A Programming Interface and Platform Support for Developing Recommendation Algorithms on Large-Scale Social Networks

Alejandro Corbellini, Daniel Godoy, Cristian Mateos,
Alejandro Zunino, and Silvia Schiaffino

ISISTAN Research Institute - Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Univ. Nacional del Centro de la Provincia de Bs. As. (UNICEN), Campus Universitario, Paraje Arroyo Seco (BBO7001B), Tandil, Buenos Aires, Argentina

Abstract. Friend recommendation algorithms in large-scale social networks such as Facebook or Twitter usually require the exploration of huge user graphs. In current solutions for parallelizing graph algorithms, the burden of dealing with distributed concerns falls on algorithm developers. In this paper, a simple yet powerful programming interface (API) to implement distributed graph traversal algorithms is presented. A case study on implementing a followee recommendation algorithm for Twitter using the API is described. This case study not only illustrates the simplicity offered by the API for developing algorithms, but also how different aspects of the distributed solutions can be treated and experimented without altering the algorithm code. Experiments evaluating the performance of different job scheduling strategies illustrate the flexibility or our approach.

1 Introduction

Friend recommendation algorithms try to infer missing edges among members of a social network that are likely to be established in the near future. For large-scale social networks such as Twitter or Facebook, these algorithms face challenges related to the amount of data to be processed. Graph-specific databases or frameworks for parallel processing of graph algorithms have arisen to address these issues. These supports do not provide by themselves means to improve different aspects of the cluster usage according to the algorithm requirements and, thus, the responsibility falls on the programmer who must modify its algorithm to handle job distribution.

In this paper, we focus on providing a simple yet powerful API to implement graph traversal algorithms. By using the API and its related abstractions, new graph-based recommendation algorithms can be quickly prototyped without thinking about distribution and parallel concerns. Also, the platform provides customizable scheduling strategies or "rules" that determine the job-node mapping that the algorithm will use, allowing a fine-grained control over the cluster usage without altering the original algorithm.

In order to illustrate and evaluate the proposed approach we present a case study in which an algorithm for followee recommendation in Twitter, described in [1], was adapted to the proposed API and executed using different scheduling strategies, namely, a Round Robin strategy, a Location Aware strategy and two Memory-based strategies. For each strategy we compared the adapted algorithm performance in terms of recommendation time, memory usage and physical network usage.

N. Baloian et al. (eds.): CRIWG 2014, LNCS 8658, pp. 75–82, 2014.

The rest of this paper is organized as follows. Section 2 overviews of the existing work on distributed graph processing frameworks and APIs. Section 3 describes our proposed framework and the implementation of the back-end storage support. Section 4 presents the API of our adjacency Graph that allows the programmer to query the structure of the graph and set the scheduling strategy to be used. Section 5 presents a case study of developing a followee recommendation algorithm and its testing with a Twitter dataset. Finally, conclusions drawn from this work are summarized in Section 6.

2 Related Work

Similarly to the case study presented, there are many link-prediction algorithms that are based on the structure of the graph rather than its content. For example, SALSA [10] is an algorithm that uses the "hub" and "authority" notion of HITS [6], and the "random surfer" model from PageRank [13]. The Who To Follow [4] user recommendation algorithm developed by Twitter, uses PageRank to build the initial group of results and then uses that group as an input to a SALSA-based recommendation algorithm. A more specialized recommendation algorithm that recommends a person's contacts in the case of emergency scenarios based on her social network can be found in [15].

Several frameworks that support graph processing algorithms in distributed environments can be found in the literature. For example, HipG [7] allows modeling hierarchical parallel algorithms, which includes divide-and-conquer graph algorithms. Pregel [11] is a closed-source framework used at Google that provides a computational model for large-scale graph processing. Sedge [18] implements the Pregel model, but focuses on graph partitioning. Trinity [14] is a graph processing framework created by Microsoft, that provides some interesting features such as online query processing and native graph representation. Graph processing frameworks usually load data from a persistent store and then construct a distributed in-memory representation of the graph. As stated in [4], on large-scale graphs this can be a problem if the growth of the amount of physical memory to store the graph cannot keep pace with the growth of the graph.

