PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism

Raghavan Raman
Oracle Labs
raghavan.raman@oracle.com

Oskar van Rest
Oracle Labs
oskar.van.rest@oracle.com

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Zhe Wu
Oracle
alan.wu@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

Jay Banerjee
Oracle
jayanta.banerjee@oracle.com

ABSTRACT

Subgraph isomorphism, or finding matching patterns in a graph, is a classic graph problem that has many practical use cases. There are even commercialized solutions for this problem such as RDF databases with their support for SPARQL queries. In this paper, we present an efficient, parallel *in-memory* solution to this problem. Our solution exploits efficient data representations as well as algorithmic extensions, both tailored for parallel, in-memory processing. Moreover, when processing RDF data, we reduce the problem size by converting certain nodes and edges into properties. We also propose a new graph query language where such a conversion can be encoded. Our evaluation shows that our solution can achieve significant performance boost over an existing secondary storage based RDF database.

1. INTRODUCTION

With the recent growing interest on graph databases, it has become more important to have a fast solution to the classic problem of subgraph isomorphism, or finding matching patterns in a large graph. Theoretically, the problem of subgraph isomorphism involves finding all subgraphs of graph D that are isomorphic to another graph Q. The graph D (the data graph) is typically huge while the graph Q (the query graph) is typically small. The term isomorphic in the above definition says that there is a one-to-one mapping between the nodes and edges of the two graphs. Note that when the nodes (edges) of the graph Q have certain properties, the matching nodes (edges) in graph D must have the same-valued properties. Although the general problem of subgraph isomorphism is NP-Hard, the actual instances of this problem are much more tractable in practical contexts, because nodes and edges are associated with unique identifiers and/or a set of properties in popular graph data models.

GRADES'14, June 22 - 27 2014, Snowbird, UT, USA

http://dx.doi.org/10.1145/2621934.2621939

Consequently, many solutions have been proposed for this problem, including commercialized ones. First, there are graph databases [9, 3] which adopt the classic RDF graph data model [16] and SPARQL, a query language for RDF data [17]; when a SPARQL query is submitted, the RDF database finds all matching subgraphs and graph elements of the query, by solving the subgraph isomorphism problem. Similar pattern-matching operations are also supported by graph databases [10] that adopt the more recent Property Graph (PG) data model [1]. Noticeably, these graph databases are all disk-oriented so that they can process even large graphs. Second, there are also in-memory solutions for this problem [5, 8, 12, 11]. However, most of them are single-threaded and/or do not seem to handle large-sized graphs very well on SMP systems.

In this work, we describe a new parallel and efficient in-memory solution to the subgraph isomorphism problem. Unlike existing in-memory solutions [5, 12, 11], our system is designed to handle very large graphs (i.e. billions of edges). Specifically, we exploit efficient in-memory data representations both for graph and partial solutions. Our solution captures the inherent large degree of parallelism of the problem and exploits contemporary big multi-processor machines. We also modified the conventional backtracking algorithm [8], for the sake of fast, parallel, in-memory processing.

Our solution is applicable to both RDF model and PG model. When our solution is applied to RDF model, however, we convert some of RDF nodes and edges into properties, and thereby reducing the problem size. We introduce a new graph matching query language, GMQL, which can be used to query both RDF graphs and PG graphs. Our query processor has preliminary support for automated conversion from SPARQL to GMQL.

We evaluate our solution with LUBM [4], a standard benchmark for RDF and SPARQL. When compared to an existing disk-oriented RDF database, our solution achieves orders of magnitude performance improvement. The experiments also show that our solution scales well on large SMP systems.

The main contributions of this paper are as follows:

- A parallel and efficient in-memory solution to the problem of subgraph isomorphism that can be used for RDF data model as well as PG data model.

- Evaluation of our solution on a standard RDF benchmark, LUBM, including a comparison with an existing RDF database.
- The design of GMQL, an intuitive high-level graph query language for PG model, and its conversion from SPARQL.

The rest of the paper is organized as follows. Section 2 describes the background and the framework we use to build our solution and also discusses related work. Section 3 presents our in-memory solution to subgraph isomorphism. Section 4 evaluates our solution on LUBM benchmark. Section 5 presents our query language, GMQL, and Section 6 concludes.

2. BACKGROUND

We now describe the framework we use for our parallel in-memory solution to the problem of subgraph isomorphism.

Parallel Graph analytiX (PGX) is a framework that supports in-memory graph analysis. It includes parallel and efficient implementations of various graph algorithms. Some of these algorithms are part of the runtime of the Green-Marl Domain Specific Language [6]. Additionally, PGX also includes libraries to efficiently execute graph algorithms. For example, PGX includes an efficient implementation of the priority heap data structure which can be used by the Green-Marl compiler to implement a heap used in an algorithm like Dijkstra's shortest path.

