On Compressing Graph Databases

Antonio Maccioni
Roma Tre University
maccioni@dia.uniroma3.it

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Distributed and parallel databases are not always a scalable solution for graphs because of the low locality in the data matching the query. A different approach for scalability is to consider compressed versions of the input graph in order to reduce the cost of join operations at query time. These methods proved to be effective when the queries are given in advance but are not general enough to be adopted since we cannot reconstruct the original graph with decompression. This means that the results of the query are approximations of the exact answers. We want to discuss with the audience a novel compression technique that enables the database engine to evaluate graph queries (e.g., graph pattern matching queries) in an exact way. Preliminary experiments show that our approach is scalable with respect to the size of the graph and that this method can be employed as a layer over existing database systems with minimal effort.

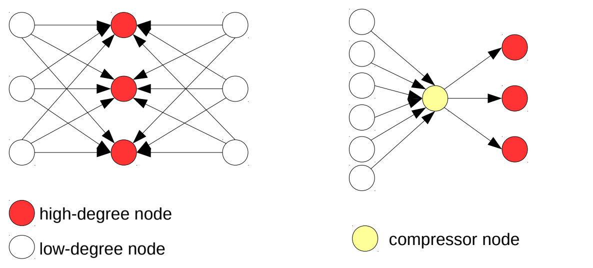
1. OVERVIEW

The impossibility to reduce the complexity of the algorithms for graph query answering (e.g., graph pattern matching) has pushed many researchers to devise alternative solutions for improving performance. One of them is to reduce the size of the input by transforming the original graph G into a smaller graph G_c [2, 5, 4]. The work in [5] proposes a compression for the graph that preserves the query answering of a class of queries. Intuitively, the portion of the graph that would be untouched by those queries is pruned. This is a *lossy* compression of the graph which is, unfortunately, unsuitable for database systems that are designed for users who demand exact answers to their graph queries. R2G [4] generates graphs from relational databases reducing the number of nodes and consequently the cost of I/O with respect to the competitor graph database management systems. This method works only for relational databases modeled through property graphs [4]. Another way to reduce the size of the graph is by *graph summarization*, which is accomplished by sampling graph G into a summary [2]. The summary embeds structural information about the graph and can be helpful in the optimization of the query plan but not computing queries directly over G.

Conversely, we use a general and lossless compression technique that greatly reduces the edgeset of the graph and also adds few new nodes to G_c. Moreover, we integrate a compression mechanism with the query execution [1]. Our approach avoids the need to use approximate algorithms for graph querying [3], improving, at the same time, scalability and practicability of graph database management systems.

2. SPARSIFICATION OF GRAPHS

The scalability of graph pattern matching systems is limited by the presence of high-degree nodes. In fact, most real-world graphs follow a *power-law* [7], which yield a few nodes that are connected to a large number of other nodes. We consider high-degree, the nodes having a number of incoming edges exceeding a given threshold τ. Query processing operations involving high-degree nodes are extremely skewed, taking far more time than equivalent operations on "low-degree" nodes since they produce an explosion of the number of intermediate results at query time. The problem only gets worse as the graph evolves: the degree of such nodes increases linearly or super-linearly with respect to the increase in graph size, and new nodes become high-degree.



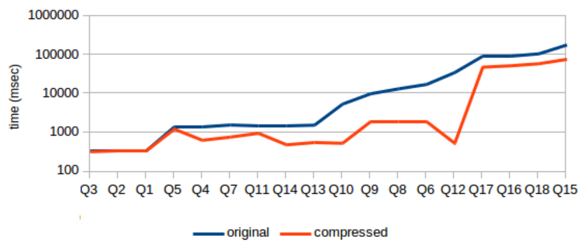
(a) Original graph (b) Compressed graph

Figure 1: Sparsification.