In this work we created a distributed graph store that allow users to perform distributed computation over its data. In its current implementation the store provides a graph traversal API over an existing *back-end* store. This allowed us to focus on the development of the API and its abstractions rather than the inners of the graph storage.

There are many implementations of graph APIs over existing storage supports. For example, FlockDB [17] is a graph database that uses MySQL as an storage back-end and provides a graph API on top of SQL. Similarly, Titan [2] is a graph database that provides a graph traversal API while supporting several storage backends. On the other hand, there are graph databases that use structures specially suited for linked data. For example, Neo4j [12] is a popular graph database that uses data structures adjusted to its graph representation.

3 Distributed Execution and Storage Support

We based our proposed approach on a set of tools created to efficiently distribute code and data across a number of nodes. From the set of tools, the Distributed Execution

Framework and the Distributed Key-Value Store are the most relevant to the graph API proposed and are described below.

3.1 Distributed Execution Framework

A Distributed Job Execution Framework (DEF) was developed to simplify the creation of programs that send data and code (jobs) between computational nodes. The framework is composed of several software layers, implemented in Java, that provide abstractions at different levels to execute distributed recommendation algorithms. The first layer is the Network Module that handles networking details. The second layer is an RPC module that handles remote method calling and marshalling (conversion of objects to bytes and viceversa) of parameters and return values. The topmost layer is the Job Execution module that uses the RPC Module to communicate to other Job Execution modules. This layer sends jobs to execute on a specific node, it gathers results and handles errors.

3.2 Distributed Key-Value Store

Our graph implementation (Section 4) is built upon a distributed key-value store (KVS) that saves the target graph in a distributed manner. A KVS has, at least, two operations: a *put* operation that stores a given value under a given unique key, and a *get* operation that retrieves the value associated to a given key. Thus, the structural information of a graph (links and vertices) can be saved in a key-value store by assigning every vertex a unique key, e.g. a Twitter user identifier. Then, the list of vertices pointing to and from a vertex is stored as a value under the vertex key. This type of structure is called an Adjacency Graph.

Although there are many databases that provide a key-value API [3,16], a DEF-based implementation provide us fine-grained control over data distribution, memory consumption and peer-vertex allocation.

4 Adjacency Graph

As mentioned before, we created an Adjacency Graph for representing the graph structure and persisted it in a distributed KVS. This graph implementation uses the DEF model to perform queries and distribute data handling among a number of nodes. Such distribution can be customized using Scheduling Strategies that map a group of vertices being queried to an specific node. Next, we will describe the API of the Adjacency Graph and the Scheduling Strategies that will be used for the experiments.

4.1 Adjacency Graph API

The starting point to obtain information about the graph is the AdjacencyGraph object and its getVertex(id:int) method. This method creates a VertexQuery, i.e., a query that represents a vertex or group of vertices. This query corresponds to a ListQuery type, which means it is a query that returns a list of vertices. A ListQuery contains a handful of methods to query connected vertices:

- *incoming/outgoing:ListQuery*
  Returns a ListQuery representing the list of vertices that have *outgoing* (or *incoming*) connections to the current list of vertices.
- *countIncoming/countOutgoing:CountQuery*
  Counts the amount of unique vertices with incoming (or outgoing) connections to the current list of vertices. The CountQuery object represents a dictionary of vertices and number of occurrences of each vertex.
- *union/intersect(q:ListQuery):ListQuery*
  Executes both *q* and the current query and intersects (or unites, if union is used) their results, creating a new ListQuery.
- *remove(q:ListQuery):ListQuery*
  Removes the list of vertices resulting from executing *q* from the current list.

Regarding the CountQuery object, a user can call the *top(n:integer):TopQuery* method to obtain the top *n* vertices with most appearances. Both ListQuery and CountQuery create new queries for each of the mentioned methods to programmatically concatenate an arbitrary number of queries.

