards a scalable out-of-core KNN computation

In this chapter, we explore a novel approach to compute K-Nearest Neighbors (KNN) algorithm on a large set of users by leveraging disk and memory efficiently on a commodity PC [23]. The system is designed to minimize random accesses to disk as well as the amount of data loaded/unloaded from/to disk so as to better utilize the computational power, thus improving the algorithmic efficiency.

In [23], we study the effect of loading/unloading data from/to disk in some specific orders, based on the premise that the KNN results do not depend on the order in which we process the data. By analyzing these alternatives, we aim to find an efficient way to reduce the amount of data loaded from disk as well as to improve the memory usage.

The design described in this chapter is a preliminary approach towards a scalable algorithm for computing KNN on single machines. Consequently, this work helped us to discover and understand in depth the main computational challenges behind an out-of-core KNN algorithm running on such a computational setup. Observing the actual implementation and the experimental results, we analyze in Section §3.3, the main drawbacks of this preliminary approach, which are addressed later in our main contribution in Chapter §4.

3.1 Introduction

Frameworks such as GraphChi [63] and X-Stream [91] are increasingly gaining attention for their ability to perform scalable computation on large graphs by leveraging disk and memory on a single commodity PC. These frameworks rely on the graph structure to remain the same for the entire period of computation of various algorithms such as PageRank [80] and triangle counting [93]. As a consequence, these frameworks are not applicable to algorithms that require the graph structure to change during their computation. In this work, we focus on one such algorithm: K-Nearest Neighbors (KNN).

In simple words, the KNN computation proceeds in iterations, as follows. At each iteration t, computing KNN of a user i requires a similarity comparison of its profile with each of the profiles of all its neighbors and neighbors' neighbors, and then the top-K most similar users from this neighborhood constitute the new KNN of user i for the next iteration $t+1$. Although there are plenty of different implementations of the algorithm, we have chosen that of *HyRec* [13] (detailed in Section §2.3.2.1) due to its simplicity and efficiency.

We model the collection of KNN of each user by a directed graph $G(t)$ where each (user) vertex has at most K-outdegree neighbors. KNN computation changes the graph from $G(t)$ to $G(t+1)$, requiring the removal of edges to former neighbors and the addition of edges to new neighbors. Such features are not supported in either GraphChi or X-Stream. In addition to $G(t)$, we have a set of user profiles $F(t)$ at iteration t, which can also change over time to $F(t+1)$.

3.2 System design

Given the system constraints of a commodity PC with limited memory, our system aims to scale KNN for a large number of users whose profiles change over time by leveraging memory and disk in an efficient manner. The main rationale of our approach is to minimize random accesses to disk as well as the amount of data loaded/unloaded from/to disk. We note that inefficient accesses of disk leads to poor utility in terms of computational power, thus affecting the algorithmic efficiency of KNN computation.

3.2.1 Overview

Our approach for computing KNN at each iteration t proceeds in five phases, as shown in Figure §3.1. Firstly, the KNN graph $G(t)$ is partitioned in m partitions such that the disk and memory operations in the future phases are minimized. Secondly, we build a hash table to hold all the unique tuples (s, d) where s is a user and d is either a neighbor or a neighbor's neighbor of s. Thirdly, we create a partition interaction graph which helps in deciding the order in which partitions are loaded and unloaded so as to calculate the similarity between users in tuples generated in the previous phase efficiently. We develop some heuristics to minimize the number of operations performed to complete the process. Fourthly, we generate each user's top-K most similar neighbors from its set of neighbors and neighbors' neighbors, thus resulting overall in the new KNN graph $G(t+1)$. Finally, all the changes in the user profiles during this iteration t are *lazily* updated to $F(t+1)$ for the next iteration.

3.2.2 KNN iteration

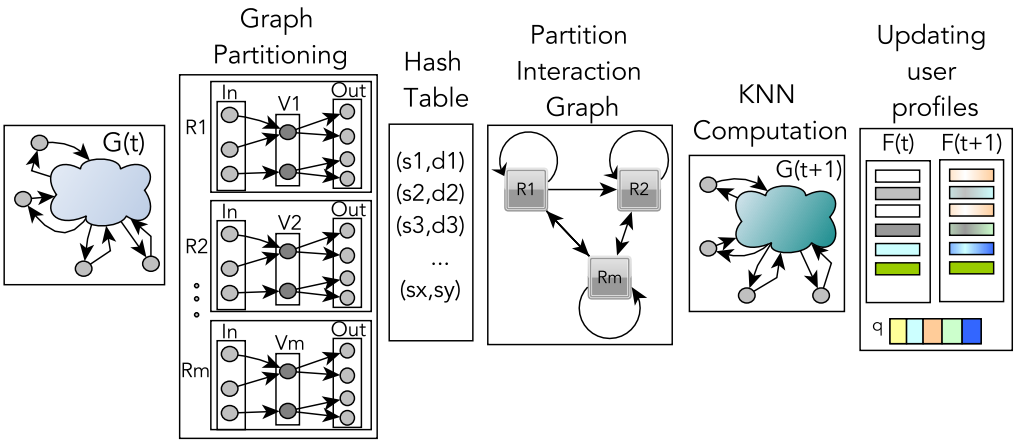


Figure 3.1: *5 phases: input $G(t)$, 1) KNN graph partitioning, 2) Hash Table, 3) Partition Interaction Graph, 4) KNN computation, 5) Updating profiles.*

3.2.2.1 Partitioning

The first phase of our approach performs *KNN graph partitioning* such that only a few small pieces of the graph as well as related data structures can be stored in memory at any given point in time while the rest are stored on disk which can be accessed efficiently later. The input of this phase is a directed KNN graph $G(t)$ at iteration t which could be at any stage in the computation: initial, intermediate, or near-convergence.

We divide $G(t)$ into m partitions, each of which corresponds to a fixed number of users $\frac{n}{m}$ where n is the number of users in $G(t)$. A partition R_i is composed of a subset V_i of $\frac{n}{m}$ users, both the in-edges and out-edges of the users V_i, and the profiles of these users. The criteria for partitioning $G(t)$ is that the total sum of the (unique) source vertices N_i^{in} of in-edges and the (unique) destination vertices N_i^{out} of out-edges in each partition i is minimized:

$$min \sum_{i=1}^{m} (N_i^{in} + N_i^{out}) \tag{3.1}$$

Such a partitioning mechanism enables a greater extent of data locality in the fourth phase.

For efficient access of neighbors' neighbors, we sort the in-edges $\{(s, v) \in R_i\}$ and the out-edges $\{(v, d) \in R_i\}$, where $v \in V_i$ and vertices s and d belong to any of the m partitions, by the vertex id v in their respective lists. One can now read the files of in-edge and out-edge lists sequentially to generate tuples (s, d) which are essentially neighbors' neighbors, since the vertex v acts as a *bridge* between s and d.

3.2.2.2 Hash table

The second phase of our approach is the creation and population of a *hash table* H. We use a hash table to avoid generating duplicate tuples which can occur due to cycles (e.g., vertices a, b and c have edges to each other) or paths with same start and end vertices but with a different bridge vertex (e.g., vertex a has out-edges to vertices b and c each of which in turn have out-edges to vertex d).

H is populated with unique tuples (s, d) representing neighbors' neighbors obtained on the first phase, as well as directed edges from the graph $G(t)$. Once H has all the tuples, the system has to compare the profiles of all tuples $\{(s, d) \in H\}$ to calculate the similarity values in phase 4. Since each tuple's

s and d could belong to different partitions, accessing their profiles from respective partitions in an arbitrary fashion can lead to poor performance due to various random accesses to disk as well as loading/unloading of partitions from/to disk.

3.2.2.3 Partition Interaction Graph

The third phase is the creation and traversal of the *partition interaction graph* which helps in deciding the order in which all the tuples' similarity scores are computed. In the partition interaction graph, each node represents a partition R_i from the first phase, and a directed edge (R_i, R_j) represents all the tuples $\{(s, d) \in H\}$ such that $s \in R_i$ and $d \in R_j$. In our memory constrained environment, we load the profiles of at most two partitions R_i and R_j at any point in order to compute the similarity scores of all the tuples $\{(a, b)\}$ such that vertices a and b belong to either of R_i and R_j. We note that when all the edges in the partition interaction graph are parsed, it means that the similarity scores of all the corresponding tuples in H have been computed.

3.2.2.4 KNN computation

The fourth phase performs *KNN computation* using the partition interaction graph and the profiles $F(t)$ to generate $G(t+1)$ which is the new KNN graph for the next iteration. First, the partition interaction graph is parsed in the order specified by one of the heuristics detailed in Section §3.2.3, such that the profiles of at most two partitions R_i and R_j are loaded into memory at a time. Next, each tuple (s, d) where $s \in R_i$ and $d \in R_j$ is read sequentially, and then a similarity score $sim(s, d)$ is computed based on their profiles. When the similarity scores for all tuples in each partition are computed, we can generate the K-most similar neighbors for each user, resulting in $G(t+1)$.

3.2.2.5 Update profiles

Finally, the fifth phase is responsible for *updating user profiles* from $F(t)$ to $F(t+1)$. Throughout the iteration t, any changes in the profiles of the users are stored in a queue q but not incorporated into $F(t)$. In this phase, the queue is read to update the profiles to $F(t+1)$. After completing this phase, the system returns to the first phase of the next iteration $t+1$, while the queue is ready for new profile updates to store.

Datasets [69]	**Nodes**	**Edges**	**Sequential**	**High-Low**	**Low-High**
Wikipedia Vote	7,115	100,762	211,856	204,706	202,290
General Relativity	5,241	14,484	34,506	32,220	31,256
High Energy	12,006	118,489	252,754	242,132	240,872
Astro Physics	18,771	198,050	420,442	400,050	401,770
E-mail	36,692	183,831	399,604	382,928	379,312
Gnutella	26,518	65,369	157,040	144,072	132,710

Table 3.1: *Number of load/unload operations using partition interaction graph.*

3.2.3 Heuristics for processing the Partition Interaction Graph

We describe a few heuristics to decide the order in which the partition interaction graph is parsed in phase 4. The *sequential* heuristic loads the partition starting from number 1, processes all its edges in the partition interaction graph, removes this partition from further consideration, and continues with next partition number 2, and so on until all edges and nodes are parsed. The *degree-based* heuristic has two versions depending on the order for the next edge executed. The first version starts processing vertices with the highest degree, choosing the next edge to be processed according to the degree of the destination vertex from highest to lowest degrees. The other version of this algorithm also starts processing vertices with the highest degree, but the next edge is selected on the criteria from lowest to highest degrees of the destination vertices.

Table §3.1 presents a preliminary evaluation of these heuristics on various datasets. If the partition interaction graph structure were to resemble these networks, we observe that our simple degree-based heuristics typically have 5-15% fewer partition load/unload operations than the sequential one, suggesting scope for improvement with better heuristics.

3.3 Concluding remarks

In this chapter we presented a preliminary design for scaling the KNN computation on a memory-constrained machine. Along with the algorithm's

design, we presented few heuristics that aim to reduce the number of data load/unload operations performed to process the set of partitions involved in the KNN computation. Our observation is that the order in which the partitions are processed impacts the number of load/unload operations. Our preliminary evaluation showed that a good heuristic can reduce the number of operations in a 5-15% with respect to those of a sequential heuristic. These results give us some space for improvement through better heuristics adapted to the intrinsic characteristics of the data.

Unfortunately, despite the fact that we observe that some heuristics are an efficient method to reduce the number of load/unload operations, we also remark few drawbacks on our initial design. Firstly, at phase 1, the algorithm sorts both the list of in-edges and out-edges of the graph. Such a sorting operation is costly, particularly for those large graphs that span billions of edges or more.

Secondly, our algorithm generates the set of tuples containing users' neighbors and neighbors' neighbors, which are written to disk in phase 2, incurring in $O(N(2K + K^2))$ operations, being N the number of vertices (users) in the graph, and K the number of neighbors of each vertex. Additionally, same number of operations are performed while the *partition interaction graph* is parsed on the phase 3 to read the tuples from disk. This set of tuples in disk is continuously accessed while the algorithm performs the profile comparisons to select the next tuple to process. Accordingly, for a large numbers of users in the system N, writing/reading operations performed on phases 2 and 3 respectively, adversely affect algorithm's scalability.

Finally, we observe that the hash table, populated in phase 2, is a useful tool to reduce the amount of duplicated tuples computed during the KNN computation. But, while the hash table presents few advantages over others data structures, it generates large memory footprints, particularly for large number of tuples, as is the normal case on processing large graphs. Besides, we observe that a hash table does not perform well for lookups, a fundamental operation on phase 2.

In Chapter §4, we present our main contribution, a scalable out-of-core algorithm for computing KNN, which addresses the main drawbacks of our preliminary design detailed above.

Chapter 4

Out-of-core KNN approach

In this chapter, we focus on the challenge of KNN computation over large datasets on a single commodity PC with limited memory. In this chapter, we leverage the experience gathered in our preliminary study (Chapter §3) to propose a novel approach to compute KNN on large datasets. Consequently, in this chapter we present *Pons* [24], a memory-efficient out-of-core algorithm for computing KNN on a single commodity PC. The main rationale of our approach is to minimize random accesses to disk, maximize sequential accesses to data and efficient usage of only the available memory.

We evaluate our approach on large datasets, in terms of performance and memory consumption. The evaluation shows that our approach requires only 7% of the time needed by an in-memory baseline to compute a KNN graph.

In the remainder of this chapter, we first discuss the motivations behind this work in Section §4.1. In Section §4.2 we introduce some preliminaries on the K-nearest neighbors problem, along with two basic approaches for computing KNN: in-memory approach (Section §4.2.1), and disk-based approach (Section §4.2.2). In Section §4.3 we describe in details our out-of-core approach for computing KNN on large datasets. Then, in Section §4.5 we describe the experimental setup used to evaluate *Pons* in Section §4.6. Finally, we present our conclusions in Section §4.7.

4.1 Introduction

Our first motivation for this work is derived from the fact that processing KNN efficiently on large datasets calls for in-memory solutions, this sort of

approach intends to store all data into memory for performing better in comparison to disk-based approaches. To do so, current datasets demand large memory, whose cost is not always affordable. Access to powerful machines is often limited, either by lack of resources for all users' needs, or by their complete absence.

The second motivation is that KNN computation has to be often performed offline, because it consumes significant resources. KNN algorithms usually cohabit on a given machine with other applications. Consequently, it is very seldom that it can enjoy the usage of the entire set of machine's resources, be it memory or CPU. For instance, *HyRec* [13], a hybrid recommender system, implements a KNN strategy to search similar users. *HyRec* devotes only a small fraction of its runtime and system resources for KNN computation. The rest is dedicated to recommendation tasks or system maintenance.

Finally, our last motivation comes from the fact that current graph frameworks [43, 63, 91] can efficiently compute well-known graph algorithms, processing large datasets in a short time. Those systems rely on the static nature of the data, i.e., data remaining the same for the entire period of computation. Unfortunately, to the best of our knowledge, they do not efficiently support some KNN fundamental operations such as neighborhood modification or neighbors' neighbors accesses. Typically they do not support any operation that modifies the graph itself [63, 91]. KNN's goal is precisely to change the graph topology.

Summarizing, our work is motivated by the fact that: (i) KNN is computationally expensive, (ii) KNN has to be mainly performed offline, and (iii) current graph processing frameworks do not support efficiently operations required for KNN computation.

We present *Pons*, an out-of-core algorithm for computing KNN on large datasets that do not completely fit in memory, leveraging efficiently both disk and the available memory. The main rationale of our approach is to minimize random accesses to disk, and to favor, as much as possible, sequential reading of large blocks of data from disk. Our main contributions of the paper are as follows:

- We propose *Pons*, an out-of-core approach for computing KNN on large datasets, using at most the available memory, and not the total amount required for a fully in-memory approach.

- *Pons* is designed to solve the non-trivial challenge of finding neighbors' neighbors of each entity during the KNN computation.

- Our experiments performed on large-scale datasets show that *Pons* computes KNN in only around 7% of the time required by an in-memory computation.
- *Pons* shows to be also capable of computing online, using only a limited fraction of the system's memory, freeing up resources for other tasks if needed.

4.2 Preliminaries

Given N entities with their profiles in a D-dimensional space, the *K-Nearest Neighbors* (KNN) algorithm aims to find the K-closest neighbors for each entity. The distance between any two entities is computed based on a given metric (as cosine similarity or Jaccard coefficient) that compares their profiles. A classic application of KNN includes finding the K-most similar users for any given user in a system such as IMDb, where a user's profile comprises of her preferences of various movies.

For computing the exact KNN it can be employed a *brute-force approach*, which has a time complexity of $O(N^2)$ profile comparisons being very inefficient for a large N. To address this concern, *approximate KNN* algorithms (KNN now onwards) adopt an iterative approach. At the first iteration ($t = 0$), each entity v chooses uniformly at random a set of K entities as its neighbors. Each subsequent iteration t proceeds as follows: each entity v selects K-closest neighbors among its candidate set, comprising its K current neighbors, its K^2 neighbors' neighbors, and K random entities [13]. At the end of iteration t, each entity's new K-closest neighbors are used in the computation for the next iteration $t + 1$. The algorithm ends when the average distance between each entity and its neighbors does not change considerably over several iterations.

The KNN state at each iteration t can be modeled by a directed graph $G^{(t)} = (V, E^{(t)})$, where V is a set of $N(= |V|)$ entities and $E^{(t)}$ represents edges between each entity and its neighbors. A directed edge $(u, v) \in E^{(t)}$ denotes (i) v is u's out-neighbor and (ii) u is v's in-neighbor. Let B_v denote the set of out-neighbors of the entity v. Furthermore, each entity v has exactly $K(= |B_v|)$ out-neighbors, while having any number (including 0 to $N - 1$) of in-neighbors. Also, we note that the total number of out-edges and in-edges in $G^{(t)}$ is NK.

Let F represent the set of profiles of all entities, and F_v denote the profile of entity v. In many scenarios in the fields of recommender systems and information retrieval, the profiles of entities are typically sparse. For instance, in IMDb, the number of movies an average user rates is significantly less than

the total number of movies, D, present in its database. In such a scenario, a user v's profile can be represented by a sparse vector F_v in a D-dimensional space ($|F_v| << D$). For the sake of simplicity, we consider each entity v's profile length to be utmost P ($\geq |F_v|$). In image classification and clustering systems, however, each entity v's profile (e.g., feature vector) is typically of high dimension in the sense that v's profile length is approximately $|F_v| \approx D$. With the above notation, we formally define the *average distance (AD)* for all entities and their respective neighbors at iteration t as:

$$AD^{(t)} = \frac{\sum_{u \in V} \sum_{v \in B_u} Dist(F_u, F_v)}{NK} \tag{4.1}$$

$Dist(F_u, F_v)$ measures the distance between the profiles of u and v. The KNN computation is considered converged when the difference between the average distances across iterations is minimal: $|AD^{(t+1)} - AD^{(t)}| < \epsilon$, for a small ϵ.

4.2.1 In-memory approach

A simple, yet efficient, way to implement KNN is using an *in-memory approach*, where all the data structures required during the entire period of computation are stored in memory. Algorithm §4 shows the pseudo-code for an in-memory implementation. Initially, the graph $G^{(0)}_{(mem)}$ and profiles F are loaded into memory from disk (lines §2-§3). At each iteration t, each vertex v selects K-closest neighbors from its candidate set C_v comprising its neighbors (B_v), its neighbors' neighbors ($\bigcup_{u \in B_v} B_u$), and a set of K random vertices ($Rnd(K)$). Closest neighbors of all vertices put together results in the graph $G^{(t+1)}_{(mem)}$, i.e., KNN graph of the next iteration.

