Flexible Query Answering over Graph-modeled Data

Antonio Maccioni
Roma Tre University
maccioni@dia.uniroma3.it
(supervised by Riccardo Torlone)*
Expected graduation date: Fall, 2015

ABSTRACT

The lack of familiarity that users have with information systems has led to different flexible methods to access data (keyword search, faceted search, similarity search, etc.). Since flexible query answering techniques differ from one another, their integration in the same system is hard. Flexible query capabilities require, in fact, ad-hoc representations of the datasets, which often result in duplications and computational overhead. Moreover, if we want to query heterogeneous data sources, the problem becomes almost impossible. To address such variety in one fell swoop, we propose a meta-approach for different kinds of flexible query answering over heterogeneous data sources. We consider structured and semi-structured sources that can be modeled through graph databases. To improve the platform storing the data, we have conducted research on the representation of graph databases. In particular, we have devised a layer that compresses the graph database without having to decompress it back into the original graph before query execution.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems - Query processing

Keywords

Flexible queries; Keyword search; Graph database modeling; Graph compression; Approximate graph querying

1. INTRODUCTION

The main purpose of Information Systems is to satisfy the information need of any kind of user. This achievement becomes more challenging when users are unaware of the content and organization of an underlying database or when they ignore query languages. This scenario is quite frequent, as many data access points are directly exposed on the Web to random users. In these cases, Information Systems rely on flexible query answering capabilities to take away the barriers that these non-expert users encounter when searching for information. In the evaluation of flexible queries, we want the "best" answers (usually called top-k) that might, possibly, only match the query approximately (*approximate query answering*), or we want to consider queries expressed with lack of structural constraints (*relaxed query answering*). An example of relaxed query answering is the Google-based method of searching over the Web, where users input their keywords of interest and the engine attempts to find the more relevant web pages accordingly.

Since flexible query answering methods differ from one another and traditional data integration techniques do not apply to them, their co-existence in the same database system is difficult. Flexible query capabilities require, in fact, ad-hoc representations of data, which are duplicated with respect to the dataset used for current operations. Maintaining two versions of the same database is clearly an overhead that should be avoided. Moreover, since in a typical scenario, the organizations's information is scattered on several sources that are governed by different heterogeneous systems, the problem becomes almost impossible. Let us imagine a B2B company keeping contracts in the form of unstructured data, storing customer data in a relational database and managing the information of the provided web services with XML documents. It turns out that in order to answer queries like "which are the cities of the customers who purchased a service of a given category during a given day", the informative system of the company should be re-engineered. This scenario is common whenever users want to be able to search multiple types of data generated by an enterprise through a simple search interface (Enterprise Search). These issues are emphasized with flexible answering since, in this case, the core of the problem is the lack of information in the query, which induces an uncertainty that brings overhead in the process. For instance, in relational keyword search, the uncertainty leads to an enormous number of join operations that the RDBMS has to compute in order to generate the answers [17]. Therefore, it is complicated for an Enterprise Search tool (e.g., an Enterprise Search Engine such as Microsoft Azure Search, Google Search Appliance, etc.) to support different flexible query answering capabilities. As a matter of fact, the current implementations of flexible query capabilities on Informative Systems are non-interoperable, each one using an ad-hoc representation of the dataset.

To address such heterogeneity in one fell swoop, we propose a meta-approach, called FleQSy, for different kinds of flexible query answering over semi-structured and structured data. FleQSy pools the commons operations of answering flexible queries by adopting a high-level abstraction of

SIGMOD'15 PhD Symposium, May 31, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3529-4/15/05 ...$15.00.
http://dx.doi.org/10.1145/2744680.2744686.

*Part of the work has been supervised by Daniel Abadi (Yale Uni) the process. In this way, the framework FleQSy integrates methodologically how different flexible query answering are solved. It allows us to avoid data transformations that bring unnecessary duplication and computational overhead.

We also model uniformly our structured and semi-structured data using graph databases so that FleQSy has the view of only one informative source. The framework has been applied to solve keyword query answering over relational [8] and RDF data [5], and to solve approximate graph matching over RDF databases [7].