4.2 Scheduling Strategies

When a user requests the incoming or outgoing vertices of a given group of vertices, a scheduling strategy is used to divide the request in several jobs and each job is assigned to a specific node. Indeed, many job allocation strategies can be used for this purpose. Then, we implemented and compared four scheduling strategies:

- Round Robin: This strategy simply divides the given list of vertices by the amount of nodes and assigns a sublist to each of the nodes.
- Available Memory: This strategy checks the memory currently available on each node and divides the given list of vertices accordingly.
- Total Memory: The Total Memory strategy uses the maximum amount of memory that a peer can use to divide the list of requests.
- Location Aware: This strategy takes advantage of the *locality* of the vertices, dividing the input into different lists of vertices grouped by their storage location.

The selection of strategies depends on the type of improvement that the user wants. For example, a Location Aware policy is the less network-consuming strategy. This is because the request for each incoming or outgoing list of a vertex is made on the node that stores the vertex data. However, if the vertices are not well balanced among nodes, a node may receive many requests to be locally processed, degrading its performance. Moreover, the Location Aware strategy must process the given vertices list and calculate the location of each vertex to make the division.

5 Case Study

The algorithm used as case of study, proposed in [1], is based on the characterization of Twitter users made in several studies [5,8], by which users are mainly divided in two categories: information sources and information seekers. Information sources tend

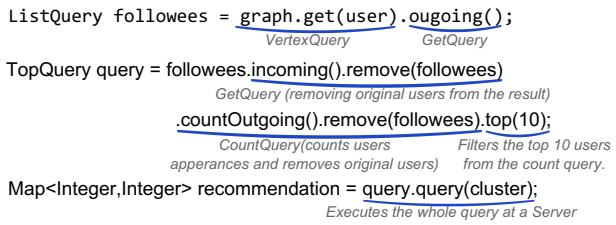


Fig. 1. The studied recommendation algorithm expressed using the DEF Query API

to have a large amount of followers as they are actually posting useful information or news. In turn, information seekers follow several users to obtain the information they are looking for, but rarely post a tweet themselves. The aim of the algorithm is finding candidate information sources to recommend to a target information seeker. To do so, it makes the asumption that those followees of the target's followers are similar to the target, and then, the users that they follow may be of interest to the target.

Next, we will describe the adaptation of the algorithm to our API and then the experiments' layout and results.

5.1 Followee Recommendation Algorithm Adaptation

The original recommendation algorithm was adapted to execute on top of our Adjacency Graph. Using the query API presented above, we expressed the algorithm as shown in Figure 1, which resulted in the following steps:

1. Get the followees (*outgoing*) of a user.
2. Get the followers (*incoming*) of the followees and remove the original followees from the list.
3. Count the followees (*countOutgoing*) of the followers and remove the original followees from the map.
4. Obtain the top n elements with most appearances ($n = 10$ in this example).

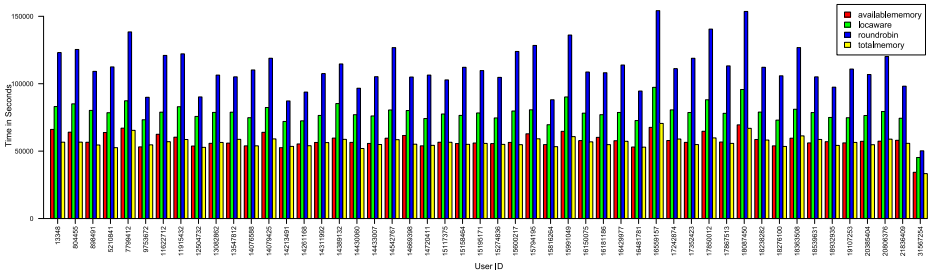
Essentially, the algorithm collects lists of users at different steps, and finally joins the final list to elaborate a ranking of candidate information sources.