In each iteration, every vertex performs up to $O(2K + K^2)$ profile comparisons. If a distance metric such as cosine similarity or Euclidean distance is used for profile comparisons, the overall time complexity for each iteration is $O(NP(2K + K^2))$. We note that the impact of heap updates (line §14) on overall time is little, since we are often interested in small values of $K(\approx 10 - 20)$ [13]. In terms of space complexity, this approach requires $O(N(2K + P))$ memory. Each of the KNN graphs of the current and the next iterations ($G^{(t)}_{(mem)}$, $G^{(t+1)}_{(mem)}$) consume $O(NK)$ memory, while the profiles consume $O(NP)$ memory. Although highly efficient, such an approach is feasible only when all data structures consume less than the memory limit of the machine.

Algorithm 4: In-memory KNN

Data: Graph file: File(G), Profiles file: File(F)
Result: Each vertex $v \in G$ finds its KNN.

```
1  begin
2      G^(0)_(mem) ← Read initial graph from File(G)
3      F_(mem) ← Read all profiles from File(F)
4      foreach Iteration t until convergence do
5          G^(t+1)_(mem) ← ϕ
6          foreach Vertex v ∈ G^(t)_(mem) do
7              Read B_v from G^(t)_(mem)
8              C_v ← B_v ∪ (⋃_{u∈B_v} B_u) ∪ Rnd(K)
9              TopK ← ϕ
10             Read F_v from F_(mem)
11             foreach Candidate w ∈ C_v do
12                 Read F_w from F_(mem)
13                 distValue ← Dist(F_v, F_w)
14                 UpdateHeap(TopK, w, distValue)
15             Insert(G^(t+1)_(mem), v, TopK)
```

4.2.2 Disk-based approach

In contrast to the above in-memory approach, the *disk-based approach* stores all the data –the two KNN graphs and the profiles– on disk and accesses small segments of this data at any instance. Algorithm §5 shows the pseudo-code for a disk-based implementation of the KNN algorithm. Each iteration t proceeds as follows. In order to form a candidate set C_v, a vertex v first obtains its out-neighbors B_v by reading the KNN graph $G^{(t)}_{(disk)}$ stored on disk (line §4), and then obtains each of its neighbors' neighbors by reading the disk again (lines §6-§8), and finally selects a set of K random vertices which does not require any disk operations. For profile comparisons, first the profile F_v is read from the profiles file (line §11) into memory. Vertex v's K closest neighbors are obtained by comparing its profile with each vertex w in the candidate set C_v whose profile F_w is read into memory from the profiles file one at a time (lines §12-§15). Vertex v's new K closest neighbors are written into the new iteration's KNN graph file (line §16).

Although the in-memory and disk-based approaches perform the same KNN computation, the way these approaches access data is significantly dif-

Algorithm 5: Disk-based KNN

Data: Graph file: File(G), Profiles file: File(F)
Result: Each vertex $v \in G$ finds its KNN.

```
1  begin
2      foreach Iteration t until convergence do
3          foreach Vertex v ∈ G^(t)_(disk) do
4              Read B_v from File(G^(t)_(disk))
5              Initialize candidate set C_v ← B_v
6              foreach Neighbor w ∈ B_v do
7                  Read B_w from File(G^(t)_(disk))
8                  C_v ← C_v ∪ B_w
9              C_v ← C_v ∪ Rnd(K)
10             TopK ← ϕ
11             Read F_v from File(F)
12             foreach Candidate w ∈ C_v do
13                 Read F_w from File(F)
14                 distValue ← Dist(F_v, F_w)
15                 UpdateHeap(TopK, w, distValue)
16             File(G^(t+1)_(disk)).Write(v, TopK)
```

ferent. The in-memory approach accesses all data–the two KNN graphs and the profiles–in the machine's main memory. In contrast, the disk-based approach accesses the same data via various disk operations such as random seeks, sequential reads, and writes, which are orders of magnitude slower in comparison to memory-based operations. On the upside, the disk-based approach consumes minimal memory with a space complexity of $O(K^2 + P)$. More specifically, a vertex v's candidate set C_v occupies upto $O(2K + K^2)$ memory, and the heap consumes $O(2K)$ memory, while only two profiles are loaded into memory at any instance thus consuming $O(2P)$ memory.

4.3 *Pons*

Thus the challenge of KNN computation can be essentially viewed as a trade-off between computational efficiency and memory consumption. Although efficient, an in-memory approach (Section §4.2.1) consumes a significant amount of memory. In contrast, a fully disk-based approach (Section §4.2.2) is very inefficient due to disk operations, albeit consuming little

memory. In this section, we propose *Pons*[†1], an out-of-core approach which aims to address this trade-off.

4.3.1 Overview

Pons is primarily designed to efficiently compute the KNN algorithm on a large set of vertices' profiles in a stand-alone memory-constrained machine. More specifically, given a large set of vertices' profiles and an upper bound of main-memory X_{limit}, that can be allocated for the KNN computation, *Pons* leverages this limited main memory as well as the machine's disk to perform KNN computation in an efficient manner.

The performance of *Pons* relies on its ability to divide all the data –KNN graph and vertices' profiles– into smaller segments such that the subsequent access to these data segments during the computation is highly efficient, while adhering to the limited memory constraint. *Pons* is designed following two fundamental principles: (i) *write once, read multiple times*, since KNN computation requires multiple lookups of various vertices' neighbors and profiles, and (ii) *make maximum usage of the data loaded into memory*, since disk operations are very expensive in terms of efficiency.

Algorithm 6: *Pons*

```
   Data: Graph file: File(G), Profiles file: File(F)
   Result: Each vertex v ∈ G finds its KNN.
1  begin
2      foreach Iteration t do
3          1. Partioning(GlobalOutEdges)
4          2. Create In-edge Partition Files
5          3. Create Out-edge Partition Files
6          4. Write Profile Partition Files
7          5. Compute Distances
8          Update(GlobalOutEdges)
```

We now present a brief overview of our approach, as illustrated in Algorithm §6, and Figure §4.1. *Pons* takes two input files containing vertices, their random out-neighbors, and their profiles. It performs the KNN computation iteratively as follows. The goal of each iteration t is to compute K-closest neighbors for each vertex. To do so, iteration t executes 5 phases (Algorithm §6, lines §2-§8 and Figure §4.1). First phase divides the vertices

[†1]The term 'pons' is Latin for 'bridge'.

into M partitions such that a single partition is assigned up to $\lceil N/M \rceil$ vertices. This phase parses the global out-edge file containing vertices and their out-neighbors and generates a K-out-neighborhood file for each partition.

We note here that the choice of the number of partitions (M) depends on factors such as the memory limit (X_{limit}), the number of vertices (N), the number of neighbors K, the vertices' profile length (P), and other auxiliary data structures that are instantiated. *Pons* is designed such that utmost (i) a heap of $O(\lceil N/M \rceil K)$ size with respect to a partition i, (ii) profiles of two partitions i and j consuming $O(\lceil N/M \rceil P)$ memory, (iii) other auxiliary data structures can be accommodated into memory all at the same time, while adhering to the memory limit (X_{limit}).

Based on the partitions created, phases 2, 3, and 4 generate various files corresponding to each partition. In the phase 5, these files enable efficient (i) finding of neighbors' neighbors of each vertex, and (ii) distance computation of the profiles of neighbors' neighbors with that of the vertex. The second phase uses each partition i's K-out-neighborhood file to generate i's in-edge partition files. Each partition i's in-edge files represent a set of vertices (which could belong to any partition) and their in-neighbors which belong to partition i. The third phase parses the global out-edge file to generate each partition j's out-edge partition files. Each partition j's out-edge files represent a set of vertices (which could belong to any partition) and their out-neighbors which belong to partition j. The fourth phase parses the global profile file to generate each partition's profile file.

The fifth phase aims to generate an output of a set of new K-closest neighbors for each vertex for the next iteration $t+1$. We recall that the next iteration's new K-closest neighbors is selected from a candidate set of vertices which includes neighbors, neighbors' neighbors, and a set of random vertices. While accessing each vertex's neighbors in the global out-edge file or generating a set of random vertices is straightforward, finding each vertex's neighbors' neighbors efficiently is non-trivial.

We now describe the main intuition behind *Pons*' mechanism for finding a vertex's neighbors' neighbors. By comparing i's in-edge partition file with j's out-edge partition file, *Pons* identifies the common 'bridge' vertices between these partitions i and j. A bridge vertex b indicates that there exists a source vertex s belonging to partition i having an out-edge (s, b) to the bridge vertex b, and there exists a destination vertex d belonging to partition j having an in-edge (b, d) from the bridge vertex b. Here b is in essence a bridge between s and d, thus enabling s to find its neighbor b's neighbor d. Using this approach for each pair of partitions i and j, the distance of a vertex and each of its neighbors' neighbors can be computed.

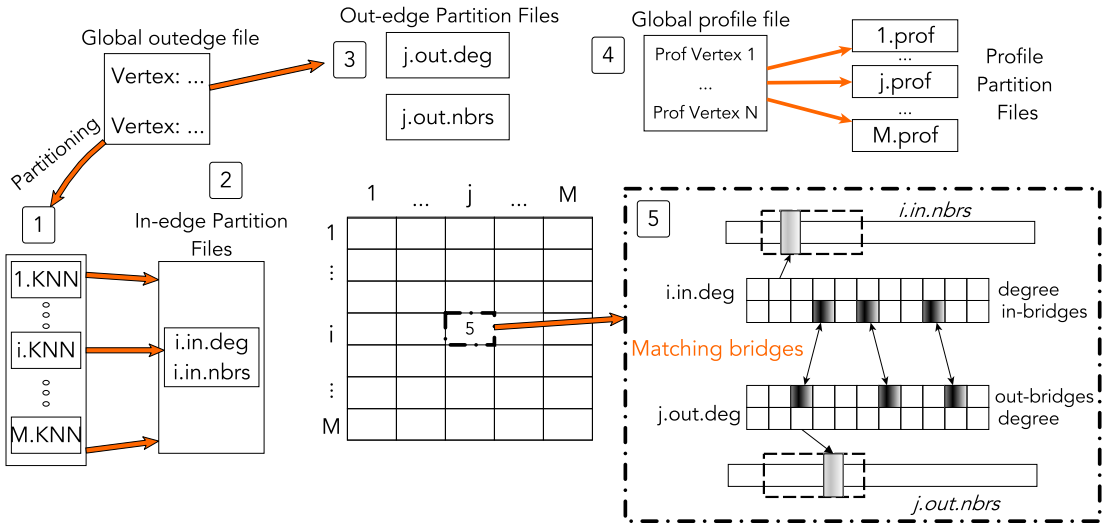


Figure 4.1: *Pons executes 5 phases: (1) Partitioning, (2) In-Edge Partition Files, (3) Out-Edge Partition Files, (4) Profile Partition Files, and (5) Distance Computation.*

As *Pons* is designed to accommodate the profiles of only two partitions at a time in memory, *Pons* adopts the following approach for each partition i. First, it loads into memory i's profile as well as the bridge vertices of i's in-edge partition file. Next, an empty heap is allocated for each vertex which is assigned to partition i. A vertex s' heap is used to accommodate utmost K-closest neighbors. For each partition j, the common bridge vertices with i are identified and subsequently all the relevant pairs (s, d) are generated with s and d belonging to i and j respectively, as discussed above. For each generated pair (s, d), the distance between the source vertex s and the destination vertex d are computed, and then the heap corresponding to the source vertex s is updated with the distance score and the destination vertex d. Once all the partitions $j = [1, M]$ are processed, the heaps of each vertex s belonging to partition i would effectively have the new K-closest neighbors, which are written to the next iteration's global out-edge file. Once all the partitions $i = [1, M]$ are processed, *Pons* moves on to the next iteration $t + 1$.

4.3.1.1 Example

Figure §4.2 shows an example graph containing $N = 6$ nodes and $M = 3$ partitions. Let vertices A and T be assigned to partition 1 (red), U and C

to partition 2 (blue), and *W* and I to partition 3 (green). Figure §4.3 shows various in-edge and out-edge partition files corresponding to their respective partitions. For instance, in the `1.in.nbrs` file, U and *W* (denoted by dotted circles) can be considered as bridge vertices with *A* (bold red), which belongs to partition 1, as the in-neighbor for both of them.

To generate *A*'s neighbors' neighbors, `1.in.nbrs` is compared with each partition j's out-edge file `j.out.nbrs`. For instance, if `1.in.nbrs` is compared with `3.out.nbrs`, 2 common bridge vertices U and *W* are found. This implies that U and *W* can facilitate in finding *A*'s neighbors' neighbors which belong to partition 3. As shown in Figure §4.4, vertex *A* finds its neighbors' neighbor I, via bridge vertices U and *W*.

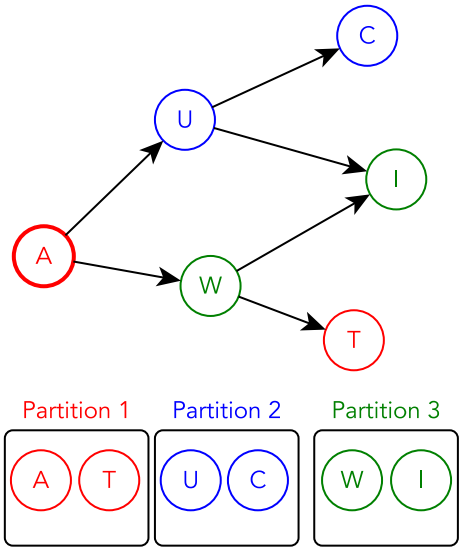


Figure 4.2: *Example graph. A's out-neighbors and A's neighbors' neighbors.*

4.4 KNN iteration

At iteration t, *Pons* takes two input files: *global out-edge file* containing the KNN graph $G^{(t)}$, and *global profile file* containing the set of vertices' profiles. Global out-edge file stores contiguously each vertex id v along with its K initial out-neighbors' ids. Vertex ids range from 0 to $N-1$. The global profile file stores contiguously each vertex id and all the P items of its profile. These

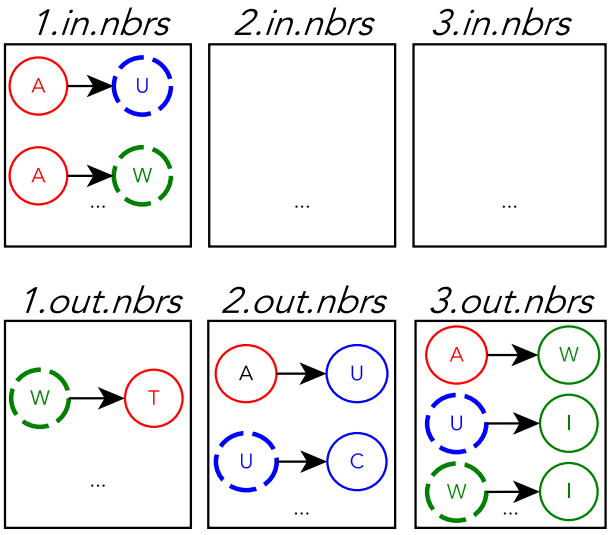


Figure 4.3: *In-edge partition files and out-edge partition files.*

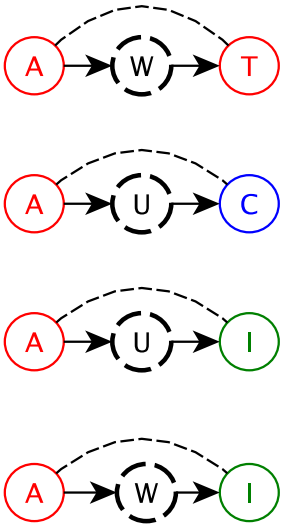


Figure 4.4: *A's neighbors' neighbors found using bridge vertices.*

files are in binary format which helps in better I/O performance (particularly for random lookups) as well as saves storage space.

4.4.1 Phase 1: Partitioning

The memory constraint of the system limits the loading of the whole graph as well as the profiles into memory. To address this issue, we divide these data structures into M partitions, each corresponding to roughly $\lceil N/M \rceil$ distinct vertices, such that the profiles of utmost two partitions ($O(\lceil N/M \rceil P)$) and a K-neighborhood heap of one partition ($O(\lceil N/M \rceil K)$) can be accommodated into memory at any instance.

We adapt the one-pass streaming graph partitioning approach proposed in [1]. More specifically, at an iteration t, we perform a single pass on the KNN graph file $G^{(t)}$ as follows. Each vertex $v \in G^{(t)}$ is assigned as a *master replica* to the partition that shares most vertices considering its id v and its neighbors' ids B_v. When v is assigned as a master replica to a partition j, all its neighbors B_v are assigned as *slave replicas* to the partition j. Formally, the vertex partitioning objective can be defined as:

$$\min \sum_{i \neq j} |W_i \cap R_j| \text{ s.t. } |W_i| \leq \lceil N/M \rceil \tag{4.2}$$

where W_i represents the set of vertices assigned as master to the partition i and R_j represents the set of vertices assigned as either master or slave to the partition j. The partitioning follows the load balancing constraint of allocating utmost $\lceil N/M \rceil$ master replicas per partition.

For efficient partitioning, we use a vector of N booleans $\texttt{Bool_Vec}(j)$ for each partition j. A bit v set to 1 in $\texttt{Bool_Vec}(j)$ means that either vertex v is assigned as a master or a slave replica to partition j. For an unassigned vertex v, the algorithm searches all available partitions $\{j \text{ s.t. } |W_j| < \lceil N/M \rceil\}$, measuring number of common (master or slave) replicas between v and its out-neighbors ($v \cup B_v$) and itself, by checking the corresponding set bits in the vector $\texttt{Bool_Vec}(j)$. The available partition j with the maximum overlap is selected as the master partition for the vertex v. The vertex v and its out-neighbors B_v are assigned as master and slave replicas respectively to the partition j, along with setting their corresponding bits in the vector $\texttt{Bool_Vec}(j)$.

Figure §4.5 shows an example of our partitioning algorithm, where master and slave replicas are depicted with bold and dotted circles respectively. Let A be an unassigned vertex with its out-neighbors B and C, which needs to be assigned to one of the available partitions. In this example, partition i shares two common replicas (B and C) with A and its neighbors. On the other hand, partition j shares only one element (C) with A and its neighbors. Finally, vertex A is assigned as a master to i.