Modeling our data sources with graphs, we came across several scalability issues in the context of graph databases. Disk-based graph databases are conceived using legacy technologies, such as those for structured data, which are clearly not well optimized for graph data. The problem of scaling queries over graphs is fundamentally harder than scaling queries over relational data. Therefore, we relief the system's workload by employing a layer that compresses the database. It enables direct querying of the compressed graph, without having to decompress it back into the original graph before query execution. We also introduce an approach that facilitates the adoption of graph databases in scenarios where traditional databases and methodologies are predominant. It migrates automatically data and queries from relational to graph databases.

The remainder of the paper is organized as follows. In Section 2, we explain the work conducted on graph data management. In Section 3 we propose the unified approach FleQSy to answer different forms of flexible queries over semi-structured and structured data. Section 4 discusses related research and Section 5 briefly sketches the results obtained so far. Finally, Section 6 draws conclusions and future research to follow-up the work of this thesis.

2. GRAPH DATA MANAGEMENT

Graph databases are rapidly emerging as an effective and efficient solution to the management of very large data sets in cases where data are naturally represented as a graph and data accesses mainly rely on traversing this graph. Many issues arise during the management of graph databases since they are fundamentally more complex than other databases. We tackle them separately in the following.

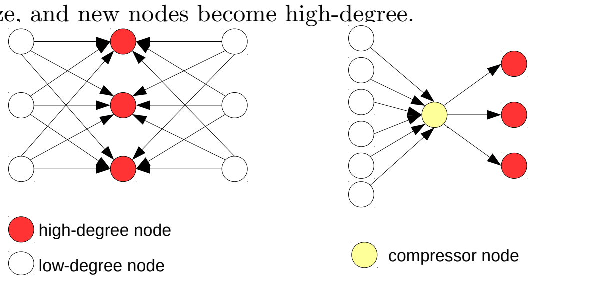
Modeling Graph Databases. The design of graph databases is based on best practices, usually suited only for a specific management system. We propose a model-driven, system-independent methodology for the design of graph databases modeled using property graphs [9]. Starting from a conceptual representation of the domain of interest expressed in the Entity-Relationship model, we propose a methodology for devising a graph database in which the data accesses for answering queries are minimized. This is achieved by aggregating in the same node data that are likely to occur together in query results. However, we assume that the queries are not known upfront and we have to rely only on the information contained in the conceptual model.

The final goal of the methodology is to produce a *template* for the final graph database. Basically, there are always similar nodes in a graph database, that is, nodes that share many attributes and are connected by the same kind of edges. We can say that homogeneous nodes identify a "data type". A *template* describes the data types occurring in a graph database and the ways they are connected. It represents a logical schema of the graph database that can be made transparent to the designer and to the user of the target database. However, the database instance is not forced to conform the template in a rigid way (i.e. graph databases are schema-less). It is rather the initial structure of the graph database that can be extended or refined later. More details about this methodology can be found in [9].

Migration of Relational to Graph Databases. As most data sources are stored in relational databases, it is difficult to migrate them to graph databases. We propose a methodology to facilitate the transition from structured to graph databases that are modeled with property graphs and persisted with Graph Database Management System (GDBMS). It is a system-independent methodology to transform a relational database into a materialized graph database and to convert relational conjunctive queries into path traversal queries. As for the modeling methodology, it uses constraints defined over the relational database to minimize the number of data accesses required by graph queries. Existing GDBMSs provide ad-hoc importers based on a naive approach that generates a node for each tuple occurring in the source database and an edge for each pair of *joinable* tuples, that is, tuples satisfying a foreign key.

Conversely, in our approach we aggregate values of different tuples in the same node to speed-up traversal operations over the target. The basic idea is to store in the same node data values that are likely to be retrieved together in the evaluation of queries. Intuitively, these values are those that belong to joinable tuple. However, by just aggregating together joinable tuples we could run the risk to invalidate the property graph constraints. Therefore, we consider a data aggregation strategy based on a more restrictive property, which we call *unifiability* [8]. This notion guarantees a balanced distribution of data among the nodes of the target graph database and an efficient evaluation of queries over the target that correspond to joins over the source. The query translation technique first generates a graph-based structure, called *query pattern*, that denotes the sub-graphs of the target graph database including the result of the query. A query template is then translated into a path traversal query.

