Implementing Graph Query Languages over Compressed Data Structures: A Progress Report

Nicolás Lehmann[1,2] and Jorge Pérez[1,2]

[1] Department of Computer Science, Universidad de Chile
[2] Chilean Center for Semantic Web Research
[nlehmann,jperez]@dcc.uchile.cl

Abstract. In this short paper we present our preliminary results on implementing two-way regular-path queries (2RPQs) over a compressed representation of graph data. We report on several experiments comparing our approach with state-of-the-art graph database engines. Our results are encouraging; although we use a naive implementation for 2RPQs, our system exhibits a competitive performance compared with other engines.

1 Introduction

Graph databases have recently gained a lot of attention in theory and practice. This can be explained by the current need of handling Web-related data, such as on-line social networks, and RDF and Semantic Web data. One of the most important challenges in this context, is the need of handling Web-scale amounts of data, while providing a reasonable expressiveness for users that want to explore this data.

In this short paper we present our preliminary results on implementing two-way regular-path queries (2RPQs) over a compressed representation of graph data. We use 2RPQs as they are an expressive language capable of navigating graphs by using paths defined by regular expressions, using forward and backward edges while navigating the graph. 2RPQs are in the core of recently proposed standards for handling RDF data [6]. To compress the graph data we use the k^2-tree data structure [4]. Given a node v, a k^2-tree representation allows us to access the neighbors of v, as well as the nodes pointing to v, in a very efficient way. This feature plus the compression ratio of k^2-trees which permits to maintain the structure of huge graphs in main memory, implied a critical performance gain when implementing 2RPQs.

We implement a naive algorithm for 2RPQs based on the typical automata-theoretic approach [8]. Even with this naive implementation, our system exhibits a competitive performance compared with state-of-the-art commercial engines. We perform several experiments with data generated by the GDBench tool [2], comparing our implementation with Sparksee [7] (formerly known as DEX) and Neo4j [9]. Our system and Sparksee show a similar performance and Neo4j is considerably surpassed by both alternatives. Our solution also exhibits a considerable advantage in a *cold scenario* where the structures have just been loaded and the system is running for the first time.

2 Background and implementation details

Graph databases and query languages We consider a simple model of a graph database as just a graph $G = (V, E)$ in which every element in V is a node ID (or just node for short), and each edge is a triple (v_1, e, v_2) where $v_1, v_2 \in V$ and e is an edge label from an alphabet Σ. We say that (v_1, e, v_2) is a *forward e-edge* from v_1 to v_2. Symmetrically, (v_1, e, v_2) is a *backward e-edge* from v_2 to v_1. As a query language, we consider two-way regular-path queries (2RPQs) which are essentially regular expressions over $\Sigma \cup \Sigma^-$, where $\Sigma^- = \{e^- \mid e \in \Sigma\}$ is the alphabet of *backward edges*. Given a 2RPQ r, a pair of nodes (v_1, v_2) is in the evaluation of r over G, if there exists a path in G from v_1 to v_2 following forward and backward edges, such that the sequence of labels of the path belongs to the regular expression defined by r considering each backward e-edge traversed as the symbol e^-. For example, consider the 2RPQ $r = a/(b^-)^*/c$ and a graph G with edges $(v_1, a, v_2), (v_3, b, v_2), (v_4, b, v_3), (v_4, c, v_5)$. Then we have that (v_1, v_5) is in the evaluation of r over G.

K^2-trees A k^2-tree [4] is a tree-shaped structure for representing graphs that exploits sparseness and clustering features of the adjacency matrix associated to the graph. Given an adjacency matrix, a k^2-tree divides it into k^2 submatrices of the same size. Each submatrix is represented in the tree as a child of the root. For the submatrices containing only 0's the decomposition ends there, using a single 0-node to represent the whole submatrix. The submatrices with at least one 1 are recursively decomposed using the same strategy until an actual cell in the matrix is reached, which is stored as a 0- or 1-node in the last level. The tree is then implemented in a highly compacted way using *bitstrings*; every level of the tree is represented as a *bitstring* and the whole tree as the concatenation of them. Given a node v, searching for the neighbors of v as well as for the nodes pointing to v, can be achieved by just traversing the k^2-tree [4]. The traversal of the tree can be simulated using *rank queries* over the *bitstrings*, which can be implemented very efficiently [5]. Thus, the whole k^2-tree can be represented in a succinct manner while maintaining its traversal properties. Further optimizations are possible, for example, using different values of k for different levels of the tree or stopping the decomposition when the matrices reach size $k_L \times k_L$ and use DACs to compress them [3].

