Multithreading Approach to Process Real-Time Updates in KNN Algorithms

Anne-Marie Kermarrec, Nupur Mittal, and Javier Olivares(✉)

Inria Rennes, Rennes, France
{anne-marie.kermarrec,nupur.mittal,javier.olivares}@inria.fr

Abstract. K-Nearest Neighbors algorithm (KNN) is the core of a considerable amount of online services and applications, like recommendation engines, content-classifiers, information retrieval systems, etc. The users of these services change their preferences over time, aggravating the computational challenges of KNN. In this work, we present *UpKNN*: an efficient *thread-based* out-of-core approach to take the updates of users preferences into account while it computes the KNN efficiently.

1 Introduction

K-Nearest Neighbors (KNN) has been one of the most important classification techniques, specially used on recommender systems [1,2], and information retrieval applications.

KNN is a process of finding the most similar neighbors of a node/entity from a dataset. Each node of the dataset is represented for some data, commonly known as *profile*. We consider two data entities as neighbors if their profiles are similar based on a similarity metric as cosine or Jaccard.

Unfortunately, the main bottleneck of KNN is its huge memory requirements. Besides, some of the KNN applications witness high rate of changes in profiles over time, making very difficult to take these changes into account. These updates only increase the computation time considerably, making the algorithm less and less scalable. Due to this cost, many current approaches [2–4] simplify the processing assuming the dataset remains static throughout the computation. Consequently, the computation of KNN on static datasets does not consider data's dynamism, relying on content that is always outdated. Unfortunately, nowadays data changes continuously [7,8] at unimaginable rates, specially on those web-based, or recommendation systems' applications [5,6].

Hence, we propose *UpKNN*, a multithreading approach for processing real-time profile updates in KNN algorithms. *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 use of a single commodity PC, instead of a more complex computing platform, is motivated by its lower cost and ease of access for a vast majority of potential users, compared to a distributed system.

A. El Abbadi and B. Garbinato (Eds.): NETYS 2017, LNCS 10299, pp. 109–114, 2017.
DOI: 10.1007/978-3-319-59647-1_9

2 Background

Given N entities with their profiles in a D-dimensional space, the K-Nearest Neighbors (KNN) algorithm finds the K closest neighbors for each entity. The distance between two entities is computed based on a well-defined metric that takes into account their profiles. To compute KNN efficiently we adopt an approximate approach as proposed in [1].

Let us consider a set of entities U ($|U| = N$), associated with a set of items denoted by I. Each entity u has a profile UP_u, composed of items in I. *UpKNN* assumes that the N entities are randomly partitioned into M partitions, in such a way that at least one partition can be processed in memory at a time.

Corresponding to each of the M partitions there is a partition file PF_j in disk, storing the profiles UP of all the entities belonging to partition j.

To update the profiles, *UpKNN* receives an unsorted set of updates S consisting of entity-item tuples: $S = \{< u, i > \mid u \in U, i \in I\}$.

3 *UpKNN* Algorithm

3.1 Classify

UpKNN classifies each update of the set S per its entity's partition, such that all updates for the entities of a partition are applied at once, avoiding further IO operations. For a fast classification, we use a set of in-memory buffers, which are read and written in parallel. *UpKNN* performs the expensive read operations from S (on disk) in parallel with the classification, achieving a higher throughput.

Figure 1 depicts the *classify* phase. The classification separates the updates and stores them into M update files, UF_m. To do so, we have pairs of *reader-classifier* threads (T_{ri} and T_{ci}). Each pair shares a unique communication channel C_i.

Each reader thread T_{ri} reads one of the equal-sized slices of S at a time. Once the T_{ri} has read a slice from S, it puts that slice in the communication channel C_i and notifies the corresponding T_{ci}. When T_{ci} receives the notification, it reads data from C_i, freeing it for new data (from T_{ri}).

To classify the updates into the update files UF_m, each classifier thread T_{ci} has access to its M partitioned local buffer LB_i of size $M \times 4[mb]$.

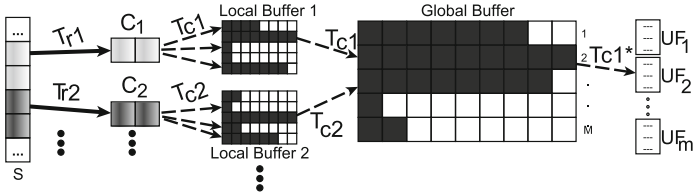

Fig. 1. Classify. *Reader* threads in continuous lines, *classifier* threads in dashed lines

Keeping the large size of updates in mind, we implement a second level of buffer called *Global buffer*, mostly to reduce synchronizations and IO operations. This buffer of size $M \times 8[mb]$ is common to all the classifier threads, consequently is protected by a *mutex*, preventing multiple classifiers to access the same partition concurrently. The size of the buffer is experimentally selected to achieve the best performance.

