Advanced Internet Computing: Group 5 Topic Analyzing Big Data

Johannes Spießberger Ralph Hoch Carola Gabriel

ABSTRACT

Targeted advertising is one of the key revenue sources for internet services. While traditional approaches tried to suggest ads to users based on statistics derived from historical data, modern approaches try to make use of big data. This paper tries to give a short overview of current scientific trends in NoSQL and big data management. As graph databases are especially fitting for modeling social interactions, this paper puts an emphasis on this type of NoSql.

Keywords

Big Data, targeted advertising, NoSQL, graph database

1. INTRODUCTION

The recent years have shown a massive growth in data used for analysis in a broad range of fields. Big data as a concept for handling this amount of information has become a huge field for scientific research and business models alike. One of the cornerstones of the technical implementation of such systems is the usage of NoSQL databases. While these are available in many different flavors, graph based approaches are the one that differ the most from traditional database systems. Modeling social interactions as graphs and extracting various information is, among other applications, one of todays key techniques for targeted advertising [6]. While the usage of such graph databases makes for a natural abstraction of some real life observations, the technical implementations of the graph database itself becomes more challenging than those of traditional RDBMs system. The following sections will summarize some recent research concerning the inner workings of graph databases.

2. GPU BASED FREQUENT GRAPH MINING

With the availability of CUDA and open-cl GPUs have become a source of cheap computation power for significantly less money than general purpose CPUs.

While not applicable to all types of computational loads, there are various fields which benefit immensely from a high parallelization degree. Especially loads without the need of communication between the threads are well suited for GPU architectures. The following sections will describe the approach of [8] to introduce GPUs in graph databases.

Frequent graph mining describes the search of reoccurring sub patterns in a set of graphs. Among various applications this technique can be used to find similar communities within social networks. Figure 1 shows a pattern P that occurs in $G1$ and $G2$

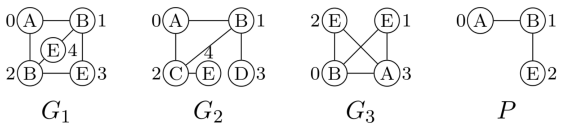


Figure 1: Graphs and pattern P [8]

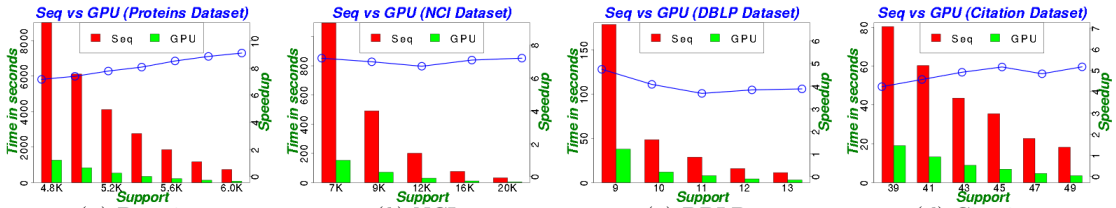
As this problem has been shown to be NP-complete [10] it is especially important to find scalable algorithms to deal with this problem. The GPU-based graph mining algorithm combines concepts from gSpan [13] and DMTL [2].

For the efficient pruning of duplicate subgraphs the DFS-code from gSpan is used. Parts of DMTL are used to store all isomorphisms for each pattern, which allows a fast computation for how many graphs in a graph database contain a certain subgraph. In order to achieve good performance it is necessary to change the data structure of the graph in main memory. Hashmaps or linked lists are not well suited for GPUs as data is not cached and therefore access locality must be considered for performance reasons.

The memory layout has to happen in a way that the id and neighborhood can be quickly looked up. To achieve this goal the combined neighborhoods of all vertices, edges and vertices are stored in separate arrays. A fourth array is introduced which holds offsets to speedup the lookup between arrays.

This memory layout including the implications of how data is looked up by gSpan and DMTL allow for a very efficient parallelization on GPUs. For various benchmarks this technique led to significant speedups. Figure 2 shows the performance comparison between a GPU and a sequential implementation for frequent graph mining on multiple datasets.

The principles of this research can also be extended to additional graph algorithms like closed graph, maximal graph and temporal graph mining.



