Approximate graph processing
Extended abstract

Renato David Silva Rosa

Instituto Superior Técnico
Lisboa, Portugal

Abstract. Graphs are important data structures which are present everywhere. In this work, we present the problem of huge graphs that are constantly changing, at real time, and present a proposal of a system based on approximate processing.
After some insights about real world graphs, and the challenges they present, a possible approach is described. This is based on tracking of the updates to the graph, which come as a stream, providing useful statistics to a user defined function, which can decide to compute the exact result, compute an approximation or to simply repeat the last answer.
The experiments show how computing an approximation can give good results in terms of the quality of the answer obtained, which is evaluated against the exact one, and in terms of reduction of time and complexity needed to answer the query.

1 Introduction

Many objects and realities of our daily lives, whether natural or created by man, take the form of networks, being constituted by individual elements that relate to each other. These networks can be formally represented as *graphs*. Graphs and their study are present in many fields of knowledge, not only in mathematics, particularly in graph theory, which studies them thoroughly, but also in social and natural sciences [6, 4].

There are abundant applications of graphs and various functions and measures that can be computed based on them: paths between vertices, distances, components, cuts, centrality measures, community identification (clustering), among many others.

1.1 Motivation

Graph processing involves large structures which, more recently, are also constantly changing. Distributed systems allow, in this context, to perform processing on a large scale, with timely results. However, in any of these systems, the issues related to the optimization of resources and response time are a constant concern, and the context of graphs is no exception.

We consider as the typical case a system which has a representation of a graph and continuously receives updates that are applied to it. There is also a function of interest that should be computed from the graph, and the changes to the graph imply its recomputation, whether made from scratch or incrementally over previous results.

Large scale processing workflows, as well as in the individual components which constitute them, use a large amount of resources, like CPU, memory, I/O, and time, necessary to compute an updated response. It happens, though, that sometimes there is little difference between the result of the new computation and the results obtained previously, despite all resources spent. However, in certain areas, it is tolerable to admit an inexact answer, but sufficiently close to the exact one, if it means significant savings in terms of computing resources and time. In that case, the system could be limited to return the previous answer or an approximated one, deferring a full computation for the moment when this is expected to produce significant results. For that, we need to have a compromise between response correction and the use of resources used to obtain that response.

Hence, resource optimization and efficiency in large-scale and continuously changing graph processing motivates this work. Some examples of use cases where the ability to process this kind of changing graphs could be used include detection of trending topics in social networks, recommendation systems, and fraud detection, among others.

1.2 Challenges

Graph processing presents particular computational challenges in terms of representation, storage and processing. Its features and its importance prompted the emergence of a large number of techniques, algorithms and systems specifically designed to promote efficiency and responsiveness.

Some real life graphs are really huge for human scale. Maybe the best example is the World Wide Web (WWW). Even human/social networks can be quite large. In these large networks, there is a major challenging feature: their heterogeneity. The vertices of some graph can be very different in terms of the degree distribution. In many actual graphs, we can find a large number of vertices with small degree but also a small number of vertices, known as hubs, with a very high number of links [4][5]. This heterogeneity presents considerable problems for a graph processing distributed system: for instance, following a simple strategy of vertex distribution among the system nodes, workload distribution can easily suffer from large imbalances, by the presence of hubs, which require, typically, much more processing.

Another important challenge relates to the fact that most networks evolve over time, either in their structure or in the properties that are associated with its elements. An object that changes continuously requires specific care. The problem of real graph volatility is one of the motivations for the contribution that we hope to provide with this work.

1.3 Goals

The issue of mutability of networks introduces the use of streams in the context of graph processing. A good way to represent the evolution of a graph is through a stream of updates that contain changes to the graph. Changes may either keep intact the graph structure, modifying only data associated with vertices and edges, but can also represent topological changes to the graph, by insertion or removal of vertices and edges. These two types of updates have different implications to distributed processing, representation, load balancing and optimization.

Instead of building a static graph and then analyze it, we can receive a stream of incremental updates. Well-known social networks such as Facebook, Twitter, blogs, have these characteristics, and are good examples of graph stream processing in real time.

The aim of our work is therefore to provide means to optimize the use of computing resources and also the response time in distributed systems for graphs.

However, each actual problem has different characteristics, and the same applies to datasets. When we talk about changing graphs, it is clear that it is not possible to have some kind of universal solution. So, our goal is to provide the means, under the form of some library or API, to reason about the changes to the data, and the past results, and make decisions. This API can be plugged, for example, with some kind of intelligent system to implement a full processing system for graphs, with approximate results, and suitable for each case.