Design and implementation details Let $G = (V, E)$ be a graph database over alphabet Σ. To simplify the correspondence between node IDs and rows and columns in an adjacency matrix representation, we first map every node ID in V and label in Σ to an integer via a dictionary encoding[3]. After the encoding, our design continues by *vertically partitioning* the data, reorganizing it into $|\Sigma|$ independent graphs, each graph containing only edges with a particular edge label. Then the whole graph is represented as an array of k^2-trees, each tree representing the graph induced by a particular edge label. Given a node v and an edge label e, we compute the direct or inverse e-neighbors of v by traversing the k^2-tree corresponding to e. Following the configuration of similar work [1], the k^2-trees we use for evaluation follow a hybrid policy using $k = 4$ for the first 5 levels and $k = 2$ for the rest. The decomposition stop when the submatrices reach size 8×8 and are encoded using DAC's.

[3] The implementation of the dictionary is orthogonal to our proposal and thus it is not considered in our evaluation in Section 3.

The evaluation of the 2RPQs follows a simple algorithm using the typical automata-theoretic approach [8]. Given a 2RPQ r, we first build the Non-deterministic Finite Automaton (NFA) associated to r, considering labels in Σ and inverse labels. Then, the graph is also considered as an NFA and the algorithm performs a breadth first search over the product automaton. In practice the product automaton cannot be constructed, but we perform the traversal implicitly. Thus the algorithm only needs to know neighbors of a node by a single label (or an inverse label) at a time, which can be efficiently computed with the k^2-tree representation as explained above.

The code is implemented in `C++` and available via Github.[4]

3 Experimental results

We compare our implementation with Sparksee [7] (version 5, February 2014) and Neo4j [9] (version 2.1, July 2014), using a machine with the following configuration: 3.40 GHz Intel Core i7-2600k (4 cores), 8 GB RAM, Archlinux OS kernel version 3.18.4. We compare the running time for several 2RPQs considering two evaluation scenarios: the *warm* and the *cold* scenarios. The warm scenario simulates the conditions of an already running server: we first perform a warm-up run, and then report the results for the second run (of the same query). The cold scenario reports the running time of the first run. The idea is to analyze how caching influences the performance. For every query tested, we run 10 000 experiments and report the average time.

In our experiments, we use the data generator provided by the graph database benchmark GDBench [2]. Graphs generated by GDBench have a simple social network structure with nodes representing persons and webpages, friend-edges between persons, and like-edges from persons to webpages. We considered graphs of different size ranging from 10 million to 40 million nodes.

Figs. 1-3 present a comparison for queries `like`, `friend/friend`, and `like/like`$^-$ in the warm scenario. Our implementation is labelled as k2tdb in the figures. For these queries, k2tdb and Sparksee show a similar performance (running times are in the same order of magnitude), while Neo4j is considerably slower. Notice that for queries involving only like-edges, k2tdb has a performance twice as good as Sparksee in a warm scenario (Fig. 1 and 3). This is consistent with the characteristics of k^2-trees which are specially suited for sparse graphs, and the subgraph of like-edges enjoys this feature.

Our next experiment considers navigational path queries which are one of the most important features of 2RPQs. Informally, we consider queries that goes from one person to their set of friends, and the friends of its friends, and so on, for several steps. More formally, we consider the queries `friend`, `friend/friend`, `friend/friend/friend`,... until five copies of the friend-edge. These queries are denoted by f1, f2, f3, f4, f5, respectively. We also test the query `friend*`, denoted by f*, which allows to navigate an arbitrary number of friend-edges. We report on the results for a graph with 20 million nodes (Fig. 4 and 5). In the warm scenario Sparksee slightly outperforms our implementation (Fig. 4), but both stays within the same order of magnitude. For the cold scenario our implementation has a better performance (Fig. 5), and the difference is

