The Out-of-core KNN Awakens: The Light Side of Computation Force on Large Datasets

Nitin Chiluka, Anne-Marie Kermarrec, and Javier Olivares(✉)

Inria, Rennes, France
nitin.chiluka@gmail.com,
{anne-marie.kermarrec,javier.olivares}@inria.fr

Abstract. K-Nearest Neighbors (KNN) is a crucial tool for many applications, e.g. recommender systems, image classification and web-related applications. However, KNN is a resource greedy operation particularly for large datasets. We focus on the challenge of KNN computation over large datasets on a single commodity PC with limited memory. We propose a novel approach to compute KNN on large datasets by leveraging both disk and main memory efficiently. 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.

Keywords: K-nearest neighbors · Out-of-core computation · Graph processing

1 Introduction

K-Nearest Neighbors (KNN) is a widely-used algorithm for many applications such as recommender systems [3–5]; information retrieval [8,13,21] in supporting similarity and proximity on stored data; and image classification [2,17,20]: finding similar images among a set of them. Generally, KNN is used for finding similar entities in a large set of candidates, by computing similarity between entities' profiles.

Although the algorithm has been well studied, the computation of KNN on large datasets remains a challenge. Large-scale KNN processing is computationally expensive, requiring a large amount of memory for efficient in-memory computation. The memory requirements of the current datasets (spanning even trillions of edges) is enormous, beyond terabytes. Such memory requirements are often unaffordable. In such scenario, one can think of an *out-of-core* computation as an option. Recent works [11,14,16,19,22] have shown that such approaches perform well on data that cannot be completely stored in memory.

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

© Springer International Publishing AG 2016
P.A. Abdulla and C. Delporte-Gallet (Eds.): NETYS 2016, LNCS 9944, pp. 295–310, 2016.
DOI: 10.1007/978-3-319-46140-3_24

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* [5], 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 [11,14,19] 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 [14,19]. 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.

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 [5]. 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:

$$AD^{(t)} = \frac{\sum_{u \in V} \sum_{v \in B_u} Dist(F_u, F_v)}{NK} \quad (1)$$

$Dist(F_u, F_v)$ 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.

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 1 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 upto $O(2K+K^2)$ 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 $K(\approx 10-20)$ [5]. 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 1. 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              Initialize heap TopK
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)
```

3 *Pons*

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 (Sect. 2.1) consumes a significant amount of memory. In this section, we propose *Pons*[1], an out-of-core approach which aims to address this trade-off.

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

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.

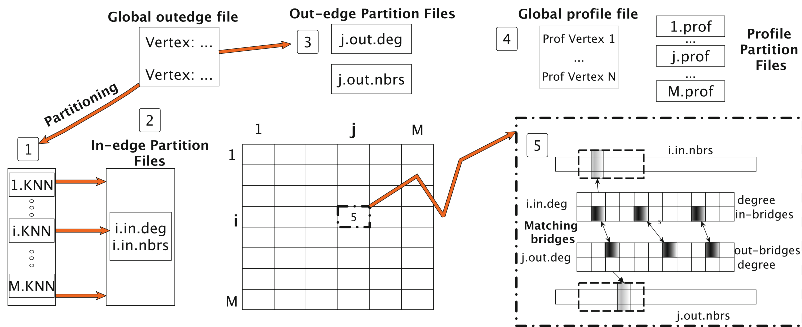


Fig. 1. *Pons* executes 5 phases: (1) Partitioning, (2) In-Edge Partition Files, (3) Out-Edge Partition Files, (4) Profile Partition Files, and (5) Distance Computation

We now present a brief overview of our approach, as illustrated in Algorithm 2, and Fig. 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 I is to compute K-closest neighbors for each vertex. To do so, iteration I executes 5 phases (Algorithm 2, lines 2–8). First phase divides the vertices 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.

Algorithm 2. *Pons*

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

```
1 begin
2     foreach Iteration I 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 note here that the choice of the number of partitions (M) depends on factors such as the memory limit (X_{limit}), the number of nodes (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 $I+1$. 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.