Compression of Graph Databases. As the size of the graph increases, so too does the challenge of querying it at high performance. Unfortunately, distributed and parallel databases are not always a scalable solution for graphs because of the low locality in the data matching the query. One solution to improving the performance of particular types of graph operations is to reduce the size of the original graph by turning it into a smaller graph. 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. We noticed that 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* [14], 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 devised *sparsification* techniques 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. 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".

There are several choices for how to compute the sparsification over the entire graph. To this end we devise two different compression strategies: a *greedy* sparsification that aims at producing the highest performing query execution plans over the compressed graph and a *space-aware* sparsification that guarantees better compression rates. These techniques can be implemented as a layer above existing graph database systems, so that the end-user can benefit from this technique without requiring modifications to the core graph database engine code. It enables the database engine to evaluate graph pattern matching queries without ever having to decompress the data. For space constraint, we leave the details about the compression strategies and the query answering algorithms to a forthcoming paper [18].

3. FLEQSY: THE UNIFIED FRAMEWORK

Logical Architecture. The high-level architecture of FleQSy is in Figure 2. Many different kind of flexible queries Q can be submitted to FleQSy through the User Interface. Then, the Query Analyser extracts from Q the information needed by the Query Engine for computing the top-k answers $a_1, a_2, \ldots, a_k$ to output. FleQSy indicates to the user the relevance of an answer with respect to the query. The relevance of the answers is given by a *scoring function* that can take into account both the query Q and the content of the database G. However, FleQSy is independent of a specific scoring function, and therefore different functions can be used for different problems.

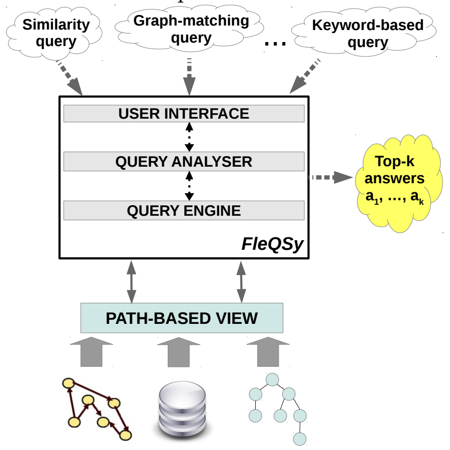


Figure 2: FleQSy: The Unified Framework

The framework is capable of managing several types of data sources (in this thesis we tackle problems over relational and RDF databases) by modeling them in a uniform way through graphs. We access the sources taking advantage of a Path-Based View of the graph, where *paths* are the basic information unit of concern. The Path-Based View is sometimes implemented in a virtual way, where the data paths are only computed at run time; other times FleQSy relies on path-based indexes to facilitate the processing [3].

Meta-approach for Query Answering. FleQSy follows a meta-approach composed of three main phases, that is, every problem is solved by instantiating an algorithm for each of the phases. The three phases are explained as follows.

Pre-processing: the query Q is analysed by the Query Analyzer to individuate the criteria that the final answers should conform to. Such criteria are constraints that answers have to satisfy. They regard both the content and the structure of the answers. Then, we use the path-based view to search for the data paths in G that match the criteria. At the end of these lookups, we have all the paths P of G that are "relevant" for Q. Let us suppose, for instance, a keyword-based query where the constraint is the inclusions of the input keywords in the answers. Therefore, during the pre-processing, we find the paths matching (e.g., containing) such keywords.

Clustering: we group together the paths of P that are similar with respect to the criteria individuated during the pre-processing. Each group is also called a *cluster*. FleQSy assesses the relevance of the paths in P by using the same scoring function used for the answers. The paths are ordered inside the clusters according to the scoring function.

Building: we generate the final answers by combining, at each run, the most relevant paths in every cluster. The algorithms used for the building decides how to combine the paths. In addition, we also aim at satisfying some properties as explained in the following.