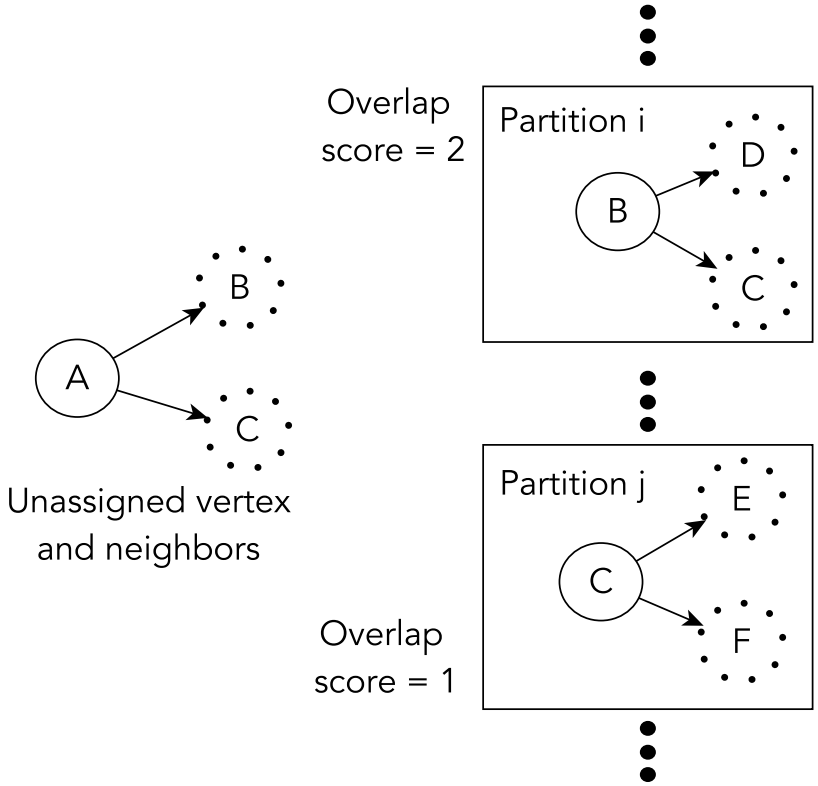


Figure 4.5: *Partitioning example.*

When a vertex v is assigned as a master replica to partition j, the vertex v and its out-neighbors B_v are written to j's K-out-neighborhood file `j.knn` that contains all the master replicas of the partition j and their respective out-neighbors.

4.4.2 Phase 2: In-edge partition files

This phase takes each partition i's K-out-neighborhood file `i.knn` as input and generates two output files representing bridge vertices and their in-neighbors. For a vertex v assigned as a master replica to partition i, each of its out-neighbors $w \in B_v$ is regarded as a 'bridge vertex' to its in-neighbor v in this phase. We note here that the master replica of a bridge vertex $w \in B_v$ could belong to any partition.

The first file `i.in.deg` stores a list of (i) all bridge vertices b whose master replica could belong to any partition, and (ii) the number of b's in-neighbors

whose master replicas belong to partition i. This list is sorted by the id of each bridge vertex b. The second file `i.in.nbrs` stores the ids of the in-neighbors of each bridge vertex b stored contiguously according to the bridge vertices' sorted ids in the `i.in.deg` file.

To efficiently convert the `i.knn` file into `i.in.deg` and `i.in.nbrs` files, we use the following approach. We first allocate in-memory (i) *bridge buffer* of $O(N)$ memory for bridge vertices, and (ii) *in-neighbor buffer* of $O(\lceil N/M \rceil K)$ memory for their in-neighbors. For each vertex v and its out-neighbors B_v in the `i.knn` file, we note that the master replica of v belongs to the partition i while each of its out-neighbors $w \in B_v$ could belong to any partition. We then perform two passes on the `i.knn` file. Firstly, we populate the bridge buffer with each bridge vertex b observed in `i.knn` and the number of vertices that have out-edges to b. In the second pass, we populate the in-neighbor buffer with each bridge vertex's in-neighbor at its corresponding position. Finally, the bridge buffer and the in-neighbor buffer are written to `i.in.deg` and `i.in.nbrs` respectively.

4.4.3 Phase 3: Out-edge partition files

This phase takes the global out-edge file as input and generates two output files per partition representing bridge vertices and their out-neighbors, similar to the previous phase. For each partition j, the first file `j.out.deg` stores a list of (i) all bridge vertices b whose master replica could belong to any partition, and (ii) the number of b's out-neighbors whose master replicas belong to partition j. This list is sorted by the id of each bridge vertex b. The second file `j.out.nbrs` stores the ids of the out-neighbors of each bridge vertex b stored contiguously according to the bridge vertices' sorted ids in the `j.out.deg` file. These files are used in the Phase 5 (in Section §4.4.5) for the KNN computation.

To efficiently convert the global out-edge file into `j.out.deg` and `j.out.nbrs` files, we leverage the format of the global out-edge file which stores vertices' ids in a sorted fashion. For each vertex v, we divide its out-neighbors B_v into M non-overlapping sets according to their respective vertices' ($w \in B_v$) master partitions. With v as a bridge vertex, each non-overlapping set corresponding to partition j is written to their respective files: the number of vertices in `j.out.deg`, and the vertices themselves in `j.out.nbrs`.

4.4.4 Phase 4: Profile partition files

This phase takes the global profile file and generates M profile partition files as output. Each vertex v's profile F_v is read from the global profile file, and then written to the profile partition file corresponding to the partition of its master replica. At the end, each profile partition file `j.prof` consumes upto $O(\lceil N/M \rceil P)$ memory or disk space. Each profile partition file subsequently allows the fast loading of the profiles of all master vertices in its partition in the Phase 5, as it facilitates sequential reading of the entire file without any random disk operations.

4.4.5 Phase 5: Distance computation

This phase uses each partition's in-edge, out-edge, and partition profile files to compute the distances between each vertex and a collection of its neighbors, neighbors' neighbors, and random vertices, generating the set of new K-closest neighbors for the next iteration.

Algorithm §7 shows the pseudo-code for this phase. Distance computation is performed at the granularity of a partition, processing sequentially each one from 1 to M (line §2-§25). Once a partition i is completely processed, each vertex $v \in W_i$ assigned to i has a set of new K-closest neighbors.

The processing of partition i primarily employs four in-memory data structures: InProf, InBrid, HeapTopK, and tuple T. First, InProf stores the profiles of vertices (W_i) in partition i read from the `i.prof` file (line §3). Second, InBrid stores the bridge vertices and their corresponding number of in-neighbors in partition i read from the `i.in.deg` file (line §4). Third, HeapTopK is a heap, which is initially empty (line §5), then it stores the scores and ids of the K-closest neighbors for each vertex $v \in W_i$. Finally, tuple T stores neighbors, neighbors' neighbors, and random neighbors' tuples for distance computation.

For computing the new KNN for each vertex $s \in W_i$, all partitions starting from 1 to M are parsed one at a time (lines §6-§25) as follows. For a partition j, its profile file `j.prof` and its out-edge bridge file `j.out.deg` are read into two in-memory data structures OutProf and OutBrid, respectively (lines §7-§8). Similar to i's in-memory data structures, OutProf stores the profiles of vertices (W_j) in partition j, and OutBrid stores the bridge vertices and their corresponding number of out-neighbors in partition j. By identifying a set of common bridge vertices between InBrid and OutBrid, we generate in paral-

lel, all ordered tuples of neighbors' neighbors as follows:

$$(s, d) \mid s \in W_i, d \in W_j, (s, b) \in E^{(t)}, (b, d) \in E^{(t)}, b \in (InBrid \cap OutBrid) \quad (4.3)$$

Here, each ordered tuple (s, d) represents a source vertex $s \in W_i$ and a destination vertex $d \in W_j$, with an out-edge (s, b) from s and an-inedge (b, d) to a bridge vertex b that is common to both $InBrid$ and $OutBrid$. We also generate in parallel, all ordered tuples of each vertex $s \in W_i$ and its immediate neighbors $(w \mid w \in B_v \cap W_j)$ which belong to the partition j. A distance metric such as cosine similarity or euclidean distance is then used to compute the distance score ($Dist(F_s, F_d)$) between each ordered tuple's source vertex s and destination vertex d. The top-K heap ($HeapTopK[s]$) corresponding to the source vertex s is updated with the destination vertex d's id and the computed distance score ($Dist(F_s, F_d)$).

4.5 Experimental setup

In this section, we describe the experimental setup used to show how *Pons* achieves the proposed goals.

4.5.1 Machine

We perform our experiments on a Apple MacBook Pro laptop, with an Intel Core i7 processor (Cache 2: 256 KB, Cache 3: 6 MB) of 4 cores, 16 GB of RAM (DDR3, 1600 MHz) and a 500 GB (6 Gb/s) solid state drive (SSD).

4.5.2 Datasets

We evaluate *Pons* on both sparse- and dense- dimensional datasets. For sparse datasets, we use Friendster [69] and Twitter [15][†2] traces. Both in Friendster and Twitter, vertices represent users, and profiles are their lists of friends in the social network.

For dense datasets, we use a large computer vision dataset (ANN-SIFT-100M) [48] which has vectors of 128 dimensions each. In this dataset, vertices represent high-dimensional vectors and their profiles represent SIFT descriptors. The SIFT descriptors are typically high dimensional feature vectors used

[†2]Twitter dataset available on: http://konect.uni-koblenz.de/networks/twitter_mpi

Algorithm 7: NNComputation(): Neighbors' neighbors computation

```
Data: In-edge partition files, Out-edge partition files, Profiles F
Result: New K-nearest neighbors for each vertex
1  begin
2      foreach (In-edge) Partition i do
3          Read InProf from File(i.prof)
4          Read InBrid from File(i.in.deg)
5          HeapTopK[W_i] ← ϕ
6          foreach (Out-edge) Partition j do
7              Read OutProf from File(j.prof)
8              Read OutBrid ← from File(j.out.deg)
9              Initialize tuple T ← ϕ
10             CndBrid ← (InBrid ∩ OutBrid) ∪ (W_i ∩ OutBrid)
11             foreach Bridge b ∈ CndBrid do
12                 in parallel
13                 Src ← ReadInNeig(i.in.nbrs, b)
14                 Dst ← ReadOutNeig(j.out.nbrs, b)
15                 AddTuples(T, Src × Dst)
16             foreach (s, d) ∈ T do
17                 in parallel
18                 dist ← Dist(F_s, F_d)
19                 UpdateHeap(HeapTopK[s], d, dist)
20             foreach s ∈ W_i do
21                 in parallel
22                 Dst ← Rnd(K) ∈ W_j
23                 Compute tuples s × Dst
24                 Update HeapTopK[s] as above
25         File(G^(t+1)).Write(HeapTopK)
```

in identifying objects in computer vision. In our experiments, we use subsets of 30 and 50 millions vectors from this dataset.

4.5.3 Evaluation metrics

We measure the performance of *Pons* in terms of execution time and memory consumption. Execution time is the (wall clock) time required for completing a defined number of KNN iterations. Memory consumption is mea-

Dataset	Vertices	P	K	Virtual Mem. [Gb]
ANN-SIFT 30M (30M)	30M	128	10	19.35
ANN-SIFT 50M (50M)	50M	128	10	30.88
Friendster (FRI)	38M	124	10	23.26
Twitter (TWI)	44M	80	10	19.43

Table 4.1: *Datasets.*

sured by the maximum memory footprint observed during the execution of the algorithm. Thus, we use maximum resident set size (RSS) of the program for measuring its peak memory consumption during the entire computation, along with this virtual memory size (VM).

4.6 Evaluation

We evaluate the performance of *Pons* on large datasets that do not fit in memory (Table §4.1 shows the size of the memory required to load the whole dataset). We compare our results with a fully in-memory implementation of the KNN algorithm (INM). We show that our solution is able to compute KNN on large datasets using only the available memory, regardless of the size of the data.

4.6.1 Performance

We evaluate *Pons* on both sparse and dense datasets. We ran one iteration of KNN both on *Pons* and on INM. We divide the vertex set on M partitions (detailed in Table §4.2), respecting the maximum available memory of the machine. For this experiment both approaches run on 8 threads.

4.6.1.1 Execution time

In Table §4.2 we present the percentage of execution time consumed by *Pons* compared to INM's execution time for various datasets. *Pons* performs the computation in only a small percentage of the time required by INM for

the same computation. For instance, *Pons* computes KNN on the Twitter dataset in 8.27% of the time used by INM. Similar values are observed on other datasets. These results are explained by the capacity of *Pons* to use only the available memory of the machine, regardless of the size of the dataset. On the other hand, an in-memory implementation of KNN needs to store the whole dataset in memory for achieving good performance. As the data does not fit in memory, the process often incurs swapping, performing poorly compared to *Pons*.

		Exec. Time	**RSS[GB]**		**Virtual Mem.[GB]**	
Dataset	M	**Pons/INM** %	**Pons**	**INM**	**Pons**	**INM**
Friendster (FRI)	5	6.95	11.23	12.79	16.86	23.26
Twitter (TWI)	4	8.27	13.04	13.78	15.55	19.43
ANN 50M (50M)	9	4.34	12.77	13.16	15.48	30.88

Table 4.2: *Relative performance Pons/INM, and memory footprint.*

4.6.1.2 Memory footprint

As we show in Table §4.2, our approach allocates at most the available memory of the machine. However, INM runs out of memory, requiring more than 23 GB in the case of Friendster. As a result, an in-memory KNN computation might not be able to efficiently accomplish the task.

4.6.2 Multithreading performance

We evaluate the performance of *Pons* and INM, in terms of execution time, on different number of threads. The memory consumption is not presented because the memory footprint is almost not impacted by the number of threads, only few small data structures are created for supporting the parallel processing.

Figure §4.6 shows the execution time of one KNN iteration on both approaches. The results confirm the capability of *Pons* to leverage multithreading to obtain better performance. Although the values do not show perfect scalability, results clearly show that *Pons*' performance increases with the number of threads. The fact that is not a linear increase is due to that some

phases do not run in parallel, mainly due to the nature of the computation, requiring multiple areas of coordination that would affect the overall performance.

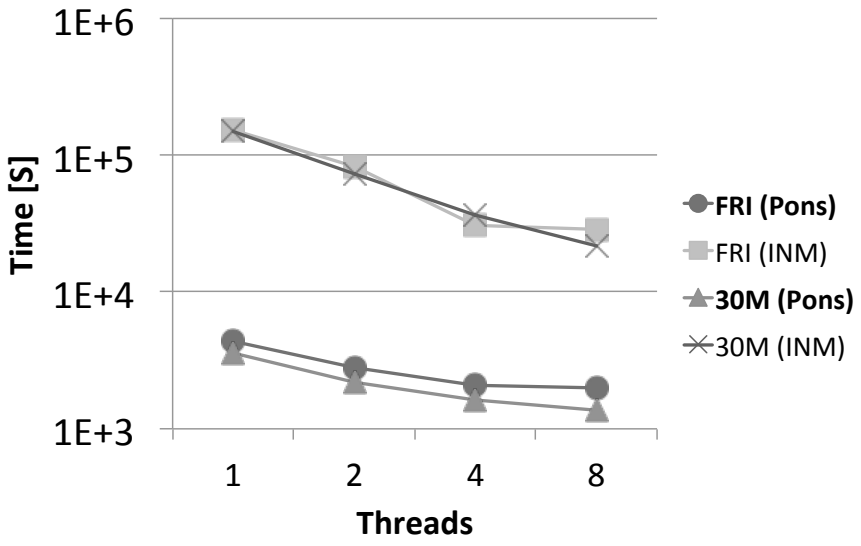


Figure 4.6: *Impact of multithreading on* Pons.

4.6.3 Performance on different memory availability

One of the motivation of this work is to find an efficient way of computing KNN online, specifically considering contexts where not all resources are available for this task. KNN computation is often just one of the layers of a larger system, therefore online computation might only afford a fraction of the resources. In this regard, we evaluate *Pons*' capacity of performing well when only a fraction of the memory is available for the computation. Figure §4.7 shows the percentage of execution time taken by *Pons* compared to INM, for computing KNN running on a memory-constrained machine.

If only 20% of the memory is allocated to KNN, *Pons* requires only 12% of the execution time taken by INM on a dense dataset. In the case of a sparse dataset, *Pons* computes KNN in only 20% of the time taken by INM, when the memory is constrained to 20% of the total. On the other hand, when 80% of the memory is available for KNN, *Pons* requires only 4%, and 8% of the INM execution time, on dense and sparse data set, respectively. These results show the ability of *Pons* of leveraging only a fraction of the memory for computing

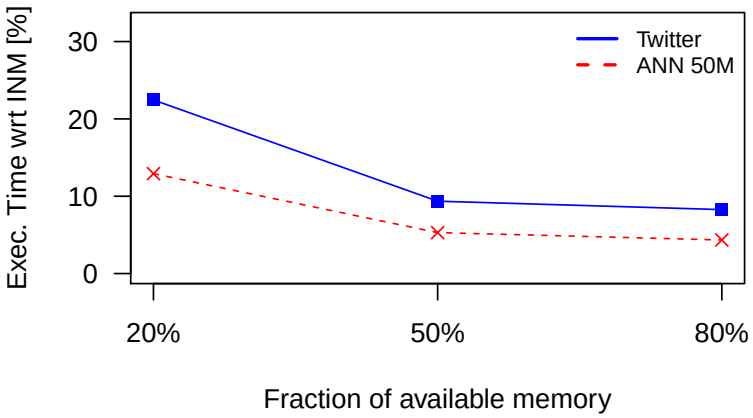


Figure 4.7: *Impact of the available memory on* Pons.

KNN, regardless of the size of data. Therefore, *Pons* lends itself to perform online KNN computation using only available resources, leaving the rest free for other processes.

4.6.4 Evaluating the number of partitions

Pons' capability to compute KNN efficiently only using the available memory relies on the appropriate choice of the number of partitions M. Larger values of M decrease the memory footprint, diminishing likewise algorithm's performance, this is due to the increase in the number of I/O operations. On the other hand, smaller values of M increase the memory footprint, but also decrease performance caused by the usage of virtual memory and consequently expensive swapping operations. An appropriate value of M allows *Pons* to achieve better performance.

4.6.4.1 Execution time

We evaluate the performance of *Pons* for different number of partitions. Figures §4.8 and §4.9 show the runtime for the optimal value, and two suboptimal values of M. The smaller suboptimal value of M causes larger runtimes

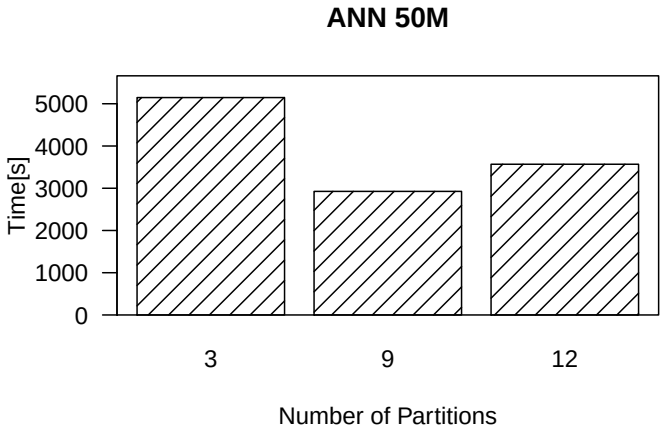


Figure 4.8: *Runtime: The impact of the number of partitions* M*. ANN-SIFT 50M dataset.*

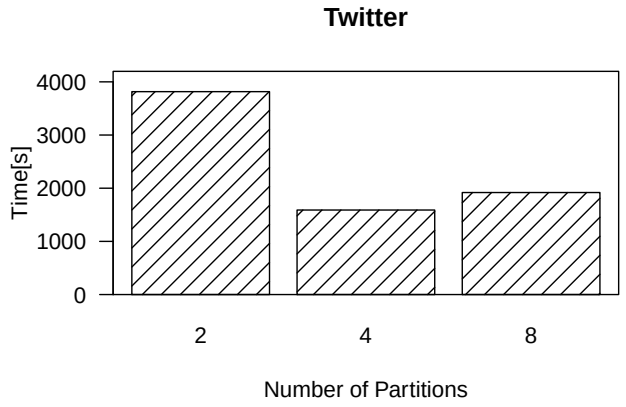


Figure 4.9: *Runtime: The impact of the number of partitions* M*. Twitter dataset.*

due to the fact that the machine runs out of memory, allocating virtual memory for completing the task. Although runtime increases, it remains lower than INM runtime (roughly 7% of INM runtime). Larger suboptimal value of M affects performance as well, by allocating less memory than it is available, thus misspending resources in cases of full availability.

