Scaling KNN Computation over Large Graphs on a PC

Nitin Chiluka, Anne-Marie Kermarrec, Javier Olivares

Scaling KNN Computation over Large Graphs on a PC

Nitin Chiluka
INRIA Rennes, France
nitin.chiluka@inria.fr

Anne-Marie Kermarrec
INRIA Rennes, France
anne-marie.kermarrec@inria.fr

Javier Olivares
INRIA Rennes, France
javier.olivares@inria.fr

ABSTRACT

This paper proposes 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. 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.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and Networks*; I.5.3 [**Computing Methodologies**]: Clustering—*Algorithms, Similarity measures*

General Terms

Algorithms, Design, Performance

Keywords

K-nearest neighbors, Dynamic Graphs

1. INTRODUCTION

Frameworks such as GraphChi [2] and X-Stream [3] 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 and triangle counting. 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) – which is widely used in recommender systems [1].

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 $t+1$.

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 $P(t)$ at iteration t, which can also change over time to $P(t+1)$.

Our goal in this work is to design a system that can compute KNN of each user efficiently in a memory constrained machine, considering that user profiles change over time.

Middleware'14: Demos and Poster, Dec 08 - December 12 2014, Bordeaux, France

http://dx.doi.org/10.1145/2678508.2678513 ...$15.00.

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.

Our approach to computing KNN at each iteration t proceeds in five phases, as shown in Figure 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 $P(t+1)$ for the next iteration.

2.1 KNN Iteration

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

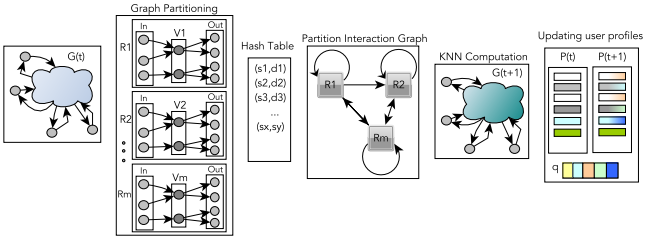

Figure 1: 5 phases: input $G(t)$, 1) KNN graph partitioning, 2)Hash Table, 3)PI graph, 4)KNN computation, 5)Updating profiles

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

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 from 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. 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.

The third phase is the creation and traversal of the **partition interaction (PI) graph** which helps in deciding the order in which all the tuples' similarity scores are computed. In the PI 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 PI graph are parsed, it means that the similarity scores of all the corresponding tuples in H have been computed.

We describe a few heuristics to decide the order in which the PI graph is parsed. The *sequential* heuristic loads the partition starting from number 1, processes all its edges in the PI 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 1: # Load/unload operations using PI graph.

Datasets	Nodes	Edges	Seq.	High-Low	Low-High
Wiki-Vote	7115	100762	211856	204706	202290
Gen. Rel.	5241	14484	34506	32220	31256
High Ener.	12006	118489	252754	242132	240872
AstroPhy.	18771	198050	420442	400050	401770
E-mail	36692	183831	399604	382928	379312
Gnutella	26518	65369	157040	144072	132710

Table 1 presents a preliminary evaluation of these heuristics on various datasets. If the PI 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.

The fourth phase performs **KNN computation** using the PI graph and the profiles $P(t)$ to generate $G(t+1)$ which is the new KNN graph for the next iteration. First, the PI graph is parsed in the order based on one of the above heuristics 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, one can generate the K-most similar neighbors for each user, resulting in $G(t+1)$.

Finally, the fifth phase is responsible for **updating user profiles** from $P(t)$ to $P(t+1)$. Throughout the iteration t, any changes in the profiles of the users are stored in a queue q but not incorporated into $P(t)$. In this phase, the queue is read to update the profiles to $P(t+1)$. 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.

Future Work. We plan to evaluate our approach using different graph sizes, amounts of memory, HDD and SSD, and multiple threads by measuring execution times as well as throughput from the disk IO operations. Furthermore, we aim to develop more heuristics for the PI graph traversal which consider the amount of time consumed for both partition load/unload operations and the similarity computation for tuples given two partitions.

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

3. REFERENCES

[1] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[2] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[3] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, 2013.