Correctness and completeness of the process are less meaningful in the context of flexible query answering. We will analyse the accuracy of the approach in terms of *precision* and *recall*. Intuitively, we are able to find relevant answers to the query since the clusters are built on top of the criteria individuated by the pre-processing and every cluster tries to contribute to the generation of the current answer.

Properties of FleQSy. In pursuing the definition of the framework, we identify different challenges that represent goals for the development of FleQSy.

Monotonicity: a ranking is monotonic if a_i is more relevant than a_{i+1}. Consequently, a query answering process is monotonic if it generates the answers following a monotonic ranking. From a practical point of view, this means to return the best (top-k) answers in the first generated instead of enduring to process blocks of n candidates, with $n > k$, out of which they select the best k in a second time. We believe that monotonicity is a relevant feature for flexible query answering because, since relaxations are less selective on the database, flexible query answering intrinsically tends to generate (much) more answers than required (i.e. the exact and the relaxed/approximated answers). Differently than the state-of-the-art, where searching and ranking are computed separately, FleQSy combines the generation with a relevance assessment of partial answers. It follows that, if query answering is monotonic, the ranking task is needless and the time-to-result is improved. For example, asynchronous applications such as data visualization apps can start the rendering process before the query answering has terminated. To guarantee the monotonicity preserving the correctness, we rely on theoretical results [7, 6] inspired by the Threshold Algorithm [10].

Scalability: this property concerns the ability of the algorithms to handle an ever-larger size of the database without increasing computational time according to the increase of size. More specifically, since we deal with flexible query answering problems that are mostly NP-hard, our goal is to relax those problems in order to reach a polynomial (possibly linear) time complexity of the algorithms with respect to the size of the database. At the same time, we do not want to decrease practical effectiveness of the process.

Distributed Implementation: this property is a specification of Scalability. More precisely, it is intended as the possibility to scale a problem through the execution of parallel algorithms over data sources that are located on different computer machines.

FleQSy in use: approximate graph matching. As an example, we show how the *approximate graph matching* over RDF data is solved within FleQSy. This problem became relevant with the advent of linked open data initiatives, where organizations are opening up their data using RDF. In this context, a user should know the OWL language to understand the organization of the data. Moreover, data do not always conform strictly to the ontology of reference, so that users write their queries by trial and error in order to avoid empty results. Instead, with FleQSy we get approximate answers, that is to retrieve the best sub-graphs of the RDF dataset that best match (not necessarily in an exact way) the input query. This is not intended as an extension of SPARQL, but just a relaxation of the graph pattern matching problem over RDF, that of course, can be expressed through a SPARQL syntax.

In literature, the problem is solved by relaxing the graph isomorphism problem with heuristics or indexing sub-components of the dataset (e.g., sub-graphs, sub-trees, paths). These methods are able to reach a polynomial time complexity but they are still not practicable for systems exposed on the Web.

In FleQSy, the *pre-processing* step decomposes Q in query paths and retrieves the paths P based on a matching between the final constant node (if any) of the query paths and the last node of the data paths. The paths of P retrieved through the same query path are grouped together (*clustering* step). Again, the paths within a cluster are ordered according to their relevance. Note that the same path can be inserted in different clusters, possibly with a different relevance. In the *building* step, we combine the paths coming from different clusters (i.e. picking the most relevant ones) and we check if their intersections are compliant to the ones in Q. If yes, the combination is an answer for Q.

4. RELATED WORK

Graph Data Management. To the best of our knowledge, there is no work that tackles specifically the problems of modeling graph databases and migrating queries from a relational to a graph database management system. Concerning the modeling, developers mainly rely on best practices and guidelines based on typical design patterns, published by practitioners in blogs or only suited for specific systems.

For importing data from a relational database, existing GDBMSs are usually equipped with facilities that rely on naive techniques in which, basically, each tuple is mapped to a node and foreign keys are mapped to edges. This approach however does not fully exploit the capabilities of GDBMSs to represent graph-shaped information. Moreover, there is no support to query translation in these systems.