4.6.4.2 Memory footprint

Figure §4.10 and §4.11 show the memory footprint for the optimal value of M, and two suboptimal values. In both cases, smaller values of M increase RSS, reaching the maximum available, unfortunately, virtual memory footprint increase as well, affecting the performance. The optimal value of M increases RSS to almost 16 GB, but virtual memory consumption remains low, allowing much of the task being performed in memory. On the other hand, a larger value of M decreases both RSS and the virtual memory footprint, performing sub optimally. Although, larger values of M affect performance, this fact allows our algorithm to perform KNN computation on machines that do not have all resources available for this task, regardless the size of the data.

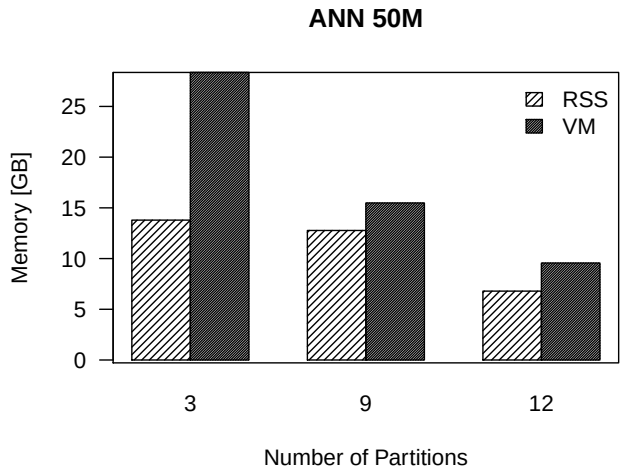


Figure 4.10: *Memory footprint: The impact of* M. *ANN-SIFT 50M dataset.*

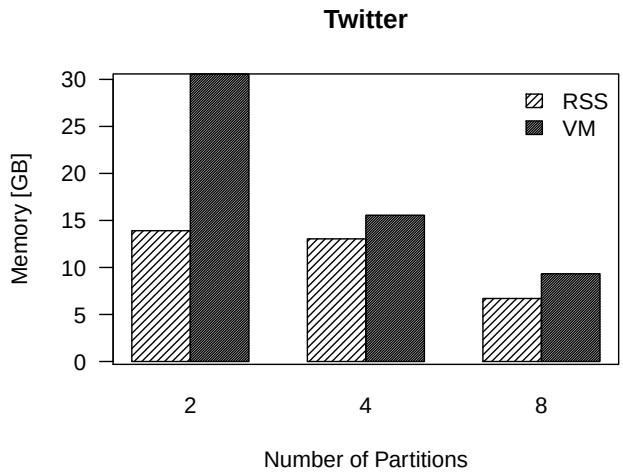


Figure 4.11: *Memory footprint: The impact of M. Twitter dataset.*

4.7 Conclusions

In this chapter we proposed *Pons*, an out-of-core algorithm for computing KNN on large datasets, leveraging efficiently both disk and the available memory. *Pons*' performance relies on its ability to partition a KNN graph and profiles into smaller chunks such that the subsequent accesses to these data segments during the computation is highly efficient, while adhering to the limited memory constraint.

We evaluate *Pons*' performance in terms of execution time and memory consumption compared with a fully in-memory algorithm. We demonstrated that *Pons* is able to compute KNN on large datasets, using only the memory available. *Pons* outperforms an in-memory baseline, computing KNN on roughly 7% of the in-memory's time, using efficiently the available memory. Our evaluation showed *Pons*' capability for computing KNN on machines with memory constraints, being also a good solution for computing KNN online, devoting few resources to this specific task.

Part III

Updating User Profiles

Chapter 5

Updating profiles in KNN algorithms

In this chapter, we focus on the challenge of computing KNN on data that changes continuously and rapidly over time. As we have mentioned previously, computing KNN is a memory intensive operation, which is aggravated with the increasing size of the datasets. Doing these computations on dynamic datasets will only increase the memory bottleneck and consequently the runtime.

In this chapter, we present *UpKNN* [53]: a *scalable* and *memory-efficient*, *thread-based* approach to take the updates on a dataset into account and still compute the KNN efficiently, keeping a check on the wall-time. Our contribution *UpKNN* processes millions of updates in real-time, running on a single commodity PC.

Our extensive experiments, performed on both dense and sparse datasets, confirm the scalability of *UpKNN*, both in number of updates processed and the threads used in the computation. *UpKNN* achieves speedups ranging from *13.64X* to *49.5X* in the processing of millions of updates, with respect to the performance of a non-partitioned baseline. These results have been achieved by performing roughly 1% of the disk operations performed by the baseline. Experiments also show that *UpKNN* is able to process an average of 3.2 millions updates per second, making our approach a good solution for online KNN processing.

The remainder of this chapter is organized as follows. Section §5.1 introduces this work. Section §5.2 provides an intuition and background, giving the notations to formally describe the problem addressed in this work. Section §5.3 presents our contribution *UpKNN* in details. In addition we also

show how our solution is implemented on top of *Pons* [24], our out-of-core approach for computing KNN on *static* profiles. Section §5.4 describes the experimental setup used to evaluate our approach. The evaluation and analysis of the results obtained are presented in Section §5.5. Finally, Section §5.6 summarizes our work in *UpKNN*.

5.1 Introduction

As we have already discussed, some of the applications involving the use of KNN witness changes in profiles over time, but the principle of KNN, computing similarities between pairs of users, makes it very difficult to take these changes into account.

Previous works [14, 32] have shown that the majority of the computation time involved in the whole process is spent on computing these similarity values. For instance, a brute-force approach for computing the *K-nearest neighbor* graph of a N entities system has a complexity of $O(N^2)$. Any kind of additional updates in the profiles of users will only mean more of such similarity computations, thereby increasing the computation time considerably, making the algorithm less and less *scalable*.

Due to this cost, many current state-of-the-art approaches [14, 23, 24, 32] simplify the processing assuming the dataset remains static throughout the computation. Unfortunately, performing KNN computation on static data brings a major downside. As we know, nowadays data changes continuously at unimaginable rates, specially on those web-based, social networks or recommendation systems' applications. Consequently, the computation of KNN on static datasets does not consider data's dynamism, relying on content that is always outdated. Although there are some current approaches [11, 13] that re-compute the KNN periodically to consider changes in profiles, to the best of our knowledge, there are no works updating profiles throughout the KNN computation.

Hence, in this chapter, we focus on this particular aspect of KNN, i.e., accounting for the changes in user preferences. As a result we propose *UpKNN*, a multithreading approach for processing real-time updates in KNN algorithms. The main novelty in our contribution is the use of a set of *parallel computing bricks* to design and code an efficient approach to address the challenges in this specific problem.

UpKNN is designed to avoid random accesses to disk, applying updates in profiles all together in parallel, thereby achieving a better performance. In

UpKNN, we move away from traditional random access approaches towards a more efficient partition-based idea. Instead of directing a stream of updates directly towards users, we propose to partition the updates, based on the existing partition based KNN approach (such as in *Pons* [24]).

UpKNN is designed to perform well on a single commodity PC, through an efficient out-of-core approach that leverages disk and main memory efficiently. The most recent works [43, 63, 91, 119, 132] have shown the out-of-core algorithms as an efficient, inexpensive and more accessible way of implementing complex algorithms on single machines. Although distributed algorithms exhibit good scalability for processing large datasets, we observe that designing, coding and debugging these algorithms is more complex than those running on single machines.

This partition of the update stream is carefully designed in two phases, employing a multithreading approach. While the first phase, *classify*, reads from the update stream to classify the user-item tuples read, into their corresponding partitions, the second phase, *merge*, is responsible for adding these updates in the existing user-profiles, stored in disk. Such an idea of directing the updates avoids the random updates all together, thereby achieving a better performance in computation time.

We perform extensive experiments to highlight this performance against a baseline approach - where the updates are applied from a non-partitioned set of data, using a multithreading programming model also. Our experiments show that *UpKNN* scales in the number of updates processed, being able to process 100 millions of updates in roughly 40 seconds on a commodity PC. Besides, *UpKNN* scales linearly with respect to the number of threads devoted to the computation.

In summary, our main contributions of the paper are as follows:

- We propose *UpKNN*, an efficient multithreading approach that addresses the challenge of performing real-time updates on KNN profiles.
- *UpKNN* reduces the number of disk operations performed during the computation, favoring the reading and writing of large chunks of data from disk.
- Our carefully designed multithreading approach leverages the use of *two-layer in-memory* buffers to reduce synchronization between threads and concurrency issues in I/O operations.
- *UpKNN* implements a two-phase approach that overlaps I/O requests and computations to improve the performance by keeping the CPU busy,

running several independent tasks together.

5.2 Background

The applications served by the well-known KNN algorithm often has temporal user-profiles, i.e., the profiles of users that change and evolve over time. For the sake of clarity, we focus on user-based KNN, where users add new items to their profiles. For example, in a recommendation system for a movies website, users will view/rate more movies over time either as the new movies are released or by simply continuing to watch movies. Therefore, in systems with changing user-profiles, a KNN also needs to account for these changes. The challenge in considering the updates of the user-profiles is that it will considerably increase the time taken for the KNN computations i.e., the run-time of the process) and in addition to it, will make the memory requirements of KNN surge. These challenges become more significant when we aim to use a single commodity PC. With the growing number of updates in the stream, this process continues to become increasingly challenging.

As a result, we propose *UpKNN*, a novel way of integrating a stream of user updates to compute the KNN in an online fashion, promising high scalability in terms of the number of updates that can be performed without compromising on the evaluation metrics. To give a detailed description of the *UpKNN* algorithm in Section §5.3, we first lay a background of notations and assumptions in Section §5.2.1. Once we have defined the data-structures and representations, we continue by giving a formal definition of the problem in Section §5.2.2.

5.2.1 Notations and assumptions

Let us consider a set of entities $V = \{v_1, v_2, ..., v_N\}$, $|V| = N$, associated with a set items denoted by $I = \{i_1, i_2, ..., i_Y\}$, $|I| = Y$. *UpKNN* assumes that the underlying KNN approach partitions the N entities into M partitions and the size of each such partition is denoted by SP, where $SP = |N/M|$. (M is a system parameter for *UpKNN*). With the aim of using a single machine for the computations, we divide the large datasets, that otherwise cannot be completely loaded into the memory, into M partitions. We chose the number of partitions such that at least one partition can be fully loaded and processed in memory at a time. As the partitioning algorithm goes beyond the focus of this work, we simply assume a random partitioning of the N entities.

Each entity v has associated a profile F_v, which is an array of items associated to her. Corresponding to each of the m partitions there is a partition file PF_m stored in disk, which stores the profiles F of all the entities belonging to partition m:

$$PF_m \equiv \{F_u \mid u \in m\}, \forall m \in M$$

To update the set of profiles, *UpKNN* receives an unsorted stream/set of updates S consisting of entity-item pairs:

$$S \equiv \{< v, i > \mid v \in V, i \in I\}$$

As we have pointed out, one of the driving factors behind the performance of *UpKNN* is the carefully designed partition scheme. As a result, *UpKNN*, relying on a *multithreading* partition based approach, reads the updates to classify and write them into M *update files*, UF_m.

5.2.2 Problem definition

The principal objective of this work is to consider the temporal changes in entities' profiles while constructing the KNN graph. Due to the response time requirements of the applications using KNN, it is of utmost importance that these changes are integrated and computed in real-time. Like any other algorithm, we are bound by memory and resources availability. Keeping these constraints in mind, we target to perform the updates on profiles and compute the KNN on a *single commodity PC*, maximizing the number of updates that *UpKNN* can sustain.

Therefore, given a stream/set of updates S, for an underlying graph with entities V associated to an item set I, *UpKNN*'s goal is to compute the KNN graph considering S in real-time, on a single commodity PC, minimizing algorithm's computational time.

5.3 *UpKNN* algorithm

As we have pointed out earlier, *UpKNN* targets to compute KNN on a single commodity PC, while sustaining a large number of real-time updates on KNN profiles. We propose to achieve this by using a two phase approach.

Our approach is based on a simple observation: the *read* operations from disk are costly in time. Intuitively, the more read operations, the larger is the time required to perform them. With *UpKNN*, we propose to parallelize these read operations and other computations to perform the updates on the profiles.

UpKNN carefully employs threads and buffers to perform the disk read/write operations efficiently, which results in impressively efficient KNN computations in terms of the wall-time, even while considering millions of updates on the profiles.

5.3.1 Classify-Merge phases

In brief, *UpKNN* is designed around two phases, namely: *Classify* and *Merge*. As the name suggests, these two phases, *classify* and *merge*, are responsible for the classification of the updates, explained shortly, and integrating them to the existing profiles respectively.

The goal of the *classify* phase is to read the entity-item tuples from the set/stream of updates S, and put them in their corresponding m update files UF_m. Following this phase, *merge* takes the UF_m files as input to produce updated profiles, i.e., it *merges* the items of PF_m with the items of the corresponding UF_m and writes the contents back to the PF_m.

5.3.1.1 Classify

The idea behind this phase is to group the updates, that will be performed on a similarly grouped data in an attempt to minimize the number of expensive read/write operations on disk.

In a nutshell, this phase is responsible for *classifying the updates* from the update set S. In order to achieve this classify, it reads the entity-item tuples from the update stream and classifies them as per entities' partition (derived from the underlying KNN algorithm). A general data structure `EntitytoPartition` is available to know univocally each entity's partition.

As we have pointed out earlier, classify phase is designed to reduce disk operations. This is achieved by using two-layer in-memory buffers, which are read and written using a multithreading approach. With this, *UpKNN* makes sure that while performing the expensive read operations (to read the update set from disk), there are threads classifying the already read data, thereby achieving a higher throughput.

Let S be the stream of updates that contains new items for users i and j: $< i, 1 >$,$< j, 2 >$, and $< i, 3 >$. An inefficient updating of these profiles results in the reading of i's profile from disk twice. The system reads i's profile from disk, inserts item 1, and writes it back to disk. Then, it processes j's item. Now, a new item appears for user i, which leads to a new reading of i's profile from disk.

In general, a set of updates does not exhibit a specific appearance order of the new items. Updating profiles following such unsorted pattern results in multiple profile readings/writings from/to disk. Consequently, the number of disk operations increases, affecting system's performance.

Aiming to reduce disk operations by minimizing profiles reading/writing from/to disk, we classify updates per the entities' partition, such that all updates of the same partition be applied at once. The computation of all the updates of a partition implies loading profiles once, without further operations over an already updated profile during the current computation.

To elaborate on this, we present Figure §5.1, depicting the complete phase of *classify* in details.

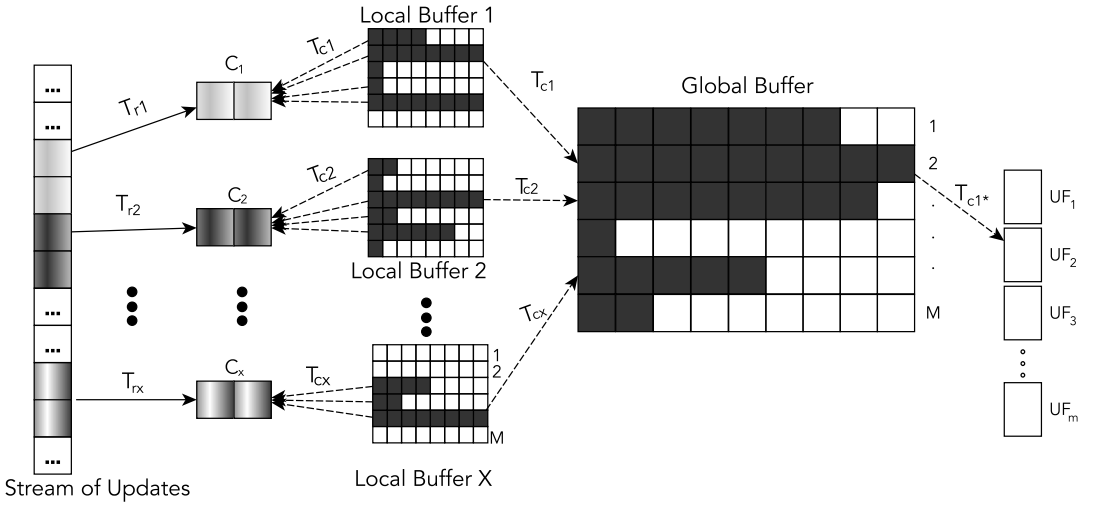


Figure 5.1: *Classify phase. Reader threads in continuous lines, classifier threads in dashed lines.*

As we can see in the figure, a stream of updates S having unsorted entity-item tuples is present in the disk. The classification process implying the use of various threads and buffers, explained shortly, classifies these updates and stores them into M update files, denoted by UF_m.

A priori, we have pairs of threads which we call *reader-classifier* threads, denoted by T_{ri} and T_{ci} respectively, as shown in the Figures §5.1 and §5.2. Each of these $T_{ri} - T_{ci}$ pair shares a unique communication channel, denoted by C_i in the figure. When the stream S is available, each reader thread T_r reads one of the equal-sized slices of S. The stream S is sliced so as to parallelize the process of reading in general, and update on a whole.

Once the T_r has read a slice from S, it puts that part of the stream in the communication channel and notifies the corresponding T_c of its presence. As T_c receives the notification, it accesses data from the communication channel, freeing it for new data (from T_r), and hence T_r is notified.

We should note that the $T_r - T_c$ threads communicate via a communication channel C, which leads to reduce unnecessary synchronization between them (only T_r accesses its slice of S on disk). As a direct result, we save computation time with this. While T_r performs longer I/O operations, T_c classify updates into partitions. This is a simple yet carefully designed and chosen mechanism to reduce the overhead involved in updating the profiles and hence to have a higher scalability.

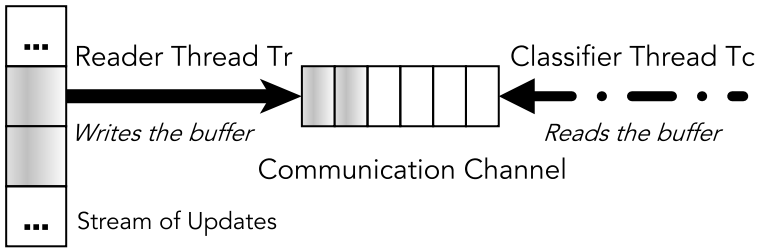


Figure 5.2: $T_r/T_c/C$ *configuration. (*T_r *in continuous lines,* T_c *in dashed lines).*

A key factor to achieve high performance in our multithreading approach is the overlap of computations and I/O operations. While a reader thread is obtaining data from the stream (I/O request), a classifier thread is reading the data from the communication channel, and classifying updates in partitions, preliminarily stored in local buffers and later written into the corresponding update files. Both tasks are performed concurrently, reducing the idle CPU time. Such I/O-computation overlap has improved the performance in I/O-intensive algorithms [43, 90, 119], of which KNN is a significant example. With such observations, we have carefully designed the classify phase of *UpKNN* deploying threads and buffers at various levels. Our evaluation Section §5.5 shows how such careful deployment helps in updating millions of updates on large number of profiles in online-computations.