PGX contains abstractions for nodes, edges, and properties in graphs. The nodes and edges of a graph are stored in the Compressed Sparse Row (CSR) format, which consists of two contiguous arrays. For every node in the graph, the first array stores the starting index to the list of edges in the second array. The second array stores the ids of the destination nodes of the outgoing edges for every node in the graph ordered by the nodes in the first array. The properties on nodes and edges are stored as separate arrays, one per property. PGX supports a variety of data types to be used as properties, including, primitive types, string, string-set, and datetime.

PGX.ISO is the graph pattern matching component in the PGX framework. Our solution to the problem of subgraph isomorphism is implemented as part of PGX.ISO.

2.1 Related Work

We now discuss some existing work in the area of subgraph isomorphism. One of the earliest solution to the problem of subgraph isomorphism was provided by Ullman [14]. The solution is based on a backtracking search algorithm that also prunes candidate nodes when their degree is less than the degree of the node being matched. A number of algorithms have been proposed to improve the Ullman algorithm. These mostly involve a better mechanism to prune candidate nodes as early as possible, identifying a good order of query nodes for matching, etc.

VF2 [11] is an algorithm for subgraph isomorphism that introduces new pruning rules to improve the performance of the Ullman algorithm. While two of these rules check the consistency of the current partial solution, there are three other rules that prune the search tree by looking ahead up to 2 steps in the search. One of these pruning rules is to eliminate candidate nodes that are not connected to the already matched nodes in the data graph.

QuickSI [12] identifies a good sequence of query graph nodes to match by using the node frequency information that is pre-computed from the data graph. The algorithm also proposes a technique to verify the consistency of partial solutions by building a feature-based index. A comparison of some of these techniques that improve the Ullman algorithm can be found in [8].

Turbo ISO [5] is one of the most recent work on subgraph isomorphism that shows the importance of using different matching orders for different regions of the data graph. They illustrate a novel method to identify the matching order for a given region in the data graph by performing a depth-first search from the starting node in the region.

Finally, Sun et al. [13] present an approach to subgraph matching for huge graphs with billion nodes. They focus on a distributed setting, whereas our focus in this work is on a shared-memory system.

3. IN-MEMORY SOLUTION

Existing solutions to the problem of subgraph isomorphism use a variant of the basic backtracking algorithm along with some filtering techniques to prune partial solutions as early as possible during matching. A generic version of the backtracking algorithm used by most of the solutions to subgraph isomorphism can be found in [8]. These earlier approaches report performance by searching for the first k embeddings (k is mostly set to 1000) of the query graph in the data graph. On the other hand, in the case of SPARQL queries, we need to find all possible embeddings of the query graph in the data graph.

As a first step to our solution, we implemented the backtracking subgraph search algorithm along with pruning and evaluated it on the LUBM benchmark. Our observations from that experiment are listed below:

- While it is quite easy to parallelize the backtracking subgraph search algorithm, load balancing across many threads becomes very difficult in such an implementation. This affects the scalability of the solution on large SMPs.
- Pruning partial solutions is not always useful. If the pruning does not remove any partial solutions, as in some LUBM benchmarks, then pruning is just doing extra work without any gains.
- Ordering of nodes for matching is very important to obtain the best possible performance.
- Data structure to store partial solutions has a significant impact on the performance.

The above factors play a significant role in achieving good performance, especially in cases where the data graph is large. While the issue of selecting the right order of nodes has been addressed in the literature [5], the other issues like parallelization and data structure to store partial solutions have not been addressed so far. We focus on building a solution to subgraph isomorphism that overcomes the above problems.

We now present our parallel in-memory solution to subgraph isomorphism. First, we discuss the ordering of query nodes in our solution, followed by the strategy we use for matching. Then, we present our parallelization methodology and the data structure we use to represent partial solu-

tions. Finally, we describe the edge-first matching approach and its benefits.

3.1 Ordering of Query Nodes

The order in which query nodes are matched affects the matching performance to a significant extent. The state of the art solution to subgraph isomorphism [5] shows that different regions of the data graph require different ordering of query nodes.

In our solution, we use a fixed order of query nodes for matching. Also, our orderings are chosen such that, at every step (except the first node), we pick a node that has at least one already matched neighbor. This strategy helps in reducing the search space as explained in Section 3.2. We plan to explore dynamic ordering and automatically choosing the right order in future. Our evaluation in Section 4 compares the performance of different orders for the LUBM query 2, which clearly shows that a fixed order is quite good for the LUBM benchmark.