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 into a smaller graph [11]. While most of the these works propose *lossy* compression of the graph, Fan et. al. [11] propose a lossless compression and a query pattern matching answering that does not need to decompress the data. The compression uses bisimulation equivalence to merge into the same hyper-node many original nodes, all sharing type and connections. In this case the queries have to be expressed via bounded simulation rather than graph isomorphism. These kind of queries work well on in-memory databases but differ from queries normally employed by disk-based graph database systems. Finally, there are other in-memory graph compressions that are adopted for graph analysis [16].

Flexible Query Answering. FleQSy is inspired by frameworks in the context of data modeling and data analysis. Apache MetaModel[1] is a framework for the modeling of heterogeneous data sources. MetaModel relies on a unified view of the underlying sources, implemented by connectors and API queries for CRUD and SQL-like operations. UnQL[2] addresses the problem of data heterogeneity through a standardized query language for SQL and NoSQL database systems. MetaModel and UnQL try to avoid duplications due to the query rewriting mechanism of the connectors, but unfortunately, they do not provide features for

[1] http://metamodel.eobjects.org/

[2] http://unql.sqlite.org/

flexible query answering. The framework MapReduce offers a high-level abstraction for solving problems of batch data analysis through rounds of two independent and subsequent functions, i.e. map and reduce. PowerGraph [13] is a framework for graph data analysis. It proposes the GAS paradigm that allows to specify the problems through the implementation of three functions, namely Gather, Apply, and Scatter. In a similar way to MapReduce and GAS paradigm, FleQSy provides a high-level abstraction for flexible query answering through one round of the *pre-processing*, *clustering* and *building* functions.

To the best of our knowledge there is no approach integrating flexible query evaluation over heterogeneous data, but existing works focus on single problems separately. In the context of relaxed query answering, there are works that propose to search over different databases by keywords [21, 20, 15]. Kite [21] answers keyword-based search queries across multiple relational databases. Qin et al. [20] integrate different solution semantics to structured keyword search within the same framework. EASE [15] computes keyword search over indexed and graph-modeled semi-structured, structured and unstructured data source. Unfortunately, these approaches heavily suffer from performance drawbacks [1]. We solved the problem of structured keyword search in FleQSy inspired by a recent trend that proposes to solve the problem beyond the traditional dichotomy of *schema-based* and *schema-free* approaches [1].

For what concerns the monotonicity, there exist related work focused on the searching of top-k answers [19, 12]. Klee [19] addresses top-k algorithms on a distributed environment of peers. It proposes variations of the Threshold Algorithm (TA) [10], where however each answer is computed on the same peer. The authors in [12] proposes a unified solution for different top-k computations, where a set of scoring metrics are given. Our monotonic query answering differs from TA because we can use non-aggregative and non-monotonic relevance metrics and, unlike [12], we use one metric at a time.

5. RESULTS

Graph Data Management. We implemented the tool R2G for migrating the relational databases using the Blueprints library[3] , which is a collection of interfaces to databases modeled through property graphs. In this way, we could compare the performances of our strategy against the naive strategy proposed by the most common GDBMSs. We used the benchmark in [4] that consists on two datasets (i.e. IMDb and Wikipedia) and 50 keyword search queries for each dataset that we transformed in SQL with a structured keyword search tool. For each dataset, we ran the 50 queries ten times and measured the average response time. We performed cold-cache experiments and warm-cache experiments. Figure 3 shows the performance for cold-cache experiments. In the figure, for each GDBMS we consider the time to perform a query on the graph database generated by using the sparse strategy (i.e. black bar) and the time to perform the same query on the graph database generated by using our strategy (i.e. white bar). Our methodology allows each system to perform consistently better. Competitors' strategy spend much time traversing a larger number of edges.