Each thread T_{ci} classifies the updates and stores them into their corresponding local buffer partitions: LB_{ij}. As soon as a partition j of the local buffer LB_i becomes full, its data is put into the corresponding partition of the global buffer by T_{ci}. Once the global buffer partition is full, the data of that particular partition j is written into the update profile file UF_j. The thread $T_{ci}*$, who made j's partition in the global buffer full, writes the update profile file UF_j. As only one thread has access to the global buffer of some partition j, when this is full, there is no need of synchronization to write UF_j file.

In *UpKNN* a key factor to achieve high performance is the overlap of computations and IO operations. While a reader thread obtains data from S (IO request), a classifier thread classifies updates in partitions, preliminarily stored on in-memory buffers and later written into the corresponding update files.

3.2 Merge

The phase merges the updates from these UF_m files with the already existing M profile files PF_m in disk. We use a set of M threads T_{me} to process the updates from these files in parallel. We have enough threads T_{me} so that each file is read and merged in parallel, leveraging IO parallelism observed on modern SSDs.

To merge, each thread T_{me} loads the updates from the corresponding update profile file UF_i into memory. These updates are inserted sequentially into a *heap* to sort them by entities' id. The purpose of sorting the updates by entities' id is to have all the occurrences of a particular user continuously. Now that the updates are sorted by id, T_{me} proceeds to read sequentially from disk the profile file PF_i (obtained from the underlying KNN approach) and to merge them with the updates from the heap. The process of merging old profiles with new items is performed in-memory. Finally, T_{me} writes the updated profiles back to PF_i. Using the same thread for reading and writing, avoids synchronization operations and related costs, and hence, achieves full parallelism in IO operations.

4 Evaluation

UpKNN is implement in C++, clang-omp++ 3.5.0, −O2 optimization. *Openmp* and *Pthreads* enable multithreading computation. We ran our experiments on a MacBook Pro laptop, Intel Core i7 4 cores, 16 GB RAM and a 500 GB SSD.

Although *UpKNN* is independent of the underlying KNN algorithm, we show a particular instance of its implementation on *Pons* [3]. *UpKNN* is evaluated on *Movielens*, which provides the movie-rating data from the Movielens website. Users' profiles are composed of their affinities for some movies. Additionally, we

Table 1. Datasets

Dataset	Users	Items	$\#Up$ (80% items)	M
Movielens (MOV)	138,493	20,000,263	16,000,210	2
Mediego (MED)	4,130,101	7,954,018	6,363,214	2

use *Mediego (MED)*'s dataset, which consists of users and the webpages they visit from various websites. In both cases, each user activity has a timestamp, which is used to divide the profiles into initial profiles (20% of the items) and the update set S (80% of the items) (Table 1).

Baseline. To the best of our knowledge, there are no out-of-core algorithms updating profiles while computing the KNN. To overcome this, we choose a natural baseline, which also uses a multithreading approach, where several threads read the updates from the update set and add them to the respective profiles.

4.1 *UpKNN*'s Performance

Runtime. Table 2 shows *UpKNN*'s and baseline's wall-time and speedup for computing the corresponding $\#Up$ (20/80% division, 10 M and 100 M randomly generated updates).

UpKNN considerably outperforms the baseline on both the datasets. *UpKNN* achieves a speedup of 49.5X on *Movielens*, taking only 3.687 s for about 16 million updates. We obtain a speedup of 47X on Mediego's dataset.

We notice from Table 3, that *UpKNN* processes more than 4 million [updates/second], for both the datasets, being consistent with the motivation of our work. *UpKNN* not only performs the computation on a single commodity PC, but also does it in real-time, making it a novel approach in itself.

In Fig. 2, we verify *UpKNN*'s scalability in terms of updates processed. Even after increasing the number of updates from 10 M to 100 M, the execution time increases only by a factor of 10.