In the following sections, after presenting the work related with this area, and describe the design, architecture and implementation of GraphApprox library, a solution that aims to achieve reduced time and resources needed to compute a response, obtaining an approximate, but good enough solution. At last, we show how the concept and the solution was evaluated, and conclude this extended abstract.

2 Related work

There is considerable work in the fields of big data, streams, graph processing, and approximate processing. Ideas from MapReduce and its implementations [7, 21] have been adapted in ways more appropriate for iterative processing over the same data, with in-memory storage, and optimization, for examples, in systems like Spark [22] or Flink [2].

2.1 Batch and stream processing

The need to store and process large amounts of data has been present for some years. That need gave arise to a lot of techniques and algorithms to optimize that processing and to assure timely and reliable results.

With this in mind, we can consider that a first, general, approach to graph processing would be the use of general-purpose systems and algorithms to graphs, with the necessary adaptation.

The need to process large amounts of data quickly, reliably, and efficiently and has led to new strategies to address the problems. Parallelization emerged as the great paradigm to address that. MapReduce pioneered the dissemination of this paradigm, and still is a reference point [7]. Systems like Hadoop implement this model and have become highly popular for the processing of big amounts of data [21].

A problem with this kind of models/systems is that they are not so well suited for problems that require iterative computation over the same data, as well for interactive analysis. To answer this, a new generation of solutions/algorithms appeared. They are based in parallelizable data structures which are kept in memory.

Apache Spark implements this approach using the so called Resilient Distributed Datasets (RDDs) [22], which aim to give performance guarantees at the same time they provide good file recovery.

Apache Flink is at the first sight very similar to Spark, but has a different basis: it is built on stream processing, and batch processing is considered a specific case of that. This approach allows to avoid some latencies and to unify batch and stream processing under the same framework [2].

There are also meta-systems, like Lambda Architecture, which combine different of these approaches to choose, for each actual problem or data set, the subsystem/algorithm more appropriate [17].

The problem with this family of solutions is that they are too much general. They have the goal to be suited for any kind of data, whatever it is, and this way is impossible to have intrinsic specialization in data like graphs. Graphs are very specific structures that offer some complexity but also some opportunities for optimization that a general solution like MapReduce cannot offer for itself [9].

2.2 Graph processing

This takes us to a group of solutions specifically dedicated to graph processing. Graphs are easily representable with tabular or key-value formats, heavily used in big data systems, but that simplicity misses a lot of opportunities for optimization that come from the very nature of graphs [9].

Distributed and parallelizable graph processing demands different ways to express algorithms, even the classic ones. A very common approach consists in thinking about the algorithm in terms of independent small computations made in each vertex, and information shared among them. One of the first well-known frameworks working this way is Pregel [15]. Many other systems and libraries provide APIs to express algorithms in this manner.

This concept had some optimization, variants and different implementations. For example, information sharing can be made through messages or shared memory (GraphLab [14] being one example of the latter approach). Other proposals made a separation of the different phases of the vertex program: messages/shared information gathering; computation over the data, applying the results; sharing of information/messages for the next step. This model is known as Gather-Apply-Scatter (GAS) and has been introduced by Powergraph [10]. The separation among phases allows the system to further optimize the data flow of the programs.

A related, but different, approach, is the one that expresses computation as small units not over vertices, but over edges (X-Stream/Chaos is a representative product of this [16]). The logic behind this approach is that the flow of information in a graph is more closely related to the edges than to the vertices. This is related to the different strategies to distribute the graph among computation nodes: some distribute the vertices, sharing edges, but the majority distribute the edges, partitioning vertices among machines. This solution is more suited for scale-free graphs, where there is a great heterogeneity in vertex degrees.

At last, let's take a sight to the graph libraries provided by generic frameworks like the ones we saw in the last section. These are usually implemented as dedicated data structures and API's for graph definition and manipulation, derived from the structures and operators made for generic data. This approach allows to include some built-in optimizations for graph problems and a friendly way to express algorithms/computations over graphs.

Spark, for example, offers GraphX, which has the goal of unifying, under the same framework, the analysis of graphs and non-structured and tabular data [9]. Resilience and fault tolerance is achieved using the underlying techniques of the framework. Flink has its graph library, too, named Gelly, similar to GraphX in the concept of graph representation (data sets of vertices and edges) and operators. Both provide the user with different ways to express algorithms on graphs.