Once the data has been read from S and is available for T_c, it is ready to be classified in the corresponding update file UF_m. To achieve this, each classifier thread T_{ci} has access to its M partitioned local buffer LB_i, each of size $M \times 4[mb]$. Therefore, the partition m belonging to the local buffer LB_i of thread T_{ci} is denoted by LB_{im}. The data read by T_c (from the communication channel) might have a large set of entity-item tuples to be updates belonging to different partitions. Keeping the large size of updates in mind, we implement a second level of buffer, mostly to reduce synchronizations and I/O operations. This buffer, having M partitions, is called *Global buffer*, common to all the classifier threads T_c. Due to this fact, each of the M partitions in the *Global buffer* is protected by a *mutex*, which prevents multiple classifiers to access the same partition concurrently. The size of the *Global buffer*, in our approach has been fixed to 8 [mb] per partition, thus leading to a total size of $M \times 8[mb]$. This size is experimentally selected by searching for the size that achieves the best performance.

Using the data read from C_i, the thread T_{ci} classifies the entity-item tuples and stores them into their corresponding local buffer partitions: LB_{im}. As soon as a partition m of any local buffer LB_i becomes full, its data is put into the corresponding partition of the global buffer by the corresponding T_{ci}. This process of putting the data, first into the partitions of local buffers followed by those of the global buffer, continues until any of the partitions of the global buffer becomes full. Once the global buffer partition is full, only then the data of that particular partition is written into the corresponding update profile file UF_m, stored in the disk. The thread $T_{ci}*$, who is responsible to write the update profile file, is the one who last wrote the data into the partition of the global buffer making it full.

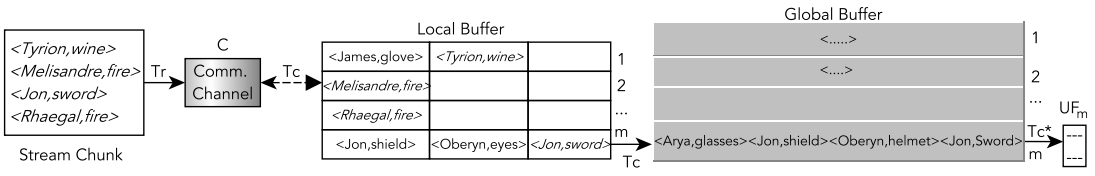


Figure 5.3: *Classification example.*

Figure §5.3 presents an example of how the classification works. A reader thread T_r reads its chunk of the stream and sends it to its classifier T_c through the communication channel C. T_c classifies the list of updates into their respective partitions. In the example, the local buffer of partition m is full, then T_c writes the content of this buffer into m's partition of the global buffer. As this partition on the global buffer is also full, T_c*, the last thread who wrote in

the global buffer, will write the content of partition m's global buffer into the corresponding file UF_m.

We use a two-layer buffer in order to perform the classification process *in-memory* as much as possible. With experimental values, we can see how the use of a global buffer, in addition to the local buffers, reduces the disk writing operations which are computationally more expensive as compared to the storage in buffers. The cost of writing a buffer, including the synchronization operations runtime, is several times lower than writing a file in parallel. In our design, as only one thread has access to the global buffer of some partition m, when this is full, there is no need of synchronization to write UF_m file.

To stop the computation, when a reader thread has nothing else to read from the stream, it notifies its classifier thread. Once the classifier has been notified, it checks the communication channel for one last time to see if there are any updates left to be classified. In case there are, it reads from them from the communication channel, continuing the whole process and terminating the thread by writing the contents of the corresponding local buffer into the global buffer.

The last classifier thread alive is in-charge of writing everything that remains on the global buffer into the updates files UF. We use a counter of alive classifiers so that each thread would know whether it is the last thread alive or not. To ensure consistency and integrity, we implement a checking condition to be sure that all reader and classifier threads are terminated, and nothing has been left unclassified.

5.3.1.2 Merge

The *classify* phase has already completed half of this task, i.e., classifying the updates read from the update stream into M files stored in the disk. The *merge* phase adds the updates from these UF_m files to the already existing M profile files PF_m in the disk.

Thus, the input of this *merge* phase is the set of update files UF_m, generated as a result of the *classify* phase. To carry out this phase effectively, a set of threads process the updates from these files in parallel. Noticeably, an obvious choice of the number of threads is same as the number of files, which in turn is same as the number of partitions, i.e., M. In other words, we ensure that we have enough threads T_m so that each thread accesses one update file at a time and hence all the files can be read and merged in parallel, leveraging internal I/O parallelism observed on modern high-speed SSDs, while adhering to the limited memory constraint of a single machine.

To perform the merge, each thread T_m loads the updates from the corresponding update file UF_m into memory. These updates are inserted sequentially into a *heap* to sort them, in this case, by entities' id. Sequential access to updates stored in files, implies sequential disk accesses, thus leading to better performance.

Once the updates are present into the heap, T_m proceeds to complete the process. The purpose of sorting the updates by entities' id on the heap is to have all the occurrences of a particular user continuously. Now that the updates are sorted by entities' id, T_m proceeds to read from disk the contents of the corresponding profile file PF_m (obtained from the underlying KNN approach) and to merge them with the updates from the heap (also read sequentially). The process of merging old profiles with new items is performed in-memory. Finally, the same thread T_m writes the updates profiles into the profile file PF_m. Using the same thread for reading and writing, avoids synchronization operations and related costs, and hence, achieves full parallelism in I/O operations.

Figure §5.4 shows an example of how the *merge* phase works. Let one of the threads T_j be in charge of processing the update partition file PF_j. T_j inserts the whole set of updates from the corresponding file into a heap. It then sequentially accesses the heap to apply the updates. We note that sorting the heap by entities' id helps the thread to apply all the updates of a user simultaneously, saving multiple read operations for each user. In this example, user *Jon*, has two items to merge into his profile. By accessing sequentially the heap, the algorithm obtains the list of *Jon*'s updates. Once the thread T_j has all items of user *Jon*, it loads *Jon*'s profile from the corresponding partition file PF_j, and merges these two updates in the profile (bolded items in PF_j). Finally, it writes back the modified profile to the corresponding profile file PF_j.

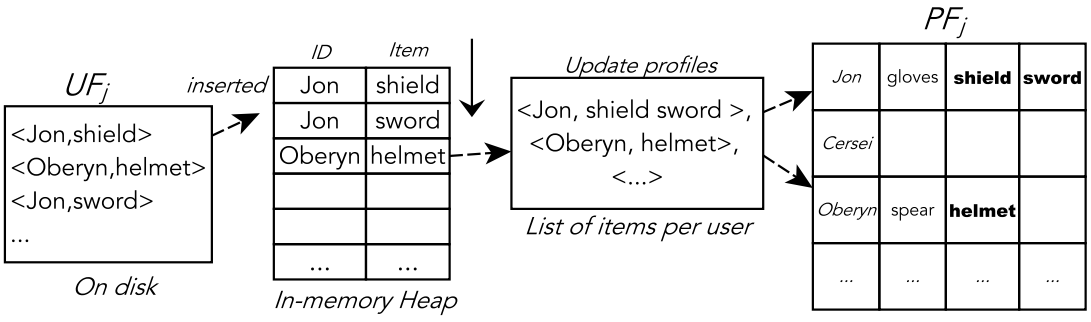


Figure 5.4: *Merge example.*

5.3.2 Implementing *UpKNN* on *Pons*

Although *UpKNN* is independent of the underlying KNN algorithm, we show a particular instance of its implementation on *Pons* [24]. *Pons* is multithreading out-of-core approach for computing KNN on large datasets. For this purpose, the algorithm is adapted to run on a stand-alone machine, partitioning data accordingly to the available memory.

As a remainder of Chapter §4, *Pons* performs the approximate KNN computation iteratively. At iteration t, the data is modeled by a directed graph $G^{(t)} = (V, E^{(t)})$, where V is the set of $N = |V|$ entities, and $E^{(t)}$ represents edges between each entity and its current K-nearest neighbors. Each entity v has a profile F_v of fixed length P.

For computing KNN, *Pons* divides the set of users in M equal-sized partitions, respecting the memory constraints. *Pons* executes iteratively 5 phases. Firstly, divides the set of users into M partitions. Secondly, creates in-edge/out-edge partition files, required for some phases of the computation. After, it creates the profile partition files `j.prof`, which store the profile of all users assigned to partition `j`. This set of files is one of the inputs of *UpKNN* (PF_m in *UpKNN*). It is important to mention that *Pons* stores the profiles sequentially sorted by the user id. This feature helps to improve the performance of the update phase, because updates inserted in the heap are also sorted by user id, thus the reading/writing of profiles from/to disk is also sequential.

Algorithm 8: *Pons* and *UpKNN*

Data: Graph file: File(G), Profiles file: File(F)
Result: Each vertex $v \in G$ finds its KNN.

```
1  begin
2      foreach Iteration t do
3          1. Partitioning(GlobalOutEdges)
4          2. Create In-edge Partition Files
5          3. Create Out-edge Partition Files
6          4. Write Profile Partition Files: PF_m
7          5. Update Profiles
8          5.1 Classify Phase
9          5.2 Merge Phase
10         6. Compute Distances
11         Update(GlobalOutEdges)
```

Algorithm §8 shows the pseudo-code of how *UpKNN* runs within a *Pons*'

execution. As we mentioned above, *UpKNN* requires the set of partition profiles PF_m to perform the updates, these files are created in line §6. Even though our approach works well within *Pons*, it can be implemented on top of other approaches as well, it is only required the access to the partitioned set of users profiles. Best performance is obtain when the accesses to the set of profile do not incur in random disk operations.

UpKNN is executed in lines §7-§9, when *Pons* has already created the set of profile files. The output of *UpKNN* is the new set of profiles that includes all new items added by the users. *Pons* continues the execution in the phase 6, performing distance computation, by comparing the profiles of a user and its current neighbors, neighbors' neighbors and K random neighbors (detailed in Section §4.4.5).

5.4 Experimental setup

UpKNN is extensively evaluated on two datasets, *Movielens* [44] and *Mediego*, detailed in Section §5.4.2

We have implemented *UpKNN* in roughly 1000 lines of *C++* code, compiled on *clang-omp++* 3.5.0, using -O2 optimization. In addition, we have used *Openmp* and *Pthreads* to enable multithreading computation.

5.4.1 Machine

We have performed our experiments on a MacBook Pro laptop, equipped with an Intel Core i7 processor (Cache 2: 256 KB, Cache 3: 6 MB) consisting of 4 cores, 16 GB of RAM (DDR3, 1600 MHz) and a 500 GB (6 Gb/s) solid state drive (SSD).

5.4.2 Datasets

Out of the two datasets that we use to evaluate *UpKNN*, Movielens is publicly available, while the other one, Mediego[†1], is not. These datasets represent two entirely different domains, one where users have shown their preferences in a huge set of movies (Movielens), and the other which has a collection of users of content edition web-sites (Mediego).

[†1] http://www.mediego.com/

1. *Movielens*: provides the movie-rating data gathered from the *Movielens* recommender website over a duration of around 7 months. In this dataset, the items are the movies. In a typical setting, a user has a rating associated to the movie she has watched. These affinities, in form of ratings from 1-5 are reflected in user-profiles. But in our case, we do not require the ratings, and hence only consider the users and their associated movies. Hence, these user-movie(s) associations are used to construct the user profiles for *UpKNN*.

2. *Mediego*: consists of users and the web-pages they visit from various different websites[†2]. Each user activity has a timestamp associated to it, which is used to partition the profiles into initial profiles and the updates.

Table §5.1 shows the statistics of the datasets in terms of number of users and items. In both the datasets, each user activity has a timestamp associated to it, which is used to partition the profiles into initial profiles and the update stream. We use 20% of the items as the initial profiles, and the remaining 80% items are used to populate the update stream S. Table §5.1 details the total number of updates (80% of items). 20%-80% division is done on the basis of the timestamps present for each user-item interaction. As a result, the first 20% of the profiles sorted on timestamps become the initial profiles and the remaining 80% that follow in time constitute the updates.

Dataset	Users	Items	#Updates **(80% items)**
Movielens (MOV)	138,493	20,000,263	16,000,210
Mediego (MED)	4,130,101	7,954,018	6,363,214

Table 5.1: *Datasets description in number of users, items and number of updates (80% of the dataset size).*

5.5 Evaluation

In this section, we present the results and evaluate the performance of *UpKNN* around various parameters.

[†2]as it is a proprietary database, we cannot disclose the names

We do an extensive analysis of the results obtained for the datasets considered and of various factors related to the datasets and of the approach itself. Specifically, we present the results and evaluate the performance of *UpKNN*, in terms of wall-time, which is the total time taken to integrate the updates in the existing profiles from the time of receiving them.

As we show in this section, *UpKNN* performs efficient profile updates in sparse and dense datasets, scaling linearly with respect to the number of update processed. Additionally, *UpKNN* is able to process 3.2 millions of updates per second in average. These good results are based on a carefully designed multithreading approach along with the use of in-memory buffers to perform faster operations, and the reduction of disk operations to avoid the impact of the disk latency in performance.

5.5.1 Performance

We evaluate the performance of *UpKNN* on both the datasets, i.e., *Movielens* and *Mediego*, over various parameters.

As we have mentioned earlier, to the best of our knowledge, currently there are no algorithms that consider the updates on entities' profiles while computing the KNN in an online fashion. This renders it difficult for us to compare *UpKNN* against other approaches. To overcome this, we choose a natural baseline, against which we can compare *UpKNN*. For a fair comparison the baseline also uses a multithreading approach where several threads read the updates from the update stream and add them to the respective profiles. In other words, a thread reads a entity-item tuple value from the stream of updates, and accordingly loads the corresponding entity's profile from the disk and modifies it with this new item. Once the profile has been modified, it is written back to the disk.

We now compare the performance of *UpKNN* with the baseline that we have set.

5.5.1.1 Runtime

Table §5.2 shows *UpKNN*'s and baseline's wall-time for computing a certain number of updates on each of the datasets, along with *UpKNN*'s speedup. Logically, as we see in the table, more the number of updates, higher is the time taken for the process. We have highlighted in bold, the best speedups achieved by *UpKNN* when considering the 80% of updates on

20% of the initial profiles[†3]. We also experiment with 100M updates, by using random updates (as the maximum number of updates for each of the dataset is less than 100M), to test *UpKNN*'s scalability, and also to see the difference with the baseline. As the datasets these days cross over billions of users, we claim that these numbers (10M, 100M updates) are a good representation of the reality.

Dataset	#Updates	*UpKNN* [sec]	**Baseline** [sec]	**Speedup**
MOV	1M	0.795	10.869	13.67X
MOV	10M	3.635	105.747	29.08X
MOV	20/80	3.687	184.513	**49.5X**
MOV	100M	39.662	1055.804	26.61X
MED	1M	5.658	20.509	3.62X
MED	20/80	1.543	72.576	**47X**
MED	10M	17.665	198.658	11.24X
MED	100M	47.329	1931.154	40.80X

Table 5.2: *UpKNN's runtime and speedup (with respect to the baseline) in updating entities' profiles.*

As highlighted in bold, we see that *UpKNN* considerably outperforms the baseline, for both the datasets. *UpKNN* achieves a speedup of 49.5X for the *Movielens* dataset, taking only 3.687 seconds for about 16 million updates. Similarly, we obtain a speedup as high as 47X for the *Mediego* dataset.

Although *UpKNN* performs consistently good whether the updates are ordered according to time or randomly, there is a difference in performance. This difference in performance can be attributed to the high user activity around a given timestamp. For example, while rating movies, a user is more likely to rate several movies at one time than to rate movie as per the time she watches them. Similarly, while reading the news, a user tends to go through several news items around a given timestamp rather than consulting one news article every 15 minutes or 1 hour. This kind of clustering of user activity around time, lets *UpKNN* process more updates per second when the updates are ordered in time, as compared to random updates. Such ordering favors heap's performance in *merge* phase.

We notice from Table §5.3, that *UpKNN* is capable of processing more than

[†3]Table §5.1 details the actual values

4 million [updates/sec], for both the datasets considered. This is consistent with the motivation of our work. Our solution not only performs on a single commodity PC, but it also performs in real-time, making it a novel approach in itself. Moreover, we analyze the wall-time taken by *UpKNN* when the updates to be applied are chosen randomly instead of ordering them by timestamp. The results are shown in Table §5.3 (3rd and 4th row). Again, for both the datasets, we are able to process more than 2 million [updates/second], even when the updates are not ordered according to the timestamp.

Dataset	#Updates	**Time[s]**	#Updates/second
MOV	20/80 (16M)	3.678	4.33M
MED	20/80 (6.3M)	1.543	4.12M
MOV	100M	39.662	2.52M
MED	100M	46.329	2.11M

Table 5.3: *Number of updates processed by second on UpKNN.*

In addition to the very little time taken by *UpKNN* to process millions of updates, with Figure §5.5, we also verify its scalability in terms of updates processed. We see that even after increasing the number of updates from 10M to 100M, the execution time increases only by a factor of 10.

Moreover, we show in Table §5.4, the rate at which the updates are processed per KNN iteration. For this, we measure the wall-time of one KNN iteration, and the number of updates processed per second (from Table §5.3).

A KNN iteration compares entities' profiles to select K-nearest neighbors of each entity in the system. However, profiles compared during the KNN computation are not affected by the update processing, since the profile comparison are made using a prior version of the profiles.

We observe high numbers of updates processed during one KNN iteration, specifically in those longer iterations. This fact shows an opportunity to update the profiles while the KNN computation runs. To do so efficiently, we use only a small fraction of the resources to update the profiles, preventing the KNN computation's performance from getting adversely affected.

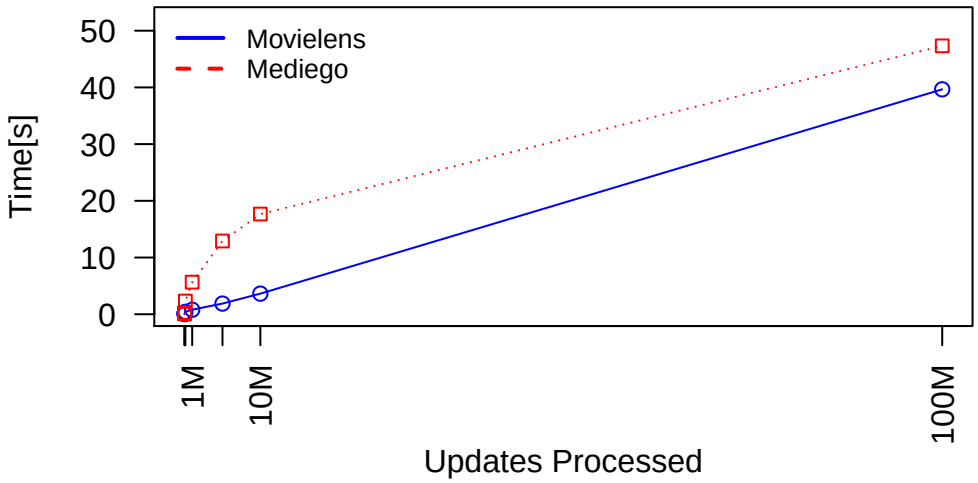


Figure 5.5: *Scalability in terms of updates processed.*

Dataset	#Updates/second	Iteration Time[s]	#Updates/iteration
MOV 20/80	**4.33M**	7.489	32.4M
MOV 100M	2.52M	50.671	127,6M
MED 20/80	**4.12M**	127.930	527M
MED 100M	2.11M	253.294	534.4M

Table 5.4: *Number of updates processed in a KNN iteration.*

5.5.1.2 Disk operations

The crux of *UpKNN* is a careful employment of threads and two-layered buffers to accelerate the processing of updates. This acceleration is achieved as a direct result of the reduction in the number of disk operations.