[4] https://github.com/nilehmann/libk2tree, https://github.com/nilehmann/k2tdb

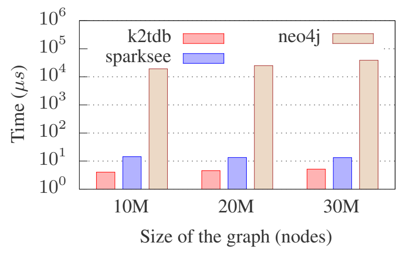

Fig. 1: Running time for query `like`

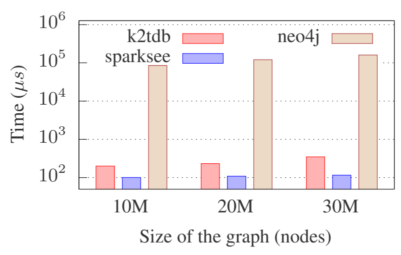

Fig. 2: Running time for `friend/friend`

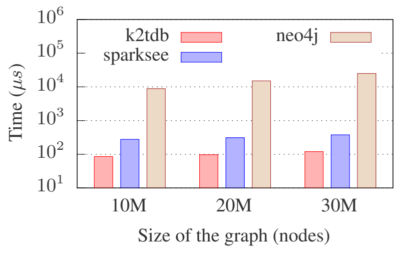

Fig. 3: Running time for `like/like`$^-$

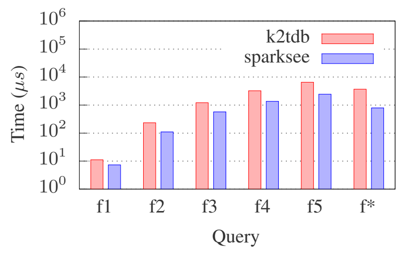

Fig. 4: Path queries in warm scenario

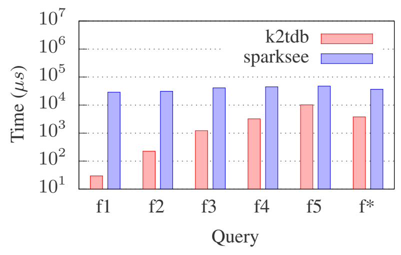

Fig. 5: Path queries in cold scenario

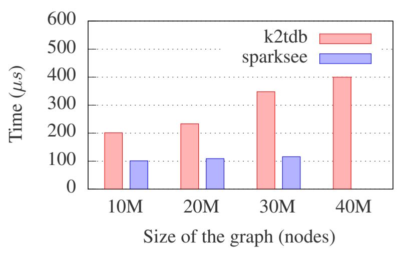

Fig. 6: Scalability test for `friend/friend`

quite substantial for the simpler queries. This behavior can be explained by the caching techniques used in Sparksee. As the portion of the graph being traversed gets larger, the probability of using the cache increases. Our solution best suits a scenario where the explored portion of the graph has not yet been visited. This presents an interesting opportunity for improving our implementation by using similar ideas for caching, for example, by decompressing and caching some portions of the graph as we traverse it.

Our last experiment is a scalability test for query `friend/friend` over graphs of increasing size (Fig. 6). The Sparksee license that we use, allows graphs with at most 1 billion objects (nodes plus edges), which disallows the loading of the 40M-node graph. Thus, we show the time up to 30M nodes for Sparksee. The growth in running time for k2tdb is more pronounced compared with Sparksee, but it still shows a linear behavior. Further experimentation with larger graphs is needed to obtain specific conclusions.

4 Conclusions and future work

Our naive implementation of 2RPQs over compressed graph structures shows a competitive performance compared with highly-optimized graph database engines. This shows the benefits of considering compressed data structures when querying graphs with expressive query languages. Our implementation shows a particularly good performance in the *cold scenario* where no caching is permitted. This presents an interesting opportunity for optimizing our implementation with caching techniques. Our ongoing work includes the implementation of 2RPQs in a less naive way, taking a more specific advantage of the way the graph is actually compressed.

References

1. Álvarez-García, S., Brisaboa, N., Fernández, J., Martínez-Prieto, M., Navarro, G.: Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems (2014), to appear
2. Angles, R., Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.L.: Benchmarking database systems for social network applications. In: GRADES. p. 15 (2013)
3. Brisaboa, N., Ladra, S., Navarro, G.: DACs: Bringing direct access to variable-length codes. Information Processing and Management (IPM) 49(1), 392–404 (2013)
4. Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. Inf. Syst. 39, 152–174 (2014)
5. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Wea 2005. pp. 27–38
6. Harris, S., Seaborne, A.: Sparql 1.1 query language. W3C Recommendation (2013)
7. Martínez-Bazan, N., Gómez-Villamor, S., Escale-Claveras, F.: DEX: A high-performance graph database management system. In: ICDE Workshops 2011. pp. 124–127
8. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. In: VLDB 1989. pp. 185–193 (1989)
9. Webber, J.: A programmatic introduction to neo4j. In: SPLASH 2012. pp. 217–218