Number of Threads. Figure 3 presents the wall-time of executing 100 M updates, varying the numbers of threads. We observe near-linear decrease in

Table 2. *UpKNN*'s performance

Data	$\#Up$	$UpKNN$[s]	Base.[s]	Speedup
MOV	10 M	3.635	105.747	29.08X
MOV	20/80	3.687	184.513	**49.5X**
MOV	100 M	39.662	1055.804	26.61X
MED	20/80	1.543	72.576	**47X**
MED	10 M	17.665	198.658	11.24X
MED	100 M	47.329	1931.154	40.80X

Table 3. $\#Up/second$

Dataset and $\#Up$	Time[s]	$\#Up/sec$
MOV 20/80 (16 M)	3.678	4.33 M
MED 20/80 (6.3 M)	1.543	4.12 M
MOV 100 M	39.662	2.52 M
MED 100 M	46.329	2.11 M

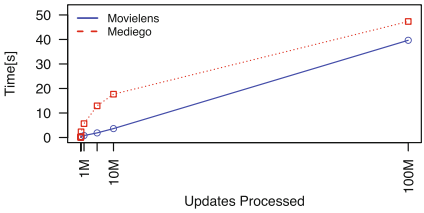

Fig. 2. Updates Scalability

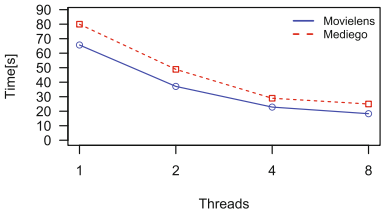

Fig. 3. Threads Scalability

the runtime when the number of threads increases. This small difference is due to the increase in threads synchronization, and to some small pieces of sequential code.

Disk Operations. As evident from Table 4, *UpKNN* reduces considerably the number of disk operations performed throughout the process. In the case of ordering the updates in time, we obtain better results than the case where the updates are randomly put in the set. In the former case, *UpKNN* takes only 0.0006% of the seeks performed by the baseline. The bytes written in our approach are reduced to only 1.98% of those of the baseline, and the bytes read are reduced to 3.88% of those of the baseline. These differences are explained by *UpKNN*'s capability to apply all the updates for a profile at once. Conversely, the baseline reads/writes the whole profile each time there is an update for it.

UpKNN's performance relies on its capacity to reduce disk operations throughout each phase of the computation. For instance, the updates (read from disk) are accessed only once on the classification. In addition, the *heap* reduces the need of multiple profile readings/writings.

Table 4. Disk Operations

MOV 20/80				MED 20/80		
	UpKNN	Baseline	%	*UpKNN*	Baseline	%
Disk seeks	29	48 M	0.0006	27	19 M	0.0001
Written [bytes]	128 M	6400 M	1.98	50 M	2570 M	1.98
Write op. [#]	12	16 M	0.0001	10	6 M	0.0001
Read [bytes]	256 M	6592 M	3.88	101 M	2621 M	3.88
Read op. [#]	127	32 M	0.0004	55	12 M	0.0004
MOV 100 M				MED 100 M		
Disk seeks	277 K	300 M	0.092	8 M	300 M	2.753
Written [bytes]	856 M	40 KM	2.118	2468 M	40 KM	6.110
Write op. [#]	138 K	100 M	0.138	4.1 M	100 M	4.130
Read [bytes]	1656 M	41 KM	4.019	3268 M	41 KM	7.933
Read op. [#]	139 K	200 M	0.069	4.1 M	200 M	2.065

5 Conclusions

We presented *UpKNN*, a multithreading *out-of-core* approach to handle updates on users-profiles, while the K-Nearest Neighbors computation is performed. The performance of our novel approach relies on a carefully designed set of in-memory buffers. *UpKNN* uses these buffers to overlap IO requests and CPU computation throughout the processing. This optimization goes together with a significant reduction in IO operations, the main bottleneck on out-of-core algorithms.

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

References

1. Boutet, A., Frey, D., Guerraoui, R., Kermarrec, A.M., Patra, R.: Hyrec: leveraging browsers for scalable recommenders. In: Middleware (2014)
2. Boutet, A., Kermarrec, A.M., Mittal, N., Taïani, F.: Being prepared in a sparse world: the case of knn graph construction. In: ICDE (2016)
3. Chiluka, N., Kermarrec, A.-M., Olivares, J.: The out-of-core KNN awakens: the light side of computation force on large datasets. In: Abdulla, P.A., Delporte-Gallet, C. (eds.) NETYS 2016. LNCS, vol. 9944, pp. 295–310. Springer, Cham (2016). doi:10.1007/978-3-319-46140-3_24
4. Dong, W., Moses, C., Li, K.: Efficient k-nearest neighbor graph construction for generic similarity measures. In: WWW (2011)
5. Lathia, N., Hailes, S., Capra, L., Amatriain, X.: Temporal diversity in recommender systems. In: SIGIR (2010)
6. Rana, C., Jain, S.: A study of dynamic features of recommender systems. Artif. Intell. Rev. **43**, 141–153 (2012)
7. Yang, C., Yu, X., Liu, Y.: Continuous knn join processing for real-time recommendation. In: ICDM (2014)
8. Yu, C., Zhang, R., Huang, Y., Xiong, H.: High-dimensional knn joins with incremental updates. Geoinformatica **14**(1), 55–82 (2010)