(a) Protein (b) NCI (c) DBLP (d) Citation

Figure 2: GPU Benchmarks [8]

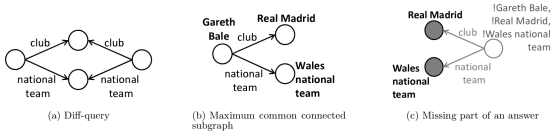
3. DIFFERENTIAL QUERIES

Abandoning the rigid predefined schema of traditional database systems is one of the key features of NoSQL. While offering more flexibility in terms of development, users have a hard time gaining deep knowledge of the data and its structure. This mostly manifests in queries giving unexpected or empty result sets.

The research provided by [11] describes an approach to mitigate this problem. Diff-queries provide users with the information about which part of a query actually matches data in the graph and which parts lead to an empty result set.

A query in a graph database can be understood as a pattern that has to match parts of the graph. These pattern can be modeled as graphs themselves. In order to be able to determine which parts of a query provide a match and which parts will finally lead to an empty result, a common connected subgraph algorithm is applied to the query and the graph data itself. The resulting common subgraph can then further be used to partition constraints in a query into matching and non-matching conditions.

Figure 3 visualizes the basic idea of differential queries. Assume a query that searches for two soccer players that play for the same club and are in the same national team. If the graph database has a player that matches parts of the query conditions, namely playing for a national team and a club, but there is no second player with the same attributes, the resulting difference graph will contain the non matching conditions.



(a) Diff-query (b) Maximum common connected subgraph (c) Missing part of an answer

Figure 3: Differential graph for data and query [12]

Due to the nature of these operations are large number of intermediate results may lead to performance problems.

This problem may be mitigated by introducing Top-K differential queries [12]. While modeling the query of a graph the user is given the option to weight certain edges, vertices or subgraphs according to his preferences. By using a new algorithm called relevance flooding, the specified weights are attached to the data in the graph database. When calculating the maximum common subgraph, not the number matching edges and vertices is maximized, but the score resulting from the weights given by the user.

4. SLQ

Another approach of easing the writing of graph database queries for non professional users is the SLQ language [14] [15].

A property graph model is used for formulating queries. Aside from having nodes and edges there are key value pairs associated with each node. These key value pairs will be further treated as keywords.

Queries are then further transformed using a library L consisting of transformations functions. Those may include simple string replacements, semantic transformations, numerical and topological transformations. These functions allow for the compensation of spelling errors or the usage of synonyms.

In an SLQ query a match in the data is derived from the mapping of a query, including possible transformation, to the data itself. It is therefore necessary to define a weight for such transformations to only show the most relevant results. As equal or predefined weighting for all transformations may not lead to an optimal query outcome, SLQ is able to learn and further improve the relevance of its transformations.

The first possibility is to use existing query logs of real users during an offline learning stage. As a high quality pair of queries and their results may not always be available an online learning approach is added. During the online operation, users can specify a number k which will then be used to display the most relevant results after applying a set of transformation. Be receiving feedback from the user SLQ is able to further improve the rankings of the transformations. Figure 4 shows the basics of this learning process.

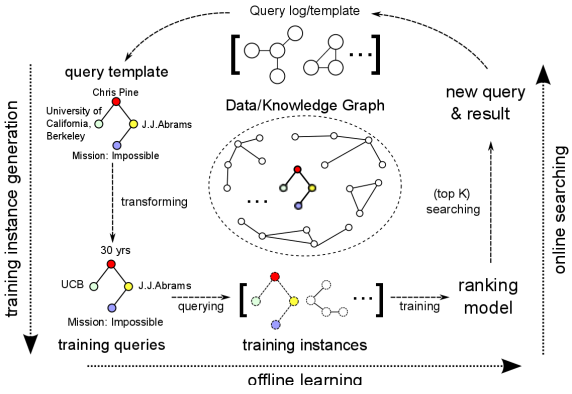


Figure 4: SLQ: graph querying framework [14]

Figure 5 gives an example for three possible queries and

their final results. The second query tries to find a person serving in the union army and was in a battle. By introducing the node Missouri the user specifies that it may be somehow related to the first conditions.

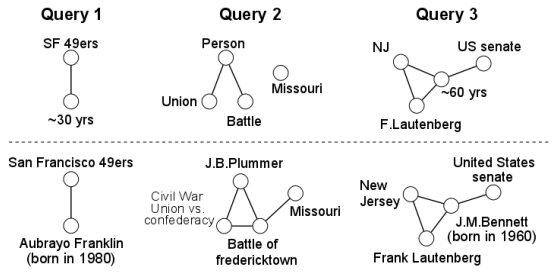


Figure 5: SLQ query examples [14]

5. INTERACTIVE PATH QUERY SPECIFICATION

In addition to the techniques above [1] specifies another way to enable users who are not familiar with any formal syntax to specify queries. GPS is a system for interactively specifying graph path queries. The basic idea is to let the user define nodes of interests and nodes that should be excluded from the result in a graphical environment. This process happens iteratively while the systems gives feedback and helps to remove unnecessary conditions from the query. The iteration may either stop when there exists exactly one query or the user is satisfied with the results.