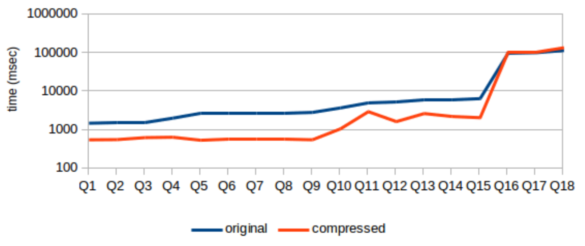
We present a *sparsification* technique that losslessly compresses the structure of the graph in order to reduce the number of adjacencies of high-degree nodes. The main intuition behind our work is that there is a large amount of information redundancy surrounding high-degree nodes that can be synthesized and eliminated. We can find clusters of low-degree nodes that are connected to the same group of high-degree nodes (*e.g.*, in a social network, people who like rock music tend to *follow* a highly overlapping group of popular rock stars). To this end, we introduce special nodes in the graph, that we call *compressor nodes*, representing common connections of clusters of related nodes to high-degree nodes. A lot of redundant edges can, in this way, be removed from G_c. For example, in the graph of Figure 1(a), six low-degree nodes (in white) are connected to the same three high-degree nodes (in red). Therefore, we can say that there exists a general "type of node" that has outgoing edges to this particular set of three high-degree nodes. We indicate that this "type of node" exists by creating a new node (shown in yellow on Figure 1(b)) that has outgoing edges to this set of three high-degree nodes. Then, we remove the edges that connect the two white nodes to this set of high-degree nodes, and instead create a single edge from each white node to the new yellow node. This new yellow node is called a "compressor node".

3. QUERYING SPARSIFIED GRAPHS

The compression presented in the previous section allows to query directly on the compressed graph, without ever having to decompress the data. For this purpose, we defined join operators for graph pattern matching over G_c that are equivalent to computing joins over G. In particular, we follow the recent developments in the state-of-the-art that decompose the query into stars and then join together partial star results at the end of the querying process [6]. A star query $q^{(i)} = (s, o_1, o_2, \ldots, o_n)$ contains a source node s and one or more destination nodes o_i at 1-hop from s, where $s, o_1, \ldots, o_n$ can also be variables. We cannot show here the pseudo-code of the join algorithm for the sake of limited space, but intuitively, we first try to compute partial results involving only high-degree nodes that are implicit or explicit in the query. These results are easy to compute through the compressor nodes. Then, we have to enrich the set of partial stars by adding edges involving low-degree nodes only. Since the size of the whole graph is reduced and the intermediate results produced by edge-joins through high-degree vertexes decrease, the performance of graph pattern matching significantly improves.



(a) No Indexing



(b) Exhaustive Indexing

Figure 2: Experimental Cold-cache Results.

Since the most efficient solution to persist graph databases and query them through pattern matching is represented by the so-called triple stores, we have developed a triple store prototype on PostgreSQL 9.1 to evaluate our approach. All the algorithms, including the state-of-the-art counterpart, have been developed on stored procedures written in PL/pgSQL. We have run different campaigns of experiments including different kind of star queries of different size (i.e. with only high-degree nodes, with only variables or with a mixed fan-out of variables and high-degree nodes). We also have performed both cold-cache and warm-cache runs and tried different indexing schemes on the database. We delegated to the DBMS the optimization of the query answering without interfering with any decision. We have outperformed the algorithms of the state-of-the-art in all the campaigns. Figure 2 shows the comparison between the state-of-the-art approaches (i.e. original) and our (i.e. compressed) with star pattern matching queries of different complexity and size. We used a publicly available dataset of Twitter connections[1] that we compressed using $\tau = 2500$. We chosen this threshold value because it resulted the most efficient after fine-tuning. Figure 2(a) shows the performance of cold-cache experiments when no indexing scheme is applied on the triple store. Figure 2(b) shows cold-cache performance with the indexing scheme used by the majority of RDF stores, where every permutation of triples is indexed [6]. Experiments demonstrate that our approach facilitates scalable query processing over large graphs and can work in conjunction with indexing to further improve query performance.

4. FUTURE WORK

We have several directions of future work. The presented approach works with general graphs but we would like to verify if further optimizations apply to more specific graphs (*e.g.*, RDF datasets and bio-informatics graphs). We also want to test if this technique can improve graph partitioning. Moreover, it would be interesting to enable an automatic fine-grained tuning that decides the best τ according to the structure of the graph and the query load of the system. Finally, we would like to investigate if other type of query answering (*e.g.*, reachability and shortest path) over real large-graphs benefit from our compression.

5. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.

[3] R. De Virgilio, A. Maccioni, and R. Torlone. Approximate querying of rdf graphs via path alignment. *Distributed and Parallel Databases*, pages 1–27, 2014.

[4] R. De Virgilio, A. Maccioni, and R. Torlone. R2G: a tool for migrating relations to graphs. In *EDBT*, pages 640–643, 2014.

[5] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[6] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, pages 439–450, 2014.

[7] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1), 2007.

[1] http://snap.stanford.edu/data/