5.2 Experiments

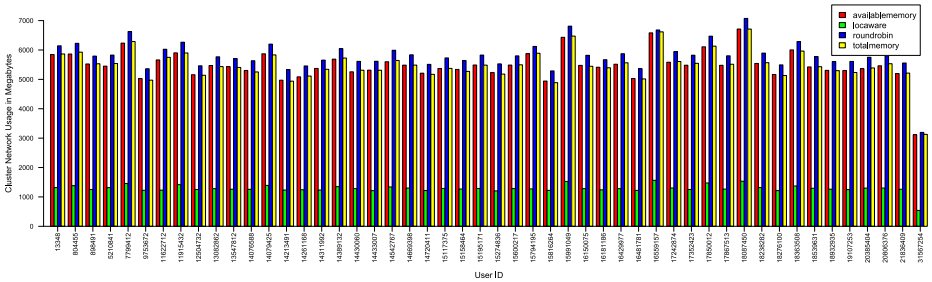
The experiments were carried out using a Twitter dataset[1] consisting of approximately 1,400 million relationships between 40 million users [9]. For selecting a representative test user list, we first filtered the list of users using the information source ratio [1], denoting to what extent the user can be considered as an information source. This ratio is defined as follows:

$$IS = \frac{followers(u)}{followers(u) + followees(u)} \tag{1}$$

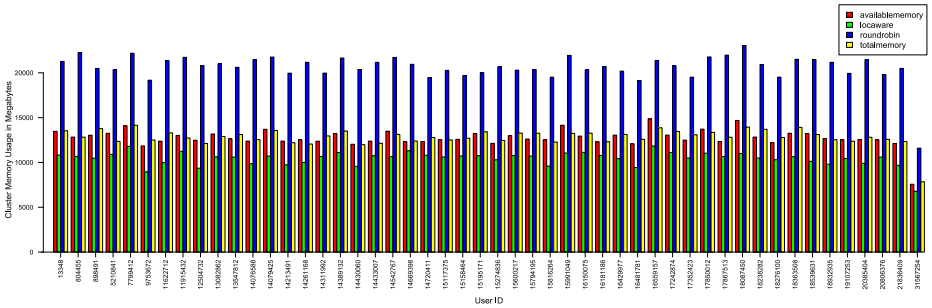
[1] http://an.kaist.ac.kr/traces/WWW2010.html



(a) Recommendation Time.



(b) Network consumption.



(c) Memory consumption.

Fig. 2. Experiment results

By this definition, an information seeker is a user that has an IS lesser than 0.5, i.e. it has more followees than followers. Moreover, we wanted to stress the platform by generating big intermediate results. To do so, we selected from the list of information seekers the top-50 users that had the longest followees lists.

Regarding the cluster characteristics, we used an heterogeneous cluster of 8 nodes. 5 of them had a 6-core AMD Phenom II CPU with 8GB of RAM, and 3 of them had a 6-core AMD FX 6100 CPU with 16 GB of RAM. Other characteristics, like network speed and hard disk, were similar for all of them.

The experiments were carried out as follows. For each user, we ran the algorithm 10 times for each strategy. For every run we measured the bytes sent over the network, the maximum memory consumed (the biggest memory spike) and the total recommendation time. Then we calculated the average for the whole execution.

The results obtained from the experiments are depicted in Figure 2. In Figure 2a we show the average time for recommendation. The memory-based strategies yielded the best recommendation times. As the algorithm consumes a lot of memory, a distribution of work based on memory results in better balance. In Figure 2b it can be seen that the amount of network consumed by the Location Aware strategy is, as we initially expected, far less than the consumption of other strategies. Finally, 2c shows the average of the maximum memory consumed on the cluster. The memory usage pattern of the Round Robin strategy yielded bigger memory spikes on every node of the cluster whereas the Location Aware strategy used the least RAM memory, which was expected as less memory was needed to make remote database requests.