2.3 Approximate processing

Approximate processing is one of the possible solutions to the problem of scalability. Frequently, in the presence of huge amounts of data, exact computation is too much expensive in terms of time or resources spent. When the problem to solve admits a not exact but sufficiently close answer, approximate processing can be helpful.

In the context of streams, and specifically graph streams, there are some known techniques to provide approximate results.

Load shedding consists in discarding some elements of the stream, when the load of new data is too much to be processed efficiently [3]. The questions here are when to discard data, at which point of the data flow (as soon as possible, of course, but not too soon), and which data to discard. Load shedding can occur at random or be based in semantic criteria, when the characteristics of data are known. It is important also be able to evaluate the impact of the shedding in the quality of the responses [18].

Sampling is another general technique of approximation. There is a lot of statistical reasoning possible to do on sampling, providing ways to obtain samples and estimate the error introduced. Sampling of graphs includes different techniques (for instance, sample vertices or edges, random walks) and is still an open field of investigation [11].

Some algorithms on graphs have also an approximate version. The approximation typically requires less resources and can offer statistical guarantees of approximation to the exact results.

Closely related to approximate processing is the concept of Quality of Data. This is a way to express the requirements, different for each problem and application, in terms of error tolerance and the level of data changing that trigger the computation, as well as requirements about latency and throughput. Usually, these requirements can be formulated as rules on three axes: time between computations; sequence, the maximum number of updates that can be applied to an object; and value, the maximum relative divergence between the updated state of an object and its previous state, or against a constant, since the last execution [8].

3 Design and implementation

We designed our solution to the issue of approximate processing of continuously changing graphs as a library, which we named GraphApprox, to be used in association with Apache Flink. In this solution, we offer an intermediate solution between exact computation and the repetition of the last answer: the approximate processing of the graph, which could likely offer better results while still reducing resource usage.

The motivating application for this approach is PageRank, a well known centrality measure for graphs, which has a rather intuitive implementation in terms of a vertex-program and shared messages.

3.1 Approximate processing API

We implemented our system as a library, writen in Java, on top of Flink. Everytime it was possible, actual details related with the characteristics of the data set, the stream of updates and the output of results were made abstract. When convenient, we used the Flink API itself to implement the dataflow and transformations on the data, through Flink *operators*, leveraging the availability of a large scale processing system. Flink provides a functional style of programming, where the user defines a chain of operators is applied to a data source, providing results to a data sink. This definition is converted in an optimized *job graph*, which is then submitted to the running instance of Flink.

Our module, which implements approximate processing of changing graphs, requires only a running instance of Flink, in the same machine or in a different one. Our program acts as an intermediary system between the graph update stream and the graph processing system/library itself, Gelly, in this case. Figure 1 shows how these parts fit together.

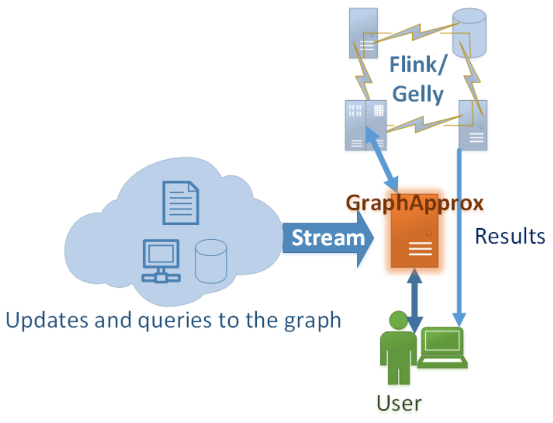

Fig. 1. General architecture of the solution

Since there is a lot of problems on graphs that can be answered this way, and considering that for each specific case the criteria would be different, our strategy was to provide some kind of API to the user. The user defines, through the API, where a running instance of Flink can be found, the initial graph, the stream of updates and a set of user-defined functions (UDFs) encapsulated in an object.

In order to provide approximate processing based on reasonable assumptions about the data, our program keeps track of all the updates to the graph received since the last computation. This tracking is made in memory, with regular data structures from Java Collections Framework and some simple data structures specifically created. For each edge arrived, data related to vertices (in- and out-degree, number of updates) is updated.