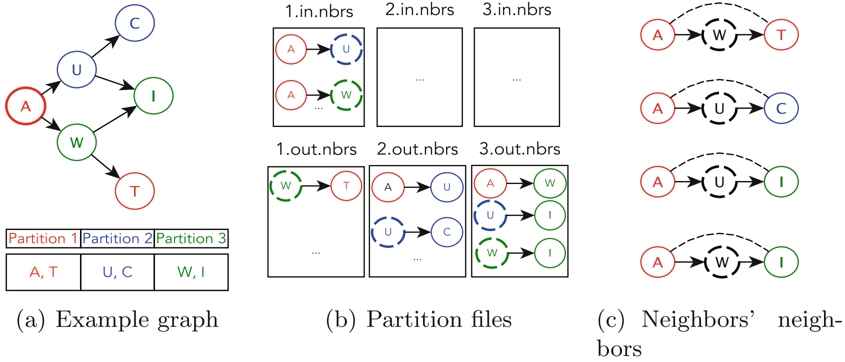
As *Pons* is designed to accommodate the profiles of only two partitions at a time in memory, *Pons* adopts the following approach for each partition i. First, it loads into memory i's profile as well as the bridge vertices of i's in-edge partition file. Next, an empty heap is allocated for each vertex 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 $I + 1$.

An illustrative example. Figure 2(a) 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 2(b) 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 Fig. 2(c), vertex A finds its neighbors' neighbor I, via bridge vertices U and W.

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



(a) Example graph (b) Partition files (c) Neighbors' neighbors

Fig. 2. [Best viewed in color.] (a) A's out-neighbors and A's neighbors' neighbors. (b) In-edge partition files and out-edge partition files. (c) A's neighbors' neighbors found using bridge vertices

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.

When a vertex v is assigned to a 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 vertices assigned to the partition j and their respective out-neighbors.

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 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 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, which could belong to any partition, and (ii) the number of b's in-neighbors that 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.

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, which could belong to any partition, and (ii) the number of b's out-neighbors that 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 Sect. 4.5) for the KNN computation.

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 is read from the global profile file, and then written to the profile partition file corresponding to the partition that it was assigned. 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 in the Phase 5, as it facilitates sequential reading of the entire file without any random disk operations.

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 3 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. $InProf$ stores the profiles of vertices (W_i) in partition i read from the `i.prof` file (line 3). $InBrid$ stores the bridge vertices and their corresponding number of in-neighbors in partition i read from the `i.in.deg` file (line 4). $HeapTopK$ is a heap, which is initially empty (line 5), stores the scores and ids of the K-closest neighbors for each vertex $v \in W_i$, and tuple T stores neighbors, neighbors' neighbors, and random neighbors' tuples for distance computation.

For computing the new KNN for each vertex $s \in W_i$, partitions 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 parallel, all ordered tuples of neighbors' neighbors as follows:

$$(s, d) \,|\, s \in W_i, d \in W_j, (s, b) \in E^{(t)}, (b, d) \in E^{(t)}, b \in (InBrid \cap OutBrid) \quad (2)$$

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 $(w|\, w \in B_v \cap W_j)$ 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 $(HeapTopK[s])$ of the source vertex s is updated with d's id and the computed distance score $(Dist(F_s, F_d))$.

Algorithm 3. $NNComputation()$: Neighbors' neighbors computation

Data: In-edge partition files, Out-edge partition files, Profiles
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 $HeapTopK[W_i] \leftarrow \phi$
6 **foreach** *(Out-edge) Partition* j **do**
7 Read $OutProf$ from File(`j.prof`)
8 Read $OutBrid \leftarrow$ from File(`j.out.deg`)
9 Initialize tuple $T \leftarrow \phi$
10 $CndBrid \leftarrow (InBrid \cap OutBrid) \cup (W_i \cap OutBrid)$
11 **foreach** $Bridge\ b \in CndBrid$ **do**
12 **in parallel**
13 $Src \leftarrow$ ReadInNeig($i.in.nbrs, b$)
14 $Dst \leftarrow$ ReadOutNeig($j.out.nbrs, b$)
15 AddTuples(T,$Src \times Dst$)
16 **foreach** $(s, d) \in T$ **do**
17 **in parallel**
18 $dist \leftarrow$ Dist(F_s, F_d)
19 UpdateHeap($HeapTopK[s], d, dist$)
20 **foreach** $s \in W_i$ **do**
21 **in parallel**
22 $Dst \leftarrow Rnd(K) \in W_j$
23 Compute tuples $s \times Dst$
24 Update $HeapTopK[s]$ as above
25 File($G^{(t+1)}$).Write($HeapTopK$)

5 Experimental Setup

We perform our experiments on a Apple MacBook Pro laptop, 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) SSD.