6 Conclusions

In this paper we presented a graph API for developing distributed link prediction algorithms without concerning about the inner job-node distribution mechanisms. For testing this approach we used an existing recommendation algorithm for the Twitter social network as case study, with data of a complete Twitter graph of 2009. The objective of the algorithm is to recommend users that share the same followers than the followees of the target user.

The proposed API hides most of the distributed processes involved in the adjacency Graph implementation. Moreover, it also offers customizable Scheduling Strategies, i.e. the way jobs are created and assigned to nodes, to be used when querying the graph API. The election of the strategy may have an impact on the execution of the algorithm and the way it uses the cluster resources. To show the effect of strategy selection, we experimented with four types of scheduling strategies: Round Robin, Location Aware, Available Memory and Total Memory.

In our experiments, the Location Aware strategy showed approximately 84% less network bandwidth consumption than the other strategies and the least overall memory usage. Despite the time for recommendation was acceptable in comparison to the Round Robin strategy (about 50% less), it was 15% slower than the Memory-based approaches, which yielded the fastest recommendations. As expected, the Round Robin strategy yielded the worst results.

Future work includes a) the improvement of the underlying graph storage, b) the implementation of new strategies and c) the testing of this approach with other graph-based recommendation algorithms.

References

1. Armentano, M., Godoy, D., Amandi, A.: Topology-based recommendation of users in micro-blogging communities. Journal of Computer Science and Technology 27(3), 624–634 (2012)

2. Aurelius. Titan (2014), http://thinkaurelius.github.io/titan/ (accessed April 14, 2014)
3. Cattell, R.: Scalable SQL and NoSQL data stores. ACM SIGMOD Record 39(4), 12–27 (2011)
4. Gupta, P., Goel, A., Lin, J., Sharma, A., Wang, D., Zadeh, R.: WTF: The who to follow service at Twitter. In: Proceedings of the 22th International World Wide Web Conference (WWW 2013), Rio de Janeiro, Brazil (2013)
5. Java, A., Song, X., Finin, T., Tseng, B.: Why we Twitter: Understanding microblogging usage and communities. In: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis, San Jose, CA, USA, pp. 56–65 (2007)
6. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. Journal of the ACM 46(5), 604–632 (1999)
7. Krepska, E., Kielmann, T., Fokkink, W., Bal, H.: HipG: Parallel processing of large-scale graphs. ACM SIGOPS Operating Systems Review 45(2), 3–13 (2011)
8. Krishnamurthy, B., Gill, P., Arlitt, M.: A few chirps about Twitter. In: Proceedings of the 1st Workshop on Online Social Networks (WOSP 2008), Seattle, USA, pp. 19–24 (2008)
9. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), Raleigh, NC, USA, pp. 591–600 (2010)
10. Lempel, R., Moran, S.: SALSA: the stochastic approach for link-structure analysis. ACM Transactions on Information Systems 19(2), 131–160 (2001)
11. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 International Conference on Management of Data (SIGMOD 2010), Indianapolis, IN, USA, pp. 135–146 (2010)
12. Inc. Neo Technology. Neo4J (2013), http://www.neo4j.org/ (accessed August 5, 2013)
13. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web, pp. 1–17 (1999)
14. Shao, B., Wang, H., Li, Y.: The Trinity Graph Engine. Technical Report MSR-TR-2012-30, Microsoft Research (March 2012)
15. da Silva, S.T.F., Oliveira, J., Borges, M.R.S.: Contextual analysis of the victims' social network for people recommendation on the emergency scenario. In: Herskovic, V., Hoppe, H.U., Jansen, M., Ziegler, J. (eds.) CRIWG 2012. LNCS, vol. 7493, pp. 200–207. Springer, Heidelberg (2012)
16. Strauch, C., Sites, U.L.S., Kriha, W.: NoSQL databases. Lecture Notes, Stuttgart Media University (2011)
17. Twitter Inc. FlockDB (2013), https://github.com/twitter/flockdb (accessed August 5, 2013)
18. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: Proceedings of the 2012 International Conference on Management of Data (SIGMOD 2012), Scottsdale, AZ, USA, pp. 517–528 (2012)