As soon as a query arrives, GraphApprox is notified. At first, one of the UDFs, *beforeUpdates()*, allows the user to decide if the updates must be applied to the graph or delayed.

Next, some data are provided to another UDF, *onQuery()*: the query itself; the updates received since the last computation; the current graph. There is also the possibility to obtain statistics about the updates and about the graph. Some of these statistics can be eagerly obtained (when they do not involve complex computation, like the number of edges added or removed, the ratio of vertices affected), but others are available only on request, when they imply some additional computation (for instance, average degree of the graph, clustering coefficient, etc.). The user who provides the function must evaluate the cost-benefit relation when implementing it.

This UDF ends returning one of three possible responses: 1. answer immediately the query with the last computed result; 2. perform an approximated computation, using the data from the updates; 3. perform a full, exact computation on the graph. Before that, there is the possibility of, possibly based on the information about the updates and the graph, changing the parameters passed to the algorithm that computes the approximated or exact result on the graph, as a side effect.

After the answer are available, another UDF, *onQueryResult()*, is called with the results of the computation and some statistics about it.

In the implementation, we had to deal with some issues, like Java 8 lambda expressions and type safety, or the need to explicitly cache some intermediate results, in order to avoid repeated work.

3.2 Approximate PageRank algorithm

Our approximation algorithm to page rank is based on graph summarization, a strategy already studied [12].

Vertices that receive more updates are more likely to have its rank changed; their direct neighbors will probably have their rank changed, too, but as we go away from the "hot" vertices, the rank is likely to remains the same. According to this idea, first, a set of "hot" vertices is chosen. There are two parameters that control this selection.

The first one is a threshold on the relative degree change of some vertex. For each vertex, between queries, when updates are processed, we register the number of updates received and the new in- and out-degree. With that information, we are able to calculate the magnitude of the changes to the degrees, relative to the previous degree. This way, and are able to select, for example, all the vertices whose in-degree changed more than 10%.

The second parameter is the neighborhood size. Starting with a set of vertices, updated above the mentioned threshold, let's call it the *kernel*, given a neighborhood size of n, we can expand the selection to include all the vertices that are at a distance at most n from some vertex in the kernel. This can be done with the operators of the underlying graph library.

The set of "hot" vertices selected, all the edges between them are selected too. Remaining vertices are coalesced in a big one, which is added to the graph, and represents all the vertices whose rank we consider not likely to change.

All the edges between vertices not in the "hot" set are discarded. The edges with target in the "hot" vertices and exterior source are maintained, now with the big vertex as its source. Edges in the opposite direction are discarded too, because, according to PageRank definition, they would only have influence on the rank of the big vertex, and that is not relevant.

Note, however, that, even removing out edges from the hot vertices, the original out-degree shoud be kept, because it is used in the calculation of the rank to be sent to each neighbor, at each iteration. That value is registered in the vertex and used during the algorithm, as if the out edges were still present.

In a similar manner, the vertices in the "hot" set are supposed to continue receiving rank from the exterior. That is the reason we kept the edges from the big vertex to the other vertices. We use each edge to keep the rank that each "hot" vertex would receive from the exterior. This way, the big vertex, at each step, sends to the other vertices the total rank they would receive from the exterior in the real graph, which is the same during the whole algorithm, according to our assumptions.

The big difference is that, using this strategy, we have less edges, which means less messages, less communication, less processing, less resources and less time. Now, we can run an adapted version of page rank in this representative and, we hope, significantly smaller graph.

4 Evaluation

We evaluated our solution, first of all, as a way to validate the design and the implementation itself, and in second place to show the potential of this approach, its versatility, and actual results.

The use of different datasets, streams, and output formats, shows that GraphApprox has a good level of abstraction. Callback methods provide dynamic behavior to the application, and can even be changed during processing. This way, GraphApprox may be plugged with some kind of high-level system which can reason about the changes to the graph and the results obtained, by means of statistical inference or machine learning, for example, and change the configuration of the application in order to achieve better results.

4.1 Data sets

We tested our solution with three real-world data sets: PolBlogs, a directed network of hyperlinks between weblogs on US politics (1490 vertices, 18091 edges) [1]; CitHepPh, Arxiv High Energy Physics paper citation network (34546 v., 421578 e.) [13]; Facebook, user-to-user links from the Facebook New Orleans networks (63731 v., 1545686 e.). [19].

Part of the datasets was used as initial graph, the remaining coming as updates. In the cases of CitHepPh and Facebook, edges have temporal information associated, so the updates were provided in the order defined by it. In the middle of updates, some queries were sent too.