To underline the performance of *UpKNN*, we present an overview of various disk operations involved in the implementation and execution of *UpKNN*. We also compare our results to the baseline set earlier (multithreading algo-

rithm to read the data stream and classify it sequentially, unlike in *UpKNN*). Table §5.5 presents the number of disk seeks, bytes read and written, and number of read and write operations for both *UpKNN* and the baseline. Columns % present the percentage of operations performed by *UpKNN* with respect to those performed by the baseline.

As evident from the Table §5.5, *UpKNN* reduces considerably the number of disk operations performed throughout the updating process. In the case of ordering the updates in time (as they appear in the source datasets) and taking the first 20% to construct initial profiles and the remaining 80% to constitute the update stream, we obtain better results than the case where the updates are randomly put in the stream (and not ordered in time). In the former case, *UpKNN* takes only 0.0006% of the seeks performed by the baseline. Similarly, the number of bytes written in our approach (with 20/80 division of profiles, ordered in time) is reduced to only 1.98% of those of the baseline, and the total bytes read is reduced to 3.88% with respect to the number of bytes read for the baseline. These differences are explained by *UpKNN*'s capability to apply all the updates for a profile at once. A profile is read from disk once, modified in memory, and written back to disk once. In the other case, the baseline reads/writes the whole profile each time there is an update for it.

We also observe similar differences in case of updates applied randomly. As shown in Table §5.5, *UpKNN* consumes very few disk operations as compared to the baseline. For updating *Mediego* dataset using 100 millions random updates, our approach performs only 2.753% of the disk seek operations performed by the baseline. The percentage of written bytes on our approach is only 6.11%, and read bytes is only 7.933% compared to the number of these operations performed by the baseline.

UpKNN's performance relies on its capacity to reduce disk operations throughout each phase of the computation. For instance, the stream of updates (read from disk) is accessed only once on the classification process. In addition, the utilization of a *heap* reduces the need of multiple profile reads/writes from and to the disk. As a result of this optimization, each profile is read/written only once.

MOV 20/80				**MED 20/80**		
	UpKNN	Baseline	%	*UpKNN*	Baseline	%
Disk seeks	29	48M	0.0006	27	19M	0.0001
Write [bytes]	128M	6400M	1.98	50M	2570M	1.98
Write [times]	12	16M	0.0001	10	6M	0.0001
Read [bytes]	256M	6592M	3.88	101M	2621M	3.88
Read [times]	127	32M	0.0004	55	12M	0.0004
MOV 1M				**MED 1M**		
	UpKNN	Baseline	%	*UpKNN*	Baseline	%
Disk seeks	276K	3M	9.22	1.7M	3M	59.221
Write [bytes]	64M	404M	15.821	367M	404M	90.81
Write [times]	138K	1M	13.842	888K	1M	88.83
Read [bytes]	72M	412M	17.456	375M	412M	90.99
Read [times]	138K	2M	6.921	888K	2M	44.41
MOV 10M				**MED 10M**		
	UpKNN	Baseline	%	*UpKNN*	Baseline	%
Disk seeks	277K	30M	0.923	7.5M	30M	25.101
Write [bytes]	136M	4040M	3.365	1601M	4040M	39.632
Write [times]	138K	10M	1.385	3.7M	10M	37.651
Read [bytes]	216M	4120M	5.241	1681M	4120M	40.804
Read [times]	138K	20M	0.692	3.7M	20M	18.826
MOV 100M				**MED 100M**		
	UpKNN	Baseline	%	*UpKNN*	Baseline	%
Disk seeks	277K	300M	0.092	8M	300M	2.753
Write [bytes]	856M	40KM	2.118	2468M	40KM	6.110
Write [times]	138K	100M	0.138	4.1M	100M	4.130
Read [bytes]	1656M	41KM	4.019	3268M	41KM	7.933
Read [times]	139K	200M	0.069	4.1M	200M	2.065

Table 5.5: *Disk operations. % of* UpKNN*'s operations with respect to those of the baseline.*

5.5.1.3 Number of threads

In this section, we evaluate *UpKNN*'s capacity to scale with respect to the number of threads available to perform the computations. We show that our approach leverages the computational resources to perform better. Figure §5.6 presents the wall-time required to execute 100 millions updates, using various numbers of threads. We observe near-linear decrease in the runtime when the number of threads increases. When the number of threads available for the computation grows, the wall-time decreases near-linearly, the small difference is mainly due to the increase in the synchronization among threads.

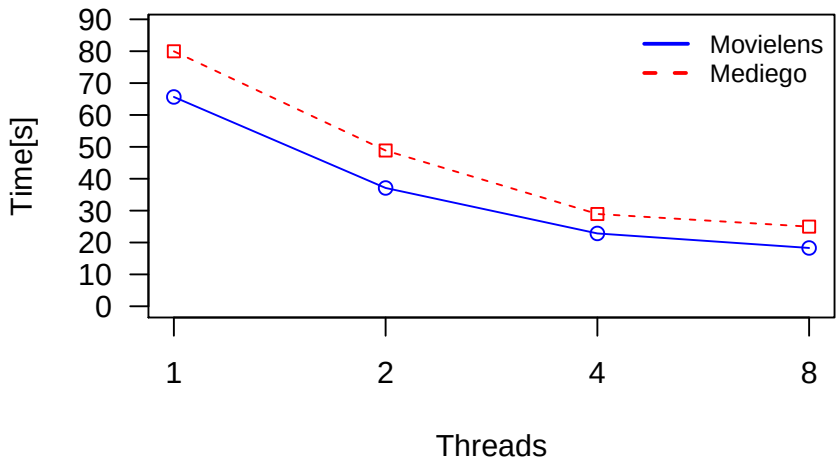


Figure 5.6: *Threads scalability.*

5.5.2 Evaluating usage of data in memory

We observe that one of the most costly operations performed throughout the computation is the reading of profiles from disk, when they have to be updated. Our approach is designed to make an efficient usage of the data loaded in memory, by applying all updates of the same profile immediately, with no further loading of the same profile from disk. *UpKNN*'s performance improves even more when a profile has numerous updates, thereby reducing the disk reads significantly.

We note that it is possible to face a scenario where the reading of profiles is done to apply only one update at a time. This possible scenario degrades performance, the time of a I/O request for reading a profile from disk is larger than the negligible time of adding only one item in memory. Our approach has been designed to apply several items into the profiles at once. Therefore, the larger the number of items applied per profile, the better the use of such profile in memory. In the next experiment, we evaluate *UpKNN*'s performance when this worst case happens.

To evaluate the performance in such a case, we used a modified version of the datasets. Each user updates her profile with only one item ($\#Updates = N$).

For the sake of a fair comparison, we compare *UpKNN*'s update time with respect to the time needed to perform exactly the same number of updates, but in an average case. This average case means that a profile may be updated with 0, 1 or more items (in total $\#Updates = N$).

Table §5.6 compares runtime of *UpKNN* facing the worst case scenario described above, and a more common case (average case in the table). Table §5.6 shows that *UpKNN*'s runtime does not increase considerably. Despite the overhead caused by the loading of a profile for only a small task in memory, *UpKNN* continues to perform well.

Dataset	$\#Updates = N$	**Worst case** [sec]	**Average case** [sec]
MOV	138,493	0.642	0.502
MED	4,130,101	16.418	11.830

Table 5.6: *Worst case (1 update per user) vs average case (same number of updates but distributed among all users).*

5.5.3 Evaluating design decisions

A fundamental optimization in the code is the implementation of a *heap* to store all updates for the same user sequentially. By doing so, each profile is fully updated at once. Such an operation helps in reducing the number of profile reads/writes from/to disk, the most time-consuming tasks performed on *UpKNN*.

While the heap reduces such costly operations, it also creates an additional overhead. The heap must sort the updates by the id of the entity, increasing runtime when the number of updates is large.

In the following, we apply 100 millions random updates, and we measure the runtime of such processing in two cases, when *UpKNN* implements the heap, and when it does not.

Figures §5.7 and §5.8 show the time required for the modification of the profiles applying 100 millions random updates. Despite the fact that the usage of a heap adds an overhead to the computation, in both the datasets the heap improves considerably the performance of the system. The runtime of updating profiles in *Movielens* and *Mediego* datasets is only roughly 4% of the time without using the heap. We conclude that the additional overhead incurred by the use of the heap is negligible, and it is essential to reach performance.

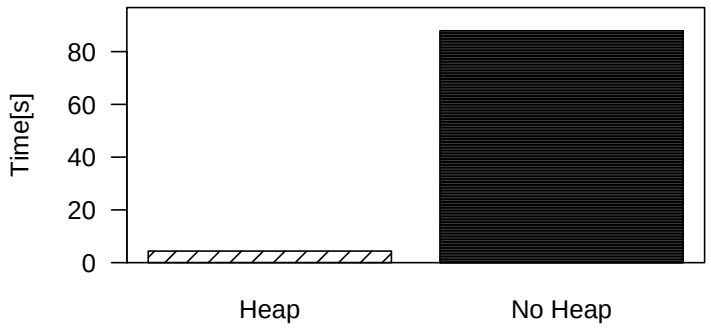


Figure 5.7: *Heap improvement (*Movielens *Dataset).*

Table §5.7 helps to understand how, despite the overhead incurred by the use of a heap, *UpKNN* achieves such a performance. The table presents the number of disk operations performed to update the profiles, comparing an execution that uses the heap, with other that does not. The % columns compare the number of operations performed using a heap with respect to those without a heap.

We observe that the heap helps to reduce the number of disk seeks,

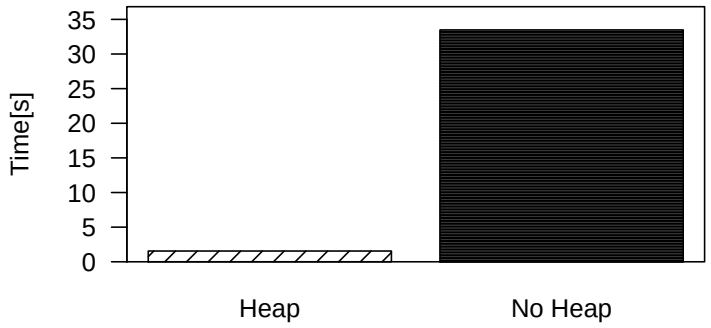


Figure 5.8: *Heap improvement (*Mediego *Dataset).*

	MOV 100M (random)			MED 100M (random)		
	Heap	No Heap	%	Heap	No Heap	%
Disk Seeks	27	32M	0.00008	27	12.7M	0.00021
Write [bytes]	50.9M	6592M	0.772	51M	2621M	1.941
Write [times]	10	16M	0.00006	10	6.3M	0.0001
Read [bytes]	230M	6720M	3.419	102M	2672M	3.809
Read [times]	55	16M	0.0003	55	6.3M	0.0008

Table 5.7: *Disk operations heap/no heap.*

read/written bytes and number of I/O operations. Without the heap, are required multiple I/O operations, each time a profile is modified. The algorithm should perform I/O tasks to read the profile from disk and write it back when the modification is done. In the opposite case, using the heap, the profile is read only once from disk, updated simultaneously with all its updates, and written back to disk.

5.6 Conclusions

We presented *UpKNN*, a multithreading approach for computing *K-Nearest Neighbors* with temporal user preferences on very large datasets. We have shown extensive experiments to claim that this scalable solution can be achieved on a single commodity PC.

This novel approach achieves high performance (measured in terms of wall-time) by moving away from traditional random access approaches towards a more efficient partition-based idea. The performance of our implementation relies on a carefully designed multithreading approach which makes use of two-layer in-memory buffers that overlap I/O requests and CPU computation throughout the processing. Such an overlap of operations is responsible for the optimization of I/O operations to read and write large blocks of data from and to disk. These overlapped operations also reduce the disk seeks and random read/write operations which are usually the main bottleneck in disk-based algorithms.

Along with a detailed description of the two-phase *UpKNN*, we have also supported our results with an exhaustive evaluation of *UpKNN* on a single commodity machine, using a well-known publicly available dataset and another larger proprietary dataset. *UpKNN* shows the capability to update 100 millions of items in roughly 40 seconds, achieving speedups ranging from 26X up to 50X compared to the baseline. We also showed that *UpKNN* scales both in the number of updates processed and the threads devoted to the computation.

Various experiments prove that *UpKNN*'s performance is a combination of the reduction of the random disk operations and efficient multithreading design that minimizes the need of thread synchronization, aiming to exploit full parallelism.

Part IV

Final Remarks

Chapter 6

Conclusions

In this chapter, we conclude this thesis contrasting the initial objectives of this work (Chapter§1) with the actual research done and the results achieved (Chapters §3, §4 and §5). Furthermore, we draw some lines of the future work, based on some open research lines stemming from this work.

6.1 Thesis summary

In this thesis we addressed the challenges of scaling the computation of the *K-Nearest Neighbors (KNN)* algorithm on single machines. Specifically, as we mentioned in Chapter §1, performing efficient KNN computation on very large datasets requires a significant amount of memory. As if that were not enough, this type of computation is a very time-consuming task. In this regard, many works have been proposed to make an efficient use of both single machines as well as distributed systems. In the one hand, using single machines to compute this algorithm brings new challenges in the development of more efficient approaches, able to process efficiently despite the fact that the memory footprints observed on very datasets go far beyond what a single machine is capable of handling. Although we may say that the single machines have increased their capacity over the years, unfortunately, the amount of data to be processed grows incredibly faster. On the other hand, notwithstanding that distributed systems overcome the limitation of the available memory in single machines, the design, implementation, and deployment of distributed algorithms still remain challenging. Besides the cost in terms of workload, distributed systems bring a significant monetary cost. Implementing efficient distributed algorithms requires a not small set of machines running together to perform the computation. Sadly, not all the potential users (namely modest research environments or small companies) can afford it.

Due to its high cost, many works on KNN computation, specially those processing KNN queries on very large datasets, perform the computation in an offline fashion. In other words, offline periods when the system is not working or only executing a reduced workload. In general, KNN algorithms represent just a layer of major systems, hence they do not have access to the full set of resources at any time. Although this offline processing decongests system's load to prioritize online tasks, it may harm the quality of the KNN outcome.

Furthermore, the high-cost of computing KNN leads to a simplification regarding data dynamism. Current state-of-the-art works only deal with static sets of data throughout the computation. This simplification reduces the computational complexity and runtime of the algorithm. Unfortunately, despite it brings some benefits, it also induces a significant shortcoming. Computing on static data does not reflect appropriately its true dynamic nature, affecting the potential results. In these days, data flows and changes rapidly, thus processing on static data (or updated infrequently), makes us lose some valuable information that arises from the dynamism observed on a small time scale.

Updating data during the computation adds new complexities to the algorithm, specially if this data is continuously accessed, as KNN does to perform the comparisons. Furthermore, as data changes rapidly, updating data in an online fashion would force the algorithm to process, not only its own KNN task, but also large streams of dynamic data simultaneously. Such a concurrent processing brings new challenges in the design and implementation of algorithms capable of computing KNN on data that changes continuously and rapidly over time.

6.2 Objectives of this thesis

In this general context, this work aimed to scale the computation of the K-Nearest Neighbors algorithm on single machines. A scalable solution must be capable of processing large current datasets within a reasonable time, considering the limitations imposed by a restricted set of computational resources. The motivation behind the use of single machines instead of more complex distributed systems, is the ease of access to this sort of computational resource and its lower cost with respect to that of the distributed systems, both in monetary terms as well as human workload. In algorithms designed for running on single machines, synchronization, data consistency, and some others well-known issues in distributed systems, do not need to be addressed. Besides, single machines have shown good performance running well-designed algo-

rithms, along with a good extent of simplicity in the design, coding, and deployment of complex algorithms.

In this work, we do not only aimed to scale the KNN computation, but we also aimed to build a lightweight approach, able to leverage the limited resources of a single commodity machine. Thus, becoming an inexpensive, but not less efficient, approach. A lightweight approach lends itself as a solution for performing online KNN computation, mainly due to its capacity to run well using fewer resources. Therefore, an online KNN approach is a valuable solution whether, along with using fewer resources, it runs within reasonable times.

Finally, another objective of this thesis was to propose an efficient solution for processing updates on data during the KNN computation. As we mentioned above, the dynamic nature of data has not been properly reflected and adequately handled on static algorithms, as current KNN state-of-the-art approaches do.

In this thesis, we fulfilled these objectives through two main contributions, which we summarize in the following.

6.3 Contributions

6.3.1 *Pons* [24]

In Chapter §4 we have presented *Pons*, an out-of-core algorithm for computing KNN on large datasets that do not completely fit in memory. Our approach computes KNN incurring a minimal cost, by storing all data in hard disk, loading and processing this data from disk into a limited section of the main memory. The main rationale of our approach is to minimize random accesses to disk, and to favor sequential readings from disk. *Pons*' performance relies on its ability to partition the data into chunks such that the subsequent accesses to these data segments is highly efficient, while adhering to the limited memory constraint of a single machine.

The experimental evaluation performed on large-scale datasets showed that *Pons* computes the KNN in roughly 7% of the time required by a fully in-memory implementation. *Pons* has shown to be also capable of computing online on machines with memory constraints, using only a limited fraction of the system's memory, freeing up resources for other tasks.

6.3.2 *UpKNN* [53]

In Chapter §5 we have presented *UpKNN*, a scalable and memory efficient, thread-based approach for processing real-time updates in KNN algorithms. By using a thread-based approach we accessed and partitioned the updates in real-time, processing millions of updates online, on a single commodity PC.

To achieve a good performance, *UpKNN* greatly reduces the number of disk operations performed during the computation, favoring the reading and writing of large chunks of data from disk. Our carefully designed multithreading approach leverages the use of two-layer in-memory buffers to reduce synchronization between threads and concurrency issues in I/O operations.

The experimental results showed *UpKNN*'s capability to update 100 millions of items in roughly 40 seconds, scaling both in the number of updates processed and the threads used in the computation. Thereby, *UpKNN* achieves speedups ranging from 13.64X to 49.5X in the processing of millions of updates, with respect to the performance of the baseline.

Various experiments proved that *UpKNN*'s performance is achieved by the right combination between the reduction of the random disk operations and our efficient multithreading design. In this regard, we showed that these results have been achieved by performing roughly 1% of the disk operations performed by the baseline. Experiments also showed that *UpKNN* processes an average of 3.2 millions updates per second, making our approach a good solution for online KNN processing.

6.4 Perspectives

In this section we look forward, describing what is beyond this work. To do so, we firstly look back upon our contributions, spotlighting the limitations of our proposal. We describe some ideas both to improve the scalability and to make a better use of the resources on our contributions. Secondly, we describe a set of possible extensions of our work for the future.

Pons

Having a look at *Pons*, we observe the use of a fixed profile size to improve the efficiency of I/O operations. Fixing this value allows us to know beforehand the size of the blocks read/written from/to disk, improving the efficiency of the I/O requests by handling optimized blocks of data during the computation. Although such an approach improves the performance, it does

not agree with reality in every case. Datasets used in KNN algorithms rarely exhibit identical profile size across the whole set of profiles. Therefore, this assumption increases the memory footprint of the algorithm, allocating wasted memory space for profiles that may contain less items than the memory reserved. Otherwise the opposite case is also possible, which implies an underestimation of the space in memory assigned to the profiles, in other words, the profiles contain more items than the fixed size assumed in the algorithm. In order to address this issue, we propose to study the possibility of maintaining a fix profile length, but it can be fixed dynamically based on certain properties of the dataset as average profile length. This dynamic fixation allows us to keep the benefits of fixed block sizes but adapting the length according to the data.