[3] https://github.com/tinkerpop/blueprints/wiki

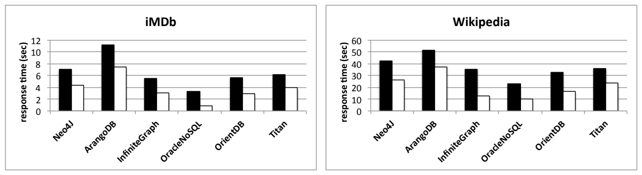


Figure 3: Path traversal: black bars refer to naive strategy and white bars refer to our strategy

In order to test the compression layer, we developed a graph database system prototype on PostgreSQL 9.1. The design of our prototype is based on popular *triple stores* that persist the graph in a single clustered table with three columns s, p, o and exhaustively index this table. We have used two directed graphs: the first is the Twitter dataset (81,306 nodes and 1,115,532 edges) available among the SNAP datasets[4] ; the second is a synthetic dataset, that we call Barabasi (161,306 nodes and 4,800,000 edges). We have created Barabasi using a generator included in JUNG[5] (Java Universal Network/Graph Framework) for random Barabasi power-law graphs. This dataset nicely approximates directed real-world graphs, such as the Web graph [2].

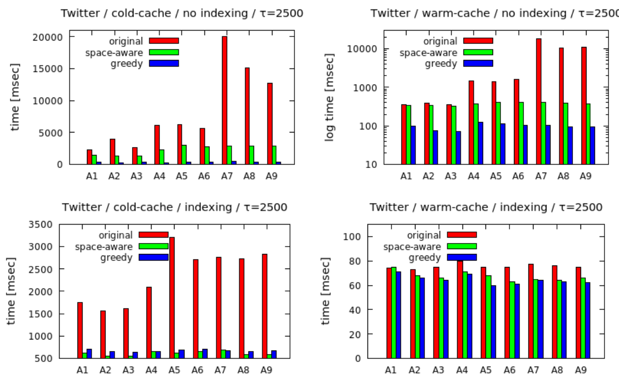


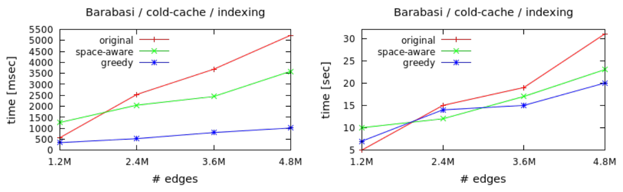
Figure 4: Results of Compression.

We sparsified, using both strategies (i.e. greedy and space-aware) introduced in Section 2, each dataset. Furthermore, for each of these two strategies, we compressed the dataset multiple times, using different threshold (τ) values. Figure 4 shows the comparison between the traditional non-compressed approach (i.e. original) and our (using graph compressed with $\tau = 2500$) with queries (star queries) of different size including only high-degree nodes over Twitter. It shows both cold-cache and warm-cache runs, and two different indexing schemes on the database: indexing where every permutation of triples is indexed and no indexing where no indexing scheme is applied on the triple store.

Although overall performance is faster when indexes can be used to accelerate matches to query constants, the relative performance of the queries are nearly identical for this set of experiments as for the set of experiments without indexing. This is because compression and indexing are complementary. Indexes help to accelerate the match of the high-degree constants in the query, and can do so for both compressed graphs and uncompressed graphs. This is because our compression algorithms change the structure of

[4] http://snap.stanford.edu/data/
[5] http://jung.sourceforge.net/

the graph, but do not change the fundamental representation of vertexes and edges. This allows the compressed graph to be indexed in the same way as the original graph. However, the compressed strategies are able to maintain their advantage over the uncompressed strategy when indexes are used because indexes only help for first steps of query processing, and the main advantage of the compressed strategies is that they keep the intermediate result set and join-input sizes small.



(a) High-degree nodes (HD) (b) HD and Variables

Figure 5: Compression with evolving graph.