We calculated first the exact PageRank, each time a query arrived. After that, we used our approximation algorithm with different parameters, and compared the size of the summary graph used and time spent.

To evaluate the results, we used Rank-Biased-Overlap (RBO) [20]. In the case of ranks, or generally when we have lists of items ordered by its importance, the measure should meet a relevant criterion: items in the top of the list should receive more importance. This means that, when comparing lists of this kind, similarities and differences on the top of the lists should have more impact than the ones at lowest levels. In order to have a meaningful evaluation, we can evaluate just the top of the list, say, the first 100 or 1000 items. But this way, when comparing two lists, some elements can appear in one of em but not in the other. Our measure must be prepared for that. We found that RBO meets those criteria and is quite efficient. Also, it represents a rather intuitive probabilistic model.

4.2 Experimental procedure

The first step was to prepare the data sets, selecting some part of the edge list as the initial graph. The remaining edges were included in streamed updates. Among the updates, some queries were inserted at random intervals.

First, for each query, we computed the exact result and gathered statistics relative to the time spent in computation. After that, we submitted the same data load to our approximated computation framework, and we evaluated the system in two big axes: the quality of the results, i.e. its correction; the resources spent in the approximate computation, namely the time spent on it. The approximate results were computed also with different sets of parameters, in order to verify its influence on result quality and resource use. All the calculations used the same Flink configuration.

4.3 Results

We started evaluating the impact of the parameter neighborhood size n on the quality of the results. As we can see in figure 2, using Facebook dataset, there is a significant difference between $n = 0$ and $n > 0$ for the value of RBO obtained in each iteration. At the same time, we can notice that the quality of the results obtained with $n = 1$ or $n = 2$ are very similar. This confirms the notion that, for PageRank, not only the most updated vertices are likely to have its rank changed, but also their immediate neighbors.

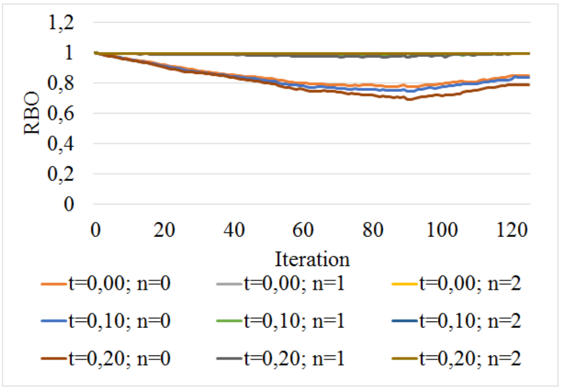

Fig. 2. RBO values with different n and t (Facebook)

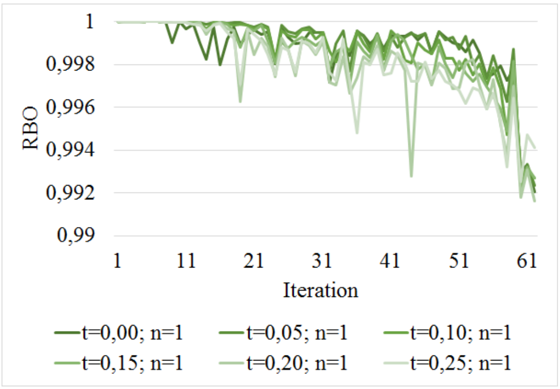

Fig. 3. RBO values for $n = 1$ and different t (CitHepPh)

Fixing now $n = 1$, we can observe in figure 3 the impact of different degree difference thresholds t on the quality of results. We can conclude, at least for the datasets used, that different values of t there is small impact on the RBO measured. An explanation for this is that the expansion of the original set of "hot" vertices with their immediate neighbors absorbs the impact of the value chosen for t.

We can now verify the impact of the approximation on the time and complexity of the computations. Figure 4 shows the results obtained with Facebook dataset with $n = 1$ and $t = 0.2$. We can observe that, with values for RBO always above 0.97, we were able to reduce computation time, a great part of the time, to values around 60% of the time needed to compute an exact response. But we can see also that this scheme does not always works: the last iterations presented a poor performance, indicating that further optimization should be done. That is the kind of decisions a system plugged to our API could make.