Datasets. We evaluate *Pons* on both sparse- and dense- dimensional datasets. For sparse datasets, we use Friendster [15] and Twitter data[2]. 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) [12] which has vectors of 128 dimensions each. Vertices represent high-dimensional vectors and their profiles represent SIFT descriptors. The SIFT descriptors are typically high dimensional feature vectors used in identifying objects in computer vision (Table 1).

Table 1. Datasets

Dataset	Vertices	P	K	VI[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

Performance. 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 measured by the maximum memory footprint observed during the execution of the algorithm. Thus, we use maximum resident set size (RSS) and virtual memory size (VI).

6 Evaluation

We evaluate the performance of *Pons* on large datasets that do not fit in memory. 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.

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 2), respecting the maximum available memory of the machine. For this experiment both approaches run on 8 threads.

Execution Time. In Table 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

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

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*.

Table 2. Relative performance comparing *Pons* and INM, and memory footprint

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

Memory Consumption. As we show in Table 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.

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 3 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*'s 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.

6.3 Performance for 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 shows the percentage of

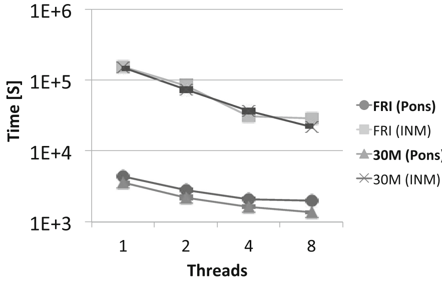


Fig. 3. Impact of multithreading

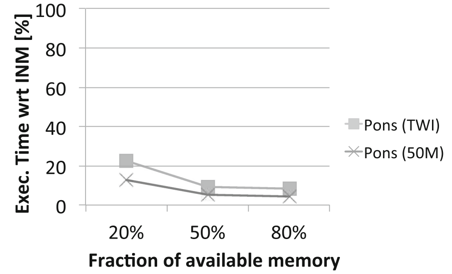


Fig. 4. Impact of the available memory

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 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.

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 IO 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.

Execution Time. We evaluate the performance of *Pons* for different number of partitions. Figures 5 and 6 show the runtime for the optimal value, and two suboptimal values of M. The smaller suboptimal value of M causes larger runtimes 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.

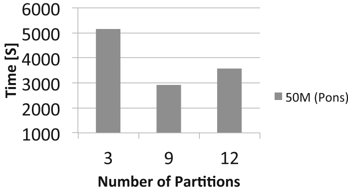


Fig. 5. Runtime: The impact of M

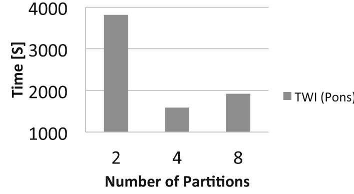


Fig. 6. Runtime: The impact of M

Memory Consumption. Figures 7 and 8 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 suboptimally. 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.

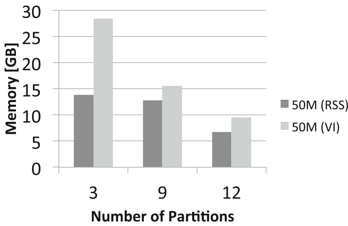


Fig. 7. The impact of M

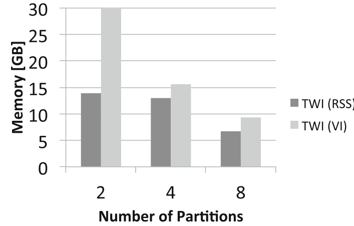


Fig. 8. The impact of M

7 Related Work

The problem of finding K-nearest neighbors has been well studied over last years. Multiple techniques have been proposed to perform this computation efficiently: branch and bound algorithms [10]; trees [1,18]; divide and conquer methods [6]; graph-based algorithms [9]. However, only a few have performed KNN computation in memory-constrained environments [7].