We now present the results of experiments we ran in order to understand how the different pattern-matching techniques scale when the size of the graph increases. As the Barabasi model simulates the evolution of a real-world network [2], we measure the performance of pattern-matching queries over BARABASI at four points during its evolution: (1) 41,305 nodes and 1,200,000 edges, (2) 81,306 nodes and 2,400,000 edges, (3) 121,306 nodes and 3,600,000 edges, and (4) 161,306 nodes and 4,800,000 edges. Due to the preferential attachment formation of the graph [2], the most expensive queries involve the same high-degree nodes in all the four datasets, which allows us to compare exactly the same queries over the evolving network. However, as the size of graph changes, the definition of "high degree" changes along with it. For the first intermediate point in the Barabasi graph, we used $\tau = 3500$; for the second: $\tau = 7500$; for the third: $\tau = 10000$; for the fourth: $\tau = 12500$. Figure 5 presents the results of our experiment obtained using the average response time of different (i.e. 9) queries for each of the diagrams. The behaviour of queries with only high-degree vertexes is linear with respect to the growth of the size of the dataset. However, the slopes of the compressed graphs are smaller, with the greedily compressed graphs the smallest. Note that when the graph is small, the space-aware strategy performs worse than the non-compressed strategy. This is because the main advantage of the space-aware strategy is compression ratio, but the effect of compression on performance is minimal for small graphs, and not worth the extra joins that the space-aware strategy introduces. For queries with both high-degree vertexes and variables, the performance of all three strategies is similar for smaller graphs, but as the size of the graph increases, the advantages of compression yield large improvements in performance.

Flexible Query Answering. We have instantiated FLEQSY on different problems. Table 1 summarizes the results achieved (so far) with respect to theoretical and experimental results. For more details on each single problem, we suggest to refer the specific papers [6, 9, 7].

6. CONCLUSION AND FUTURE WORK

In this paper we presented FLEQSY, a novel meta-approach to solve several flexible query answering problems on structured and semi-structured data. We also address few issues in the context of graph data management: modeling, migration from relational databases and compression.

Problem	Theoret. Time Complex.	Experim. Results	Monoto--nicity	Distrib. Implem.
KS over relational DBs	$O(\|P\|)$	$O(\|P\|)$	●	○
KS over RDF DBs	$O(\|P\|)$	$O(\|P\|)$	●	●
Approximate Matching over RDF DBs	$O(\|P\|^2)$	$O(\|P\|^2)$	●	○

● Fulfilled ○ Not fulfilled yet P = measure of database size

Table 1: Summary of results obtained with FleQSy.

We have several directions of future work. On FLEQSY, we are working on the aspects of distributability and we are exploring a querying process where the answers are formed with portions coming from different data sources. We are currently investigating if graph compression is beneficial to graph processing systems and to other types of query answering (e.g., reachability queries).

7. REFERENCES

[1] A. Baid, I. Rae, J. Li, A. Doan, and J. F. Naughton. Toward scalable keyword search over relational data. *PVLDB*, 2010.
[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
[3] P. Cappellari, R. De Virgilio, A. Maccioni, and M. Roantree. A path-oriented rdf index for keyword search query processing. In *DEXA*, pages 366–380, 2011.
[4] J. Coffman and A. C. Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Trans. Knowl. Data Eng.*, 26(1):30–42, 2014.
[5] R. De Virgilio and A. Maccioni. Distributed keyword search over rdf via mapreduce. In *ESWC*, pages 208–223, 2014.
[6] R. De Virgilio, A. Maccioni, and P. Cappellari. A linear and monotonic strategy to keyword search over rdf data. In *ICWE*, pages 338–353, 2013.
[7] R. De Virgilio, A. Maccioni, and R. Torlone. Approximate querying of rdf graphs via path alignment. *Distributed and Parallel Databases*, pages 1–27, 2014.
[8] R. De Virgilio, A. Maccioni, and R. Torlone. Graph-driven exploration of relational databases for efficient keyword search. In *GraphQ*, pages 208–215, 2014.
[9] R. De Virgilio, A. Maccioni, and R. Torlone. Model-driven design of graph databases. In *ER*, pages 172–185, 2014.
[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.
[11] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.
[12] S. Ge, L. Hou U, N. Mamoulis, and D. W. Cheung. Efficient all top-k computation - A unified solution for all top-k, reverse top-k and top-m influential queries. *IEEE Trans. Knowl. Data Eng.*, 25(5):1015–1027, 2013.
[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
[14] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graph evolution: densification and shrinking diameters. *TKDD*, 2007.
[15] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.
[16] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng.*, 26(12):3077–3089, 2014.
[17] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.
[18] A. Maccioni and D. J. Abadi. Scalable pattern matching over compressed graphs via sparsification. 2015 - **[submitted]**.
[19] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, 2005.
[20] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD*, pages 681–694, 2009.
[21] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, pages 346–355, 2007.