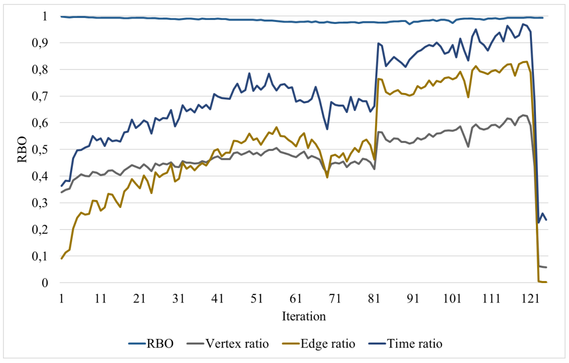

Fig. 4. Results for $n = 1$ and $t = 0.2$ (Facebook dataset)

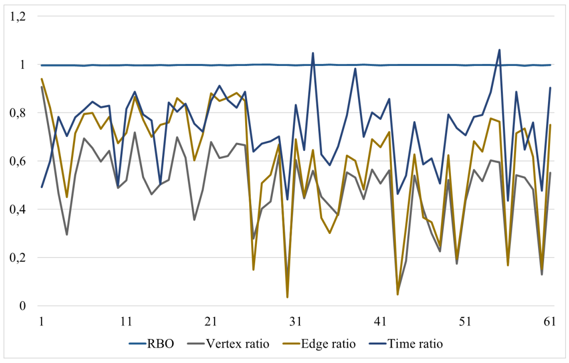

Fig. 5. Results for $n = 1$ and $t = 0.0$ (PolBlogs dataset)

Figure 5 shows the results for PolBlogs dataset, with $n = 1$ and $t = 0$. This is a smaller dataset, and that helps to explain the variations in the results, namely time ratio and proportion of vertices and edges.

Despite better quality of results when we set $n > 0$, figure 6 shows that, with $n = 0$, we can obtain substantial gains in performance and time with acceptable results. With CitHepPh dataset, $n = 0$, $t = 0.1$, we can see, especially in the last iterations, that we can achieve less than 50% of time spent with RBO values above 0.9. This shows some of the potentiality for optimization and versatility of the solution.

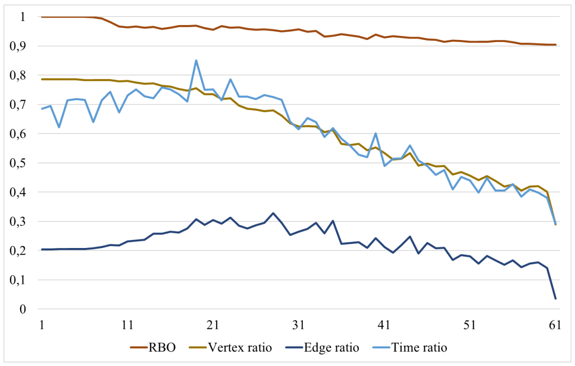

Fig. 6. Results for $n = 0$ and $t = 0.1$ (CitHepPh dataset)

From this examples becomes clear the intimate relation between the size of the graph used to compute the results and the time spent for it. The number of edges presents a direct relation with the time of the computation. This is quite expected, because the algorithm we used is based on messages between vertices: each one, at each step, sends one message to the direct neighbors.

These results show that the use of approximate processing, in this case approximating PageRank by means of graph summarization, based on tracking of updates, can be used to obtain useful results, while reducing the time needed to compute the response. This can be used to obtain more timely results, at the cost of some error introduced. The actual success of this approach depends on the dataset processed, and on the characteristics of the graph, and the stream of updates.

5 Conclusions and future work

In this paper, we saw how we can face the challenges of graph processing by means of approximate processing. We described the concept of a system that keeps track of the updates received by stream to a graph, and which is able to provide information, through an API, about the current state of the graph and statistics about the updates. The API lets the user decide, based on that data, to repeat the last answer, to compute an approximation or to fully recompute the exact answer.

To show the significance of this proposal, we tested approximate computation with some real world graphs and showed that it is possible to obtain responses with high quality and reduce resource consumption at the same time.

Future work on this project includes, at first, applying the concept to different types of algorithms, and also to different types of updates, not only topological, but also in properties associated to the graph. The next step would be plug some kind of intelligent system to our API, trying to extract patterns in data that allow to achieve further optimization.

References

[1] Lada A Adamic and Natalie Glance. “The Political Blogosphere and the 2004 U.S. Election: Divided They Blog”. In: *Proceedings of the 3rd International Workshop on Link Discovery.* LinkKDD ’05. New York, NY, USA: ACM, 2005, pp. 36–43. ISBN: 1-59593-215-1. URL: http://doi.acm.org/10.1145/1134271.1134277.