Recently, many studies [11,14,19] have explored 'out-of-core' mechanisms to process large graphs on a single commodity PC. Kyrola et al. in [14] propose *GraphChi*, a disk-based system to compute graph algorithms on large datasets. They present a sliding window computation method for processing a large graph from disk. This system is highly efficient on graphs that remain static during

the entire computation. Unfortunately, it does not show same efficiency when the graph changes over time, as the case of KNN computation. *X-Stream* [19] proposes a edge-centric graph processing system on a single shared-memory machine. Graph algorithms are performed leveraging streaming partitions, and processing sequentially edges and vertices from disk. *TurboGraph* [11] consists of a *pin-and-slide*, a parallel execution model for computing on large-scale graphs using a single machine. *Pin-and-slide* model divides the set of vertices in a list of pages, where each vertex could have several pages.

8 Conclusion

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 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.

Acknowledgments. This work was partially funded by Conicyt/Beca Doctorado en el Extranjero Folio 72140173 and Google Focused Award Web Alter-Ego.

References

1. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: ICML (2006)
2. Boiman, O., Shechtman, E., Irani, M.: In defense of nearest-neighbor based image classification. In: CVPR (2008)
3. Boutet, A., Frey, D., Guerraoui, R., Jegou, A., Kermarrec, A.M.: WHATSUP: a decentralized instant news recommender. In: IPDPS (2013)
4. Boutet, A., Frey, D., Guerraoui, R., Jegou, A., Kermarrec, A.M.: Privacy-preserving distributed collaborative filtering. In: Noubir, G., Raynal, M. (eds.) Networked Systems. LNCS, vol. 8593, pp. 169–184. Springer, Heidelberg (2014)
5. Boutet, A., Frey, D., Guerraoui, R., Kermarrec, A.M., Patra, R.: HyRec: Leveraging browsers for scalable recommenders. In: Middleware (2014)
6. Chen, J., Fang, H.R., Saad, Y.: Fast approximate KNN graph construction for high dimensional data via recursive Lanczos bisection. J. Mach. Learn. Res. **10**, 1989–2012 (2009)
7. Chiluka, N., Kermarrec, A.M., Olivares, J.: Scaling KNN computation over large graphs on a PC. In: Middleware (2014)
8. Debatty, T., Michiardi, P., Thonnard, O., Mees, W.: Building k-nn graphs from large text data. In: Big Data (2014)

9. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: WWW (2011)
10. Fukunaga, K., Narendra, P.M.: A branch and bound algorithm for computing k-nearest neighbors. IEEE Trans. Comput. **C–24**(7), 750–753 (1975)
11. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: SIGKDD (2013)
12. Jégou, H., Tavenard, R., Douze, M., Amsaleg, L.: Searching in one billion vectors: re-rank with source coding. In: ICASSP (2011)
13. Katayama, N., Satoh, S.: The SR-tree: An index structure for high-dimensional nearest neighbor queries. In: SIGMOD, vol. 26, pp. 369–380. ACM (1997)
14. Kyrola, A., Blelloch, G.E., Guestrin, C.: GraphChi: Large-scale graph computation on just a PC. In: OSDI (2012)
15. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
16. Lin, Z., Kahng, M., Sabrin, K., Chau, D., Lee, H., Kang, U.: MMAP: fast billion-scale graph computation on a PC via memory mapping. In: Big Data (2014)
17. McRoberts, R.E., Nelson, M.D., Wendt, D.G.: Stratified estimation of forest area using satellite imagery, inventory data, and the k-nearest neighbors technique. Remote Sens. Environ. **82**(2), 457–468 (2002)
18. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD (1995)
19. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: edge-centric graph processing using streaming partitions. In: SOSP (2013)
20. Wang, J., Yang, J., Yu, K., Lv, F., Huang, T., Gong, Y.: Locality-constrained linear coding for image classification. In: CVPR (2010)
21. Wong, W.K., Cheung, D.W.l., Kao, B., Mamoulis, N.: Secure KNN computation on encrypted databases. In: SIGMOD (2009)
22. Zhu, X., Han, W., Chen, W.: GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: USENIX ATC (2015)