Another critical point that we observe in *Pons* is the amount of data reread and re-written over iterations. Our approach reads/writes a set of files for both in-edges and out-edges of the graph at each iteration of the algorithm, even if the edges did not change during the previous iteration. Unfortunately, these operations are a major bottleneck in I/O-intensive algorithms, such as KNN. We propose to study the feasibility of an optimized way of reading/writing only in cases when the data has changed. An efficient way to optimize these operations is to mark changed data, and perform the I/O operations only when necessary. Reducing these I/O operations we will undoubtedly improve algorithm's scalability and performance.

A final possible adjustment for *Pons* is to optimize the partitioning algorithm. In *Pons*, we implemented a specific partitioning algorithm to increase the probability of assigning neighbors to the same partition. Such an optimization improves performance of the algorithm, through the favoring of profile comparison of entities in the same partition. In other words we aimed to promote intra-partition comparisons, thus reducing the inter-partition operations, which are considerably more expensive. A possible improvement on this algorithm is to also favor neighbors' neighbors assignation into the same partition. As well as our current partitioning algorithm, this optimization will also aim to favor intra-partition operations, processing more comparisons in the same partition, reducing costly inter-partition communications.

UpKNN

Regarding *UpKNN*, we can point out few issues to improve in order to get better performance and scalability. Firstly, we observe the *merge* phase of *UpKNN*, which is devoted to incorporate updates into profiles by using a set of threads to perform this operation. Each thread processes the whole update set of a partition, loading the updates and the set of profiles in such partition completely in memory. As the set of threads does the same operation

concurrently, we note that the total memory footprint can exceed the boundaries of the machine. This incurs in multiple accesses to the virtual memory, degrading performance. As a consequence, we observe in our experiments evaluating *UpKNN*'s scalability that the improvement is near-linear when the number of threads increases. This defect is mostly caused by the impact of the virtual memory in the processing. Consequently, we suggest to improve this operation by limiting the memory allocated, dividing it adequately among threads, favoring multithreading as much as possible.

Finally, we draw some lines regarding future work. Firstly, having in mind the fast growth of data, we need to increase the scale of the datasets tested on our contributions. The primary objective leading this thesis is to scale the computation on single machines, considering that this computational setup provides us an inexpensive albeit restricted set of resources. Although we have proved that it is feasible to scale the KNN computation on single machines, it is impossible not to notice that this scalability obviously has a limit. To go beyond, we necessarily need to think of large scale distributed systems as an option. Both distributed algorithms as well as cloud-based approaches allow us to scale the computation considerably. Although feasible, we have to address carefully a set of well-known issues in distributed systems, namely, consistency, fault tolerance, synchronization, among others.

Considering a larger scale distributed extension of our work, as we already mentioned in Chapter §2, a key issue in distributed computing is the difficulty of finding an appropriate partitioning algorithm. On the basis of distributed algorithms is the fact that the data is divided and then processed by several machines simultaneously. To divide this data we need to partition it across the machines, fulfilling the needs of this specific KNN algorithm. A fundamental feature of this partitioning algorithm will be its capacity to favor, as much as possible, the assignation of both neighbors as well as neighbors' neighbors into the same machine. Such an assignation would favor intra-machine rather than inter-machine operations, thereby improving the performance of the algorithm.

In the same distributed environment, another focus of our future work is to find an enhanced profile representation as well as an efficient distributed storage of them. A decentralized version of the profiles has to be both lightweight as well as an accurate version of them, considering that the profiles are communicated through the network during the profile comparisons. Additionally, a good profile representation should support updates over time, considering that these profile are handled by several machines, which are probably geographically far apart, hence the communication cost is non negligible at all.

Part V

Appendix

Chapter 7

Résumé

Depuis plusieurs années, nous avons assisté à une croissance écrasante des données générées. Selon le rapport d'IBM, environ 2,5 trillions octets de donnés sont créés chaque jour[†1]. Par exemple, des centaines d'heures de vidéos sont visionnées chaque minute sur *YouTube*[†2]; il y a en moyenne 350 000 *tweets* émis par minute sur *Twitter*[†3]; et 300 millions de photos sont envoyées sur *Facebook* chaque jour[†4].

Bien que l'accès à une grande variété de données peut être utile pour les utilisateurs, cette énorme quantité de données devient inutile si elle est mal classée, filtrée, ou affichée. En particulier, il est possible de recueillir de l'information pertinente après avoir répondu aux questions suivantes: (1) *Comment puis-je trouver des données similaires dans un vaste ensemble de données ?* (2) *Comment puis-je trouver des éléments semblables à ceux que j'aime dans un vaste monde tel qu'Internet ?* Et plus précisemment, (3) *Comment puis-je trouver de la musique, des photos, et des livres similaires à ceux que je connais déjà?*

La méthode des *K-plus proches voisins* (dénoté KNN pour l'anglais *K-Nearest Neighbors*) est le socle de nombreuses approches capables de répondre à ces questions. Dans cette thèse, nous nous concentrons sur des algorithmes de KNN, qui se sont révélés être une technique efficace pour trouver des données similaires au sein d'un grand ensemble de données. Bien que le KNN n'est pas la seule méthode existente, il a certainement gagné en popularité [117] grâce à sa qualité, sa simplicité et sa polyvalence.

Cependant, comme tout algorithme efficace et polyvalent, le KNN est

[†1] https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html
[†2] http://expandedramblings.com/index.php/youtube-statistics/
[†3] http://www.internetlivestats.com/twitter-statistics/
[†4] https://zephoria.com/top-15-valuable-facebook-statistics/

coûteux. Ce coût peut être exorbitant, surtout au vu de la croissance effrénée de la quantité de données générées.

Dans un monde où les données changent continuellement, effectuer des calculs efficaces de KNN sur de grandes bases de données nécessite des quantités importantes de mémoire. Par ailleurs, en plus du besoin important en mémoire, l'exécution du KNN se révèle être un processus coûteux en lui-même. Compte tenu de ces faits, de nombreux travaux ont proposé plusieurs algorithmes pour tirer parti de manière efficace des ressources locales aux machines isolées et des ressources partagées dans les systèmes répartis. D'une part, l'utilisation de machines isolées apporte de nouveaux défis sur l'utilisation de la mémoire, qui est généralement limitée. Malheureusement, stocker l'ensemble des données d'un KNN dans la mémoire n'est pas toujours possible. D'autre part, nonobstant le fait que les systèmes répartis ont surmonté la limitation en mémoire disponible pour les machines, la conception, la mise en œuvre et le déploiement d'algorithmes répartis restent encore très complexe.

Comme conséquence directe de son coût élevé, nous pouvons observer sur certains travaux de l'état de l'art [13, 14, 32] que le calcul de KNN est souvent effectué hors-ligne (généralement pendant la nuit). Bien que le calcul hors ligne de KNN décongestionne la charge du systéme pour laisser de la place aux processus plus importants, il s'éxécute malheureusement sur des données statiques ou obsolètes, ce qui pourrait être nocif pour la qualité des résultats du KNN.

Le coût élevé des calculs de KNN conduit également à une simplification en ce qui concerne le dynamisme de données. Des travaux récents dans l'état de l'art traitant du KNN ne considère uniquement que des ensembles de donnée statiques, et ce tout au long des calculs. Cela vise à accélérer les calculs et ainsi réduire le temps d'exécution. Malheureusement, les calculs effectués sur des données statiques ne reflètent pas de manière appropriée la véritable nature dynamique des données. Dans le monde actuel, où les données circulent et changent constamment, le traitement sur données statiques (ou sur des données mises à jour qu'une seule fois par jour, voire moins souvent) nous fait perdre de précieuses informations découlant de la dynamique observée à une plus fine granularité, que ce soit au niveau des minutes, secondes, voire moins.

7.1 Contributions

L'objectif principal de cette thèse est de proposer une solution efficace pour le passage à l'échelle du calcul de l'algorithme des *K-plus proches voisins (KNN)* sur des machines isolées. Une solution appropriée doit être capable de traiter de grands ensembles de données dans un délai raisonnable, compte tenu des limites imposées par les ressources d'une machine.

La motivation derrière l'utilisation des machines isolées au lieu des systèmes distribués plus complexes, est la facilité d'accès à ce type de ressource de calcul et de son coût plus faible par rapport à celle des systèmes répartis.

Dans ce travail, nous cherchons également à construire une approche légère, capable de tirer parti des ressources limitées d'une machine. Ainsi, nous souhaitons proposer une approche mois coûteuse, mais non moins efficace. Une approche légère est une solution évidente pour effectuer le calcul en ligne de KNN, principalement en raison de sa capacité à fonctionner correctement en utilisant moins de ressources.

Enfin, nous proposons une solution efficace pour le traitement des mises à jour de données lors du calcul de KNN. Comme mentionné auparavant, l'état de l'art en KNN ne reflète pas correctement ni ne traite adéquatement la nature dynamique des données traitée par des algorithmes statiques.

7.1.1 *Pons*

Notre première contribution est *Pons* [24]: un algorithme *out-of-core* pour le calcul de KNN sur de grands ensembles de données qui ne rentrent pas en mémoire. Pour ce faire, *Pons* exploite efficacement à la fois le stockage sur disque dur et la mémoire disponible. Notre approche est capable de faire le calcul de KNN à un coût minimal, en stockant toutes les données sur le disque dur, et en chargeant ensuite ces données sur une section limitée de la mémoire pour pouvoir les traiter.

Nos expériences effectuées sur de grands ensembles de donnés montrent que *Pons* calcule le KNN en uniquemet 7% du temps requis par un calcul en mémoire. Notre évaluation montre la capacité de *Pons* au calcul de KNN sur des machines avec des contraintes de mémoire, ce qui est également une bonne solution pour le calcul de KNN en ligne, en consacrant peu de ressources à cette tâche spécifique.

7.1.2 *UpKNN*

Notre deuxième contribution est *UpKNN* [53]: une approche économe en mémoire et *multithread* passant efficacement à l'échelle, pour le traitement des mises à jour en temps réel dans les algorithmes KNN. *UpKNN* traite des millions de mises à jour en ligne, alors qu'il calcule encore le KNN efficacement sur des grands ensembles de données, en utilisant une unique machine isolée. Pour obtenir de bonnes performances, *UpKNN* réduit considérablement le nombre d'opérations sur disque dur effectuées lors du calcul, ce qui favorise la lecture et l'écriture de gros blocs de données à partir du disque.

Les résultats expérimentaux montrent la capacité de *UpKNN* de mettre à jour 100 millions d'éléments en environ 40 secondes, ce qui montre un passage à l'échelle efficace à la fois en nombre de mises à jour traitées et et en nombre de threads utilisés pour le calcul. Les expériences montrent également que *UpKNN* traite une moyenne de 3,2 millions de mises à jour par seconde, indiquant que notre approche est une bonne solution pour le traitement du KNN en ligne.

7.2 Perspectives

7.2.1 *Pons*

En ayant du recul sur *Pons*, nous observons l'utilisation d'une taille de profil fixe pour améliorer l'efficacité des opérations sur disque dur. Cette valeur fixe nous permet de connaître à l'avance la taille des blocs lus et écrits depuis et sur le disque, pour l'améliorer l'efficacité des requêtes sur disque pendant le calcul. Malheureusement, les ensembles de données utilisés dans les algorithmes KNN présentent rarement des tailles de profil identiques dans l'ensemble des profils. Par conséquent, cette hypothèse augmente l'empreinte en mémoire de l'algorithme en allouant de l'espace mémoire inutile pour des profils contenant moins d'éléments. Hélas, le cas contraire est également possible. Pour résoudre ce problème, nous proposons d'étudier la possibilité de maintenir une taille de profil fixe, mais qui peut être définie de manière dynamique en fonction de certaines propriétés de l'ensemble de données comme la taille de profil moyen. Cette définition dynamique conserve les avantages des tailles de bloc fixes, mais en adaptant cette taille en fonction des données.

Un autre point essentiel que nous observons dans *Pons* est la quantité de

données lues et écrites de multiple fois itération après itération. Notre approche lit et écrit un ensemble de fichiers à la fois pour les arrêtes entrantes et pour les arrêtes sortantes du graph à chaque itération de l'algorithme, même si les arrêtes ne changent pas d'une itération à une autre. Nous proposons d'étudier la faisabilité d'un méchanisme optimisé afin de lire et écrire uniquement lorsque les données changent entre des itérations consécutives. Un moyen efficace pour optimiser ces opérations consiste à marquer les données modifiées, et effectuer les opérations sur disque uniquement si nécessaire.

Un dernier ajustement possible pour *Pons* est d'optimiser l'algorithme de partitionnement. Dans *Pons*, nous avons implémenté un algorithme de partitionnement spécifique pour augmenter la probabilité d'assigner des membres voisins à la même partition. Une telle optimisation favorise les comparaisons des profils des entités dans la même partition, ce qui réduit les calculs inter-partition, qui sont beaucoup plus chers. Une amélioration possible sur cet algorithme est de favoriser également l'assignation des voisins des voisins dans la même partition.

7.2.2 *UpKNN*

Des améliorations peuvent être apportées à la phase de fusion de *UpKNN*, qui est consacré à incorporer les mises à jour dans les profils en utilisant un ensemble de threads pour effectuer cette opération. Chaque thread traite ensemble toute les mises à jour d'une partition, en chargeant des mises à jour et l'ensemble des profils d'une partition complètement en mémoire. Comme l'ensemble des threads effectuent la même opération au même moment, nous notons que l'utilisation mémoire totale peut dépasser les limites de la machine. Cela entraîne donc des accès multiples à la mémoire virtuelle, ce qui dégrade les performances. Par conséquent, nous suggérons d'améliorer cette opération en limitant la mémoire allouée, divisant de manière adéquate entre les threads, ce qui favorise la concurrence, autant que possible.

7.3 Future Work

Ayant à l'esprit la croissance rapide de la quantité de données, nous avons besoin d'augmenter la taille des ensembles de données testés sur nos contributions. L'objectif principal motivant cette thèse est de mettre à l'échelle du calcul sur des machines isolées, étant donné que cette configuration de calcul nous donne des ressources peu coûteuses mais aussi restreintes. Bien que nous

ayons prouvé qu'il est possible de mettre à l'échelle le calcul de KNN sur une machine isolée, ce passage à l'échelle a une limite. Pour aller au-delà, nous devons nécessairement penser à des systèmes répartis à grande échelle comme une option. Les deux algorithmes répartis ainsi que des approches basées sur le nuage nous permettent de passer à l'échelle le calcul consid'erablement. Bien que possible, nous devons aborder une séérie de questions bien connues dans les systèmes répartis, à savoir, la synchronisation, la cohérence, et la tolérance aux pannes, entre autres.

Une question clé dans le KNN réparti est l'algorithme de partitionnement. Le socle commun des algorithmes de KNN répartis est le fait que les données sont divisées, et ensuite traitées par plusieurs machines en parallèle. Pour optimiser les calculs de KNN, l'algorithme de partitionnement doit favoriser l'attribution des deux voisins, ainsi que les voisins des voisins dans la même machine. Une telle assignation favoriserait des calculs intra-machine plutôt que inter-machines, ce qui améliore les performances du calcul de KNN.

Dans le même environnement réparti, un autre objectif de notre travail futur est de trouver une représentation de profil amélioré. Un stockage décentralisé de profils doit être à la fois léger, et exact, étant donné que les profils sont transmis par le réseau lors de la comparaison des profils. En outre, une bonne représentation des profils devrait soutenir les mises à jour au fil du temps, étant donné que les profils sont traités par plusieurs machines, probablement géographiquement éloignés, le coût de la communication n'est pas négligeable.

Bibliography

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. Smola. Distributed large scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 37–48, New York, NY, USA, 2013. ACM

[2] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, T. Seidl, and others. Nearest neighbor classification in 3D protein databases. In *Proceedings of the International Conference on Intelligent Systems for Molecular Biology (ISMB)*, volume 99, pages 34–43, 1999

[3] M. Attique, H.-J. Cho, and T.-S. Chung. Core: Continuous monitoring of reverse K nearest neighbors on moving objects in road networks. In R. Lee, editor, *Computer and Information Science 2015*, pages 109–124. Springer International Publishing, Cham, 2016

[4] A.-L. Barabási, R. Albert, and H. Jeong. Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1-2):173–187, 1999

[5] M. Batko, C. Gennaro, P. Savino, and P. Zezula. Scalable similarity search in metric spaces. In *DELOS Workshop: Digital Library Architectures*, pages 213–224, 2004

[6] M. Batko, C. Gennaro, and P. Zezula. A scalable nearest neighbor search in P2P systems. In W. S. Ng, B.-C. Ooi, A. M. Ouksel, and C. Sartori, editors, *Databases, Information Systems, and Peer-to-Peer Computing: Second International Workshop (DBISP2P), Toronto, Canada, August 29-30, 2004, Revised Selected Papers*, pages 79–92. Springer Berlin Heidelberg, 2005

[7] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. The GOSSPLE anonymous social network. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*,

Middleware '10, pages 191–211, Berlin, Heidelberg, 2010. Springer-Verlag

[8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In C. Beeri and P. Buneman, editors, *Proceedings of the 7th International Conference on Database Theory (ICDT), Jerusalem, Israel, January 10–12*, pages 217–235. Springer Berlin Heidelberg, 1999

[9] C. Böhm and F. Krebs. The K-nearest neighbour join: Turbo charging the KDD process. *Knowledge and Information Systems*, 6(6):728–749, 2004

[10] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, June 2008

[11] A. Boutet, D. Frey, R. Guerraoui, A. Jegou, and A. M. Kermarrec. WHATSUP: A decentralized instant news recommender. In *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 741–752, May 2013

[12] A. Boutet, D. Frey, R. Guerraoui, A. Jégou, and A.-M. Kermarrec. Privacy-preserving distributed collaborative filtering. In G. Noubir and M. Raynal, editors, *Networked Systems: Second International Conference, NETYS 2014, Marrakech, Morocco, May 15-17, 2014. Revised Selected Papers*, pages 169–184. Springer International Publishing, Cham, 2014

[13] A. Boutet, D. Frey, R. Guerraoui, A.-M. Kermarrec, and R. Patra. HyRec: Leveraging browsers for scalable recommenders. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 85–96, New York, NY, USA, 2014. ACM

[14] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taïani. Being prepared in a sparse world: the case of KNN graph construction. In *IEEE International Conference on Data Engineering (ICDE)*, Helsinki, Finland, May 2016

[15] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proceedings of the International Conference on Weblogs and Social Media (ICWSM)*, pages 10–17, 2010

[16] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. StreamRec: A real-time recommender system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1243–1246, New York, NY, USA, 2011. ACM

[17] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 380–388, New York, NY, USA, 2002. ACM

[18] C. Chen, H. Yin, J. Yao, and B. Cui. TeRec: A temporal recommender system over tweet stream. *Proceedings of the VLDB Endowment*, 6(12): 1254–1257, August 2013

[19] J. Chen and H. Y. K. Lau. A reinforcement motion planning strategy for redundant robot arms based on hierarchical clustering and K-nearest-neighbors. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 727–732, Dec 2015

[20] J. Chen, H.-r. Fang, and Y. Saad. Fast approximate KNN graph construction for high dimensional data via recursive Lanczos bisection. *Journal of Machine Learning Research*, 10:1989–2012, December 2009