Additional nodes are proposed on the basis of strategy functions which take the existing graph and the provided sample nodes. While a node itself may not contain enough information for the user to decide if it is interesting or not, GPS shows the neighborhood to help the user with his decision.

Figure 6 shows the interactions with the user when constructing a query.

6. PARALLEL ADJACENCY LISTS

Modeling social networks like Twitter has some particular challenges for graph DB which come from the uneven distribution of edges between nodes. Such graphs have the tendency to have a small number of nodes with a high number of edges and a high number of nodes with a small number of edges [7]. A a standard procedure for partitioning is to cut the graph, but in a scenario with such uneven edge distribution this becomes extremely challenging.

Partitioned Adjaceny Lists PAL [9] tries to reduce the number of random accesses while minimizing the storage space required. Edges are stored as pairs of vertices and are partitioned by a range of destination vertices and are ordered by the source ID. These partitions do not necessarily be of equal length, but should be chosen such that a partition can fit into main memory. Note that this may theoretically limit the number of incoming edges a vertex may have.

As the graph connectivity is fully represented by the edge partitions, vertex data can be stored separately and is partitioned based on the edge partitions. Retrieving vertex data can be achieved by simply calculating an offset from the edge partitions.

Edge partitions are immutable data structures and do not allow for direct insertion of edges. To counter this problem new edges a stored in a buffer and inserted if their number exceeds a certain threshold. During search operations these buffers are also considered.

Queries are built on top of primitive operations that return in or out edges of a vertex. Finding in-edges is straight forward due to the partitioning. Due to the fact that out-edges for a given node may be present in all partitions their retrieval is more expensive than in existing implementations. On the other hand the PAL structure gives improved performance in other access and writing operations.

While comparing performance for directed unweighted shortest path algorithms it has been shown that while Neo4J is up to 50 times faster than GraphChi-DB it should be noted that huge dataset, that follow the uneven distribution of incoming edges, Neo4J was not able to finish the query due to memory restrictions. While not directly measured it is expected that Neo4J in similar systems will outperform GraphChi-DB but as shown in the benchmark are not fit for natural graphs.

7. DISTRIBUTED GRAPH DATABASE

Current graph databases are mainly intended to run on a single instance and therefore face performance and availability problems when employed in large scale settings [3] [4].

Acacia [5] is a distributed graph database intended to run in hybrid cloud configurations. Hybrid clouds a characterized by running mainly in a private network and only some time are available to the public to ensure good performance. The basic system design relies on a combination of a Master-Worker pattern and graph partitioning. While a P2P approach would lead to a higher scalability the Master-Worker approach allows for better controlling in settings with less than 100 workers. The partitioning, while an expensive operation, reduces the need for inter node communication which is necessary for exchanging information concerning nodes which lie outside of a graph partition. The number of partitions corresponds to the number of workers.

Each distributed process has an associated local data store with an assigned storage quota. When adding a new graph the partitioning happens via Hadoop and Metis and the subgraphs are then inserted into local Neo4J instances. Figure 7 shows the underlying design of Acacia, note that Hadoop and Metis are only relevant for graph partitioning and are not used during other activities.

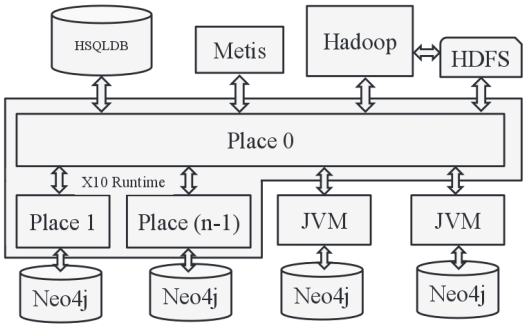


Figure 7: Acacia: system overview [5]