[2] Alexander Alexandrov et al. “The Stratosphere platform for big data analytics”. In: *The VLDB Journal* 23.6 (May 2014), pp. 939–964. ISSN: 1066-8888. URL: http://link.springer.com/10.1007/s00778-014-0357-y.

[3] Brain Brian Babcock et al. “Load Shedding Techniques for Data Stream Systems”. In: *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS.* 2003, pp. 1–3. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.1941.

[4] Albert-László Barabási. *Network Science.* Cambridge University Press, 2016. ISBN: 9781107076266. URL: http://barabasi.com/networksciencebook/.

[5] Albert-László Barabási and Réka Albert. “Emergence of Scaling in Random Networks”. In: *Science* 286.5439 (Oct. 1999), pp. 509–512. URL: http://www.sciencemag.org/content/286/5439/509.abstract.

[6] Alain Barrat, Marc Barthélemy, and Alessandro Vespignani. *Dynamical Processes on Complex Networks.* 1st. New York, NY, USA: Cambridge University Press, 2008. ISBN: 9780521879507. URL: http://www.cambridge.org/us/academic/subjects/physics/statistical-physics/dynamical-processes-complex-networks.

[7] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6.* OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 1–13. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[8] Sérgio Esteves et al. “Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets”. In: *Journal of Parallel and Distributed Computing* 79-80 (May 2015), pp. 52–66. ISSN: 07437315. URL: http://www.sciencedirect.com/science/article/pii/S0743731515000507.

[9] Joseph E Gonzalez et al. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.* OSDI’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096.

[10] Joseph E Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[11] Pili Hu and Wing Cheong Lau. "A Survey and Taxonomy of Graph Sampling". In: *CoRR* abs/1308.5 (Aug. 2013). URL: https://arxiv.org/abs/1308.5865.

[12] Amy Nicole Langville and Carl Dean Meyer. "Updating Pagerank with Iterative Aggregation". In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*. WWW Alt. '04. New York, NY, USA: ACM, 2004, pp. 392–393. ISBN: 1-58113-912-8. URL: http://doi.acm.org/10.1145/1013367.1013491.

[13] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations". In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. New York, NY, USA: ACM, 2005, pp. 177–187. ISBN: 1-59593-135-X. URL: http://doi.acm.org/10.1145/1081870.1081893.

[14] Yucheng Low et al. "GraphLab: A New Parallel Framework for Machine Learning". In: *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California, 2010. URL: http://arxiv.org/abs/1006.4990.

[15] Grzegorz Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. URL: http://doi.acm.org/10.1145/1807167.1807184.

[16] Jasmina Malicevic, Amitabha Roy, and Willy Zwaenepoel. "Scale-up Graph Processing in the Cloud: Challenges and Solutions". In: *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. CloudDP '14. New York, NY, USA: ACM, 2014, 5:1–5:6. ISBN: 978-1-4503-2714-5. URL: http://doi.acm.org/10.1145/2592784.2592789.

[17] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015. ISBN: 9781617290343. URL: https://www.manning.com/books/big-data.

[18] Nesime Tatbul et al. "Load Shedding in a Data Stream Manager". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 309–320. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315479.

[19] Bimal Viswanath et al. "On the Evolution of User Interaction in Facebook". In: *Proceedings of the 2Nd ACM Workshop on Online Social Networks*. WOSN '09. Dataset: http://socialnetworks.mpi-sws.org/data-wosn2009.html. New York, NY, USA: ACM, 2009, pp. 37–42. ISBN: 978-1-60558-445-4. URL: http://doi.acm.org/10.1145/1592665.1592675.

[20] William Webber, Alistair Moffat, and Justin Zobel. "A Similarity Measure for Indefinite Rankings". In: *ACM Trans. Inf. Syst.* 28.4 (2010), 20:1–20:38. ISSN: 1046-8188. URL: http://doi.acm.org/10.1145/1852102.1852106.

[21] Tom White. *Hadoop: The Definitive Guide*. 4th ed. O'Reilly Media, Inc., 2015. ISBN: 9781491901632. URL: http://hadoopbook.com/.

[22] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), pp. 2–2. ISSN: 00221112. URL: http://dl.acm.org/citation.cfm?id=2228301.