[21] Q. Chen, D. Li, and C. K. Tang. KNN matting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(9):2175–2188, Sept 2013

[22] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*, pages 1123–1126, New York, NY, USA, 2010. ACM

[23] N. Chiluka, A.-M. Kermarrec, and J. Olivares. Scaling KNN computation over large graphs on a PC. In *Proceedings of the Posters & Demos Session*, Middleware Posters and Demos '14, pages 9–10, New York, NY, USA, 2014. ACM

[24] N. Chiluka, A.-M. Kermarrec, and J. Olivares. The out-of-core KNN awakens: The light side of computation force on large datasets. In A. P. Abdulla and C. Delporte-Gallet, editors, *Networked Systems: 4th International Conference, NETYS 2016, Marrakech, Morocco, May 18-20, 2016, Revised Selected Papers*, pages 295–310. Springer International Publishing, 2016

[25] M. Connor and P. Kumar. Parallel construction of K-nearest neighbor graphs for point clouds. In *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics (SPBG)*, pages 25–31, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association

[26] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967

[27] R. Dash and P. K. Dash. A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science*, 2(1):42 – 57, 2016

[28] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SCG)*, pages 253–262, New York, NY, USA, 2004. ACM

[29] A. M. de Souza, R. S. Yokoyama, G. Maia, A. Loureiro, and L. Villas. Real-time path planning to prevent traffic jam through an intelligent transportation system. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 726–731, June 2016

[30] T. Debatty, P. Michiardi, O. Thonnard, and W. Mees. Building KNN graphs from large text data. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 573–578, Oct 2014

[31] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970

[32] W. Dong, C. Moses, and K. Li. Efficient K-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 577–586, New York, NY, USA, 2011. ACM

[33] M. D. Ekstrand, J. T. Riedl, J. A. Konstan, and others. Collaborative filtering recommender systems. *Foundations and Trends in Human–Computer Interaction*, 4(2):81–173, 2011

[34] K. El-Maleh, M. Klein, G. Petrucci, and P. Kabal. Speech/music discrimination for multimedia applications. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 6, pages 2445–2448 vol.4, 2000

[35] P. Franti, O. Virmajoki, and V. Hautamaki. Fast agglomerative clustering using a K-nearest neighbor graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1875–1881, Nov 2006

[36] I. Fujinaga and K. MacMillan. Realtime recognition of orchestral instruments. In *Proceedings of the international computer music conference*, volume 141, page 143, 2000

[37] V. Garcia, E. Debreuve, and M. Barlaud. Fast K nearest neighbor search using GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) 2008*, pages 1–6, June 2008

[38] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760, Sept 2010

[39] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Databases (VLDB)*, volume 99, pages 518–529, 1999

[40] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer. KNN model-based approach in classification. In R. Meersman, Z. Tari, and D. C. Schmidt, editors, *Proceedings On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE, Catania, Sicily, Italy, November 3-7, 2003.*, pages 986–996. Springer Berlin Heidelberg, 2003

[41] P. Haghani, S. Michel, P. Cudré-Mauroux, and K. Aberer. LSH at large-distributed KNN search in high dimensions. In *Proceedings of the 11th International Workshop on Web and Databases (WebDB)*, 2008

[42] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with K-nearest neighbor graph. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 22, page 1312, 2011

[43] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 77–85, New York, NY, USA, 2013. ACM

[44] F. M. Harper and J. A. Konstan. The MovieLens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):19:1–19:19, December 2015

[45] V. Hautamäki, I. Kärkkäinen, and P. Fränti. Outlier detection using K-nearest neighbour graph. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR) 2004*, pages 430–433, 2004

[46] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Proceedings of the IEEE/ACM Conference Supercomputing (SC95), 1995*, pages 28–28, 1995

[47] K. Huang, S. Li, X. Kang, and L. Fang. Spectral–spatial hyperspectral image classification based on KNN. *Sensing and Imaging*, 17(1):1–13, 2015

[48] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011

[49] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), August 2007

[50] K. Kapoor, V. Kumar, L. Terveen, J. A. Konstan, and P. Schrater. "I like to explore sometimes": Adapting to dynamic user novelty preferences. In *Proceedings of the 9th ACM Conference on Recommender Systems (RecSys)*, pages 19–26, New York, NY, USA, 2015. ACM

[51] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD)*, pages 369–380, New York, NY, USA, 1997. ACM

[52] E. Keogh and A. Mueen. Curse of dimensionality. In C. Sammut and G. I. Webb, editors, *Encyclopedia of Machine Learning*, pages 257–258. Springer US, Boston, MA, 2010

[53] A.-M. Kermarrec, N. Mittal, and J. Olivares. Multithreading approach to process real-time updates in KNN algorithms. In *(UNDER REVISION) 2017 25th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages XXX–XXX, March 2017

[54] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 169–182, New York, NY, USA, 2013. ACM

[55] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006

[56] P. Knees and M. Schedl. Semantic labeling of music. In *Music Similarity and Retrieval: An Introduction to Audio- and Web-based Strategies*, pages 85–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016

[57] Y. Koren. Collaborative filtering with temporal dynamics. *Communications of the ACM*, 53(4):89–97, April 2010

[58] P. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast and effective retrieval of medical tumor shapes. *IEEE Transactions on Knowledge and Data Engineering*, 10(6):889–904, Nov 1998

[59] S. Koziel, L. Leifsson, M. Lees, V. V. Krzhizhanovskaya, J. Dongarra, P. M. Sloot, B. Jiang, and Y. Sha. Modeling temporal dynamics of user interests in online social networks. *Procedia Computer Science*, 51:503 – 512, 2015

[60] Q. Kuang and L. Zhao. A practical GPU based KNN algorithm. In *International symposium on computer science and computational technology (ISCSCT)*, pages 151–155. Citeseer, 2009

[61] L. I. Kuncheva. Genetic algorithms editing for the K-nearest neighbors rule by a genetic algorithm. *Pattern Recognition Letters*, 16(8):809 – 814, 1995

[62] A. Kyrola. DrunkardMob: Billions of random walks on just a PC. In *Proceedings of the 7th ACM Conference on Recommender Systems (RecSys)*, pages 257–264, New York, NY, USA, 2013. ACM

[63] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX

[64] A. Kyrola and C. Guestrin. GraphChi-DB: Simple design for a scalable graph database system on just a PC. *arXiv preprint arXiv:1403.0701*, abs/1403.0701, 2014

[65] S. Larrain, C. Trattner, D. Parra, E. Graells-Garrido, and K. Nørvåg. Good times bad times: A study on recency effects in collaborative filtering for social tagging. In *Proceedings of the 9th ACM Conference on Recommender Systems (RecSys)*, pages 269–272, New York, NY, USA, 2015. ACM

[66] N. Lathia, S. Hailes, and L. Capra. kNN CF: A temporal social network. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys)*, pages 227–234, New York, NY, USA, 2008. ACM

[67] N. Lathia, S. Hailes, L. Capra, and X. Amatriain. Temporal diversity in recommender systems. In *Proceedings of the 33rd International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 210–217, New York, NY, USA, 2010. ACM

[68] L. Lee. Measures of distributional similarity. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics (ACL)*, pages 25–32, Stroudsburg, PA, USA, 1999. Association for Computational Linguistics

[69] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014

[70] S. Liang, Y. Liu, C. Wang, and L. Jian. A CUDA-based parallel implementation of K-nearest neighbor algorithm. In *The International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 291–296, Oct 2009

[71] S. Liang, C. Wang, Y. Liu, and L. Jian. CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU. In *IEEE Youth Conference on Information, Computing and Telecommunication (YC-ICT)*, pages 415–418, Sept 2009

[72] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang. MMap: Fast billion-scale graph computation on a PC via memory mapping. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 159–164, Oct 2014

[73] A. T. Lora, J. M. R. Santos, A. G. Exposito, J. L. M. Ramos, and J. C. R. Santos. Electricity market price forecasting based on weighted nearest neighbors techniques. *IEEE Transactions on Power Systems*, 22(3):1294–1301, Aug 2007

[74] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8): 716–727, April 2012

[75] E. H. C. Lu, H. S. Chen, and V. S. Tseng. Efficient approaches for multi-requests route planning in urban areas. In *2013 IEEE 14th International Conference on Mobile Data Management*, volume 1, pages 36–45, June 2013

[76] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007

[77] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*, pages 135–146, New York, NY, USA, 2010. ACM

[78] R. E. McRoberts, M. D. Nelson, and D. G. Wendt. Stratified estimation of forest area using satellite imagery, inventory data, and the K-nearest neighbors technique. *Remote Sensing of Environment*, 82(2-3):457 – 468, 2002

[79] O. Nasraoui, J. Cerwinske, C. Rojas, and F. Gonzalez. Performance of recommendation systems in dynamic streaming environments. In *Proceedings of the 2007 SIAM International Conference on Data Mining (SDM)*, chapter 63, pages 569–574. 2007

[80] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[81] J. Pan and D. Manocha. Bi-level locality sensitive hashing for K-nearest neighbor computation. In *2012 IEEE 28th International Conference on Data Engineering*, pages 378–389, April 2012

[82] R. Paredes and E. Chávez. Using the K-nearest neighbor graph for proximity searching in metric spaces. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE), Buenos Aires, Argentina, November 2-4*, pages 127–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005

[83] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of K-nearest neighbor graphs in metric spaces. In C. Álvarez and M. Serna, editors, *Proceedings of the 5th International Workshop Experimental Algorithms (WEA), Cala Galdana, Menorca, Spain, May 24-27*, pages 85–97. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006

[84] Y. Park, S. Park, W. Jung, and S. goo Lee. Reversed CF: A fast collaborative filtering algorithm using a K-nearest neighbor graph. *Expert Systems with Applications*, 42(8):4022 – 4028, 2015

[85] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society

[86] E. Plaku and L. Kavraki. Distributed computation of the KNN graph for large high-dimensional point sets. *Journal of Parallel and Distributed Computing*, 67(3):346–359, 2007

[87] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *Proceedings of the 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 51–60, Washington, DC, USA, 2013. IEEE Computer Society

[88] C. Rana and S. Jain. A study of dynamic features of recommender systems. artificial intelligent review, 2012

[89] J. L. Rodas, J. Olivares, J. A. Galindo, and D. Benavides. Hacia el uso de sistemas de recomendación en sistemas de alta variabilidad. In *Congreso Español de Informática (CEDI)*, 2016

[90] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 410–424, New York, NY, USA, 2015. ACM

[91] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, New York, NY, USA, 2013. ACM

[92] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web (WWW)*, pages 285–295, New York, NY, USA, 2001. ACM

[93] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In S. E. Nikoletseas, editor, *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA), Santorini Island, Greece, May 10-13*, pages 606–609, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg

[94] M. A. Schuh, T. Wylie, and R. A. Angryk. Mitigating the curse of dimensionality for exact KNN retrieval. In *FLAIRS Conference*, 2014

[95] Z. F. Siddiqui and M. Spiliopoulou. Combining multiple interrelated streams for incremental clustering. In M. Winslett, editor, *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM), New Orleans, LA, USA, June 2-4*, pages 535–552. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009

[96] Z. F. Siddiqui, E. Tiakas, P. Symeonidis, M. Spiliopoulou, and Y. Manolopoulos. xStreams: Recommending items to users with time-evolving preferences. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS)*, pages 22:1–22:12, New York, NY, USA, 2014. ACM

[97] N. Sismanis, N. Pitsianis, and X. Sun. Parallel search of K-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–6, Sept 2012

[98] G. Song, J. Rochas, F. Huet, and F. Magoules. Solutions for processing K nearest neighbor joins for massive data on MapReduce. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 279–287, March 2015

[99] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, New York, NY, USA, 2001. ACM

[100] A. Stupar, S. Michel, and R. Schenkel. Rankreduceprocessing K-nearest neighbor queries on top of MapReduce. In *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR)*, pages 13–18, 2010

[101] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6(14):1930–1941, September 2013

[102] S. Tan. Neighbor-weighted K-nearest neighbor for unbalanced text corpus. *Expert Systems with Applications*, 28(4):667 – 671, 2005

[103] S. Tan. An effective refinement strategy for KNN text classifier. *Expert Systems with Applications*, 30(2):290 – 298, 2006

[104] M. Tkalcic, A. Odic, A. Kosir, and J. Tasic. Affective labeling in a content-based recommender system for images. *IEEE Transactions on Multimedia*, 15(2):391–400, Feb 2013

[105] K. Trohidis, G. Tsoumakas, G. Kalliris, and I. P. Vlahavas. Multi-label classification of music into emotions. In *Proceedings of the 9th International Conference on Music Information Retrieval (ISMIR)*, volume 8, pages 325–330, 2008

[106] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 333–342, New York, NY, USA, 2014. ACM

[107] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 507–516, New York, NY, USA, 2013. ACM

[108] E. Valari and A. N. Papadopoulos. Continuous similarity computation over streaming graphs. In *Machine Learning and Knowledge Discovery in Databases*, pages 638–653. Springer, 2013

[109] K. Verstrepen and B. Goethals. Unifying nearest neighbors collaborative filtering. In *Proceedings of the 8th ACM Conference on Recommender Systems (RecSys)*, pages 177–184, New York, NY, USA, 2014. ACM

[110] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001

[111] J. S. Vitter. External memory algorithms. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 359–416. Springer US, Boston, MA, 2002

[112] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable KNN graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1106–1113, June 2012

[113] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3360–3367, June 2010

[114] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering*, pages 568–579, March 2014

[115] L. Wang, R. Gopal, R. Shankar, and J. Pancras. On the brink: Predicting business failure with mobile location-based checkins. *Decision Support Systems*, 76:3 – 13, 2015.

[116] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis. Secure KNN computation on encrypted databases. In *Proceedings of the 2009 ACM International Conference on Management of Data (SIGMOD)*, pages 139–152, New York, NY, USA, 2009. ACM

[117] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008

[118] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for KNN join processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 756–767. VLDB Endowment, 2004

[119] Z. Xiaowei, H. Wentao, and C. Wenguang. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association

[120] N. Xu, L. Chen, and B. Cui. LogGP: A log-based dynamic graph partitioning method. *Proceedings of the VLDB Endowment*, 7(14):1917–1928, October 2014

[121] Y. Xu, X. Ning, X. Gao, and F. Yu. Quality and safety news topic tracking algorithm based on improved K-nearest neighbor. In *Proceedings of The fourth International Conference on Information Science and Cloud Computing (ISCC2015). 18-19 December. Guangzhou, China*, 2015

[122] J. Yang, W. Li, S. Wang, J. Lu, and L. Zou. Classification of children with attention deficit hyperactivity disorder using pca and K-nearest neighbors during interference control task. In R. Wang and X. Pan, editors, *Proceedings of the Fifth International Conference on Cognitive Neurodynamics: Advances in Cognitive Neurodynamics (V)*, pages 447–453. Springer Singapore, Singapore, 2016

[123] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 42–49, New York, NY, USA, 1999. ACM

[124] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning (ICML)*, volume 97, pages 412–420, 1997

[125] C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based KNN join processing for high-dimensional data. *Information and Software Technology*, 49(4):332 – 344, 2007

[126] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[127] C. Yu, R. Zhang, Y. Huang, and H. Xiong. High-dimensional KNN joins with incremental updates. volume 14, pages 55–82, 2009

[128] C. Zhang, F. Li, and J. Jestes. Efficient parallel KNN joins for large data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 38–49, New York, NY, USA, 2012. ACM

[129] W. E. Zhang, Q. Z. Sheng, Y. Qin, L. Yao, A. Shemshadi, and K. Taylor. SECF: Improving SPARQL querying performance with proactive fetching and caching. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, pages 362–367, New York, NY, USA, 2016. ACM

[130] Y. F. Zhao, F. Y. Wang, H. Gao, F. H. Zhu, Y. S. Lv, and P. J. Ye. Content-based recommendation for traffic signal control. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 1183–1188, Sept 2015

[131] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Architecture-aware graph repartitioning for data-intensive scientific computing. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 78–85, Oct 2014

[132] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. Priebe, and A. Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, 2015. USENIX Association

Abstract

The *K-Nearest Neighbors (KNN)* is an efficient method to find similar data among a large set of it. Over the years, a huge number of applications have used *KNN*'s capabilities to discover similarities within the data generated in diverse areas such as business, medicine, music, and computer science. Despite years of research have brought several approaches of this algorithm, its implementation still remains a challenge, particularly today where the data is growing at unthinkable rates. In this context, running *KNN* on large datasets brings two major issues: huge memory footprints and very long runtimes. Because of these high costs in terms of computational resources and time, *KNN* state-of-the-art works do not consider the fact that data can change over time, assuming always that the data remains static throughout the computation, which unfortunately does not conform to reality at all.

In this thesis, we address these challenges in our contributions. Firstly, we propose an *out-of-core* approach to compute *KNN* on large datasets, using a commodity single PC. We advocate this approach as an inexpensive way to scale the *KNN* computation compared to the high cost of a distributed algorithm, both in terms of computational resources as well as coding, debugging and deployment effort. Secondly, we propose a multithreading *out-of-core* approach to face the challenges of computing *KNN* on data that changes rapidly and continuously over time.

After a thorough evaluation, we observe that our main contributions address the challenges of computing the *KNN* on large datasets, leveraging the restricted resources of a single machine, decreasing runtimes compared to that of the baselines, and scaling the computation both on static and dynamic datasets.

Résumé

La technique des *K-plus proches voisins* (K-Nearest Neighbors (KNN) en Anglais) est une méthode efficace pour trouver des données similaires au sein d'un grand ensemble de données. Au fil des années, un grand nombre d'applications ont utilisé les capacités du KNN pour découvrir des similitudes dans des jeux de données de divers domaines tels que les affaires, la médecine, la musique, ou l'informatique. Bien que des années de recherche aient apporté plusieurs approches de cet algorithme, sa mise en œuvre reste un défi, en particulier aujourd'hui alors que les quantités de données croissent à des vitesses inimaginables. Dans ce contexte, l'exécution du KNN sur de grands ensembles pose deux problèmes majeurs: d'énormes empreintes mémoire et de très longs temps d'exécution. En raison de ces coût élevés en termes de ressources de calcul et de temps, les travaux de l'état de l'art ne considèrent pas le fait que les données peuvent changer au fil du temps, et supposent toujours que les données restent statiques tout au long du calcul, ce qui n'est malheureusement pas du tout conforme à la réalité.

Nos contributions dans cette thèse répondent à ces défis. Tout d'abord, nous proposons une approche *out-of-core* pour calculer les KNN sur de grands ensembles de données en utilisant un seul ordinateur. Nous préconisons cette approche comme un moyen moins coûteux pour faire passer à l'échelle le calcul des KNN par rapport au coût élevé d'un algorithme distribué, tant en termes de ressources de calcul que de temps de développement, de débogage et de déploiement. Deuxièmement, nous proposons une approche *out-of-core multithreadée* (i.e. utilisant plusieurs fils d'exécution) pour faire face aux défis du calcul des KNN sur des données qui changent rapidement et continuellement au cours du temps.

Après une évaluation approfondie, nous constatons que nos principales contributions font face aux défis du calcul des KNN sur de grands ensembles de données, en tirant parti des ressources limitées d'une machine unique, en diminuant les temps d'exécution par rapport aux performances actuelles, et en permettant le passage à l'échelle du calcul, à la fois sur des données statiques et des données dynamiques.

VU:

Le Directeur de Thèse

VU:

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

David ALIS

VU après soutenance pour autorisation de publication:

Le Président de Jury,