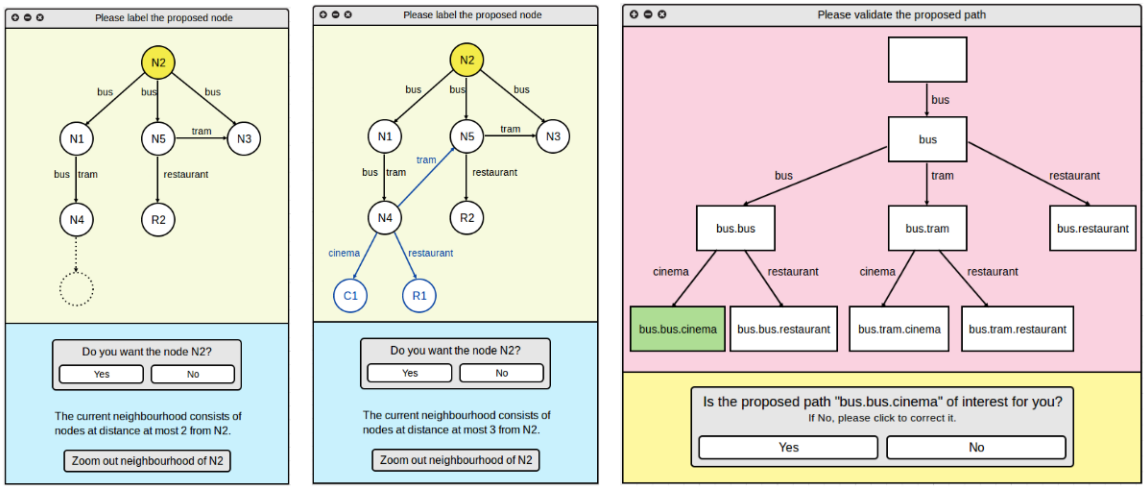


Figure 6: Example of GPS interaction with user [1]

Among other implemented algorithms a benchmark for the average degree distribution was run on 4 workers. Figure 8 shows the elapsed time in comparison to the number of edges in a graph.

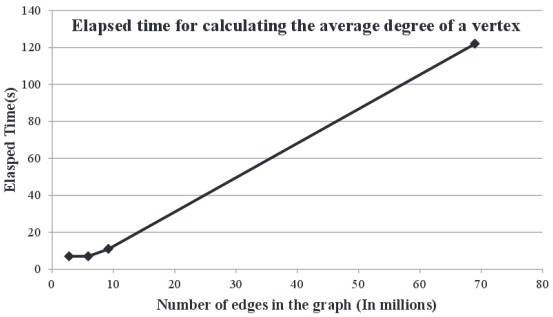


Figure 8: Acacia: Benchmark [5]

The paper concludes that Acacia is able to handle medium size graph data, but the loading process and the included graph partitioning are the limiting factors.

8. REFERENCES

[1] A. Bonifati, R. Ciucanu, and A. Lemay. Interactive path query specification on graph databases. EDBT, 2015.

[2] V. Chaoji, M. Al Hasan, S. Salem, and M. J. Zaki. An integrated, generic approach to pattern mining: data mining template library. *Data Mining and Knowledge Discovery*, 17(3):457–495, 2008.

[3] M. Dayarathna and T. Suzumura. Xgdbench: A benchmarking platform for graph stores in exascale clouds. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 363–370. IEEE Computer Society, 2012.

[4] M. Dayarathna and T. Suzumura. Graph database benchmarking on cloud environments with xgdbench. *Automated Software Engineering*, pages 1–25, 2013.

[5] M. Dayarathna and T. Suzumura. Towards scalable distributed graph database engine for hybrid clouds. In *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*, pages 1–8. IEEE Press, 2014.

[6] W.-S. Y. J.-B. Dia and H.-C. C. H.-T. Lin. mining social networks for targeted advertising. *System*, 2006.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[8] R. Kessl, N. Talukder, P. Anchuri, and M. J. Zaki. Parallel graph mining with gpus. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 1–16, 2014.

[9] A. Kyrola and C. Guestrin. Graphchi-db: Simple design for a scalable graph database system–on just a pc. *arXiv preprint arXiv:1403.0701*, 2014.

[10] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[11] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner. Graphmcs: Discover the unknown in large data graphs. In *EDBT/ICDT Workshops*, pages 200–207, 2014.

[12] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner. Top-k differential queries in graph databases. In *Advances in Databases and Information Systems*, pages 112–125. Springer, 2014.

[13] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 721–724. IEEE, 2002.

[14] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment*, 7(7), 2014.

[15] S. Yang, Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu, and X. Yan. Slq: A user-friendly graph querying system. 2014.