3.2 Matching Strategy

Every query node that we match has at least one neighbor that has already been matched (except for the first node). Let us first consider the case where the node-to-be-matched has exactly one already matched neighbor. Matching such a node involves iterating over all the neighbors of the matches of its neighbor that has already been matched. For example, matching a node B whose neighbor A has already been matched involves iterating over the neighbors of the matches of A and looking for matches with B. This strategy reduces the search space for B because matching B requires searching only the neighbors of the matches of A as against searching the entire data graph in case none of B's neighbors have been matched.

Let us now consider the case where the node-to-be-matched has more than one already matched neighbor. In this case, matching this node involves finding the nodes that are common among the neighbors of the matches of all its neighbors that have already been matched. The node-to-be-matched in this case is referred to as a *join* node. For example, matching a node C whose neighbors A and B have already been matched involves finding the nodes that are common among the neighbors of the matches of A and B. PGX includes a fast and efficient method to find such common neighbors. We use this common neighbor implementation in PGX to match join nodes. The details of the common neighbor implementation in PGX are beyond the scope of this paper.

Note that a "neighbor" in this case could mean either an outgoing neighbor or an incoming neighbor depending on the direction of the corresponding query graph edge.

3.3 Parallelization

The backtracking search algorithm for subgraph isomorphism is a depth-first approach since it explores a partial solution until completion before moving on to the next partial solution. The load imbalance in the parallel version of this algorithm is primarily due to the differences in the amount of work in different parts of the search tree.

Our solution solves this problem by performing the matching in a breadth-first manner. First, we find all the nodes in the data graph that match the first node in the query graph. Then, we match the second node in the query graph

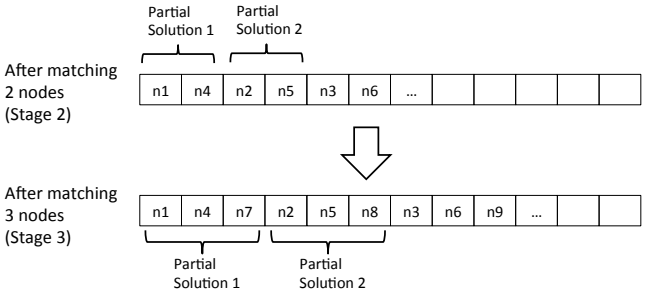

Figure 1: Partial Solution Representation

for all the partial solutions obtained in the first stage and so on. We continue matching one query node at every stage for all the partial solutions obtained in the previous stage, in a breadth-first manner.

We parallelize the breadth-first matching as follows: In the first stage, all the nodes in the data graph are distributed uniformly among all the threads. In every subsequent stage, all the partial solutions obtained in the previous stage are distributed uniformly among all the threads. At every stage, the threads store the partial solutions they generate in a thread-local storage. At the end of every stage, after all the threads complete their matching, we copy the thread-local partial solutions into a single global storage. This copy is done in parallel by all the threads after performing a prefix-sum operation to compute their positions in the global storage. Since every thread begins with the same number of partial solutions at every stage in matching, there is a better balance in the amount of work among all the threads. Note that there may still be some load imbalance among the threads. But, in practice, we noticed that this approach results in an efficient parallel solution that scales well on large SMPs.

3.4 Partial Solution Representation

Most existing subgraph matching algorithms store a pair of nodes (u, v), for every query node u, in their partial solutions, where v is the data node that matches u. The pair is necessary because, in general, the algorithm can match the query nodes in any order. Since we fix the matching order of query nodes in our solution, we omit the query nodes and store only the data nodes in the partial solution. The position of the data node in the partial solution specifies the query node that it matches to. This representation reduces the memory usage of partial solutions by half and is very important for our solution since we expect to have millions of partial solutions for some queries.

We noticed that storing every partial solution as a separate data structure, say an array, resulted in the creation of a large number of such very small data structures due to a very high number of partial solutions, thereby resulting in poor performance. In order to avoid this problem, we store all partial solutions "inlined" in one big array. This array is compact, i.e., it reserves space to store only the nodes that have been matched so far. For example, if there are k partial solutions at the end of the second stage, the array is just big enough to store $k * 2$ elements. The inlined representation in a single array also provides more spatial locality while iterating over the partial solutions.

Figure 1 shows an example of how our partial solution representation changes from stage 2 (after matching 2 query nodes) to stage 3 (after matching 3 query nodes). After

Table 1: Statistics of graphs generated from LUBM

	LUBM 8K	LUBM 25K
# Nodes	173 million	542 million
# Edges	658 million	2.05 billion

stage 2, every group of two nodes in the array constitutes a partial solution and after stage 3, every group of three nodes in the array constitutes a partial solution.

3.5 Edge-First Matching

At every stage in our solution (starting from the second stage), the matching step involves looking for a node that is connected to given data node x and matches a given query node q. This involves iterating over all the edges from x and the corresponding neighbors of x looking for matches with the corresponding edge and neighbor in the query graph. This matching can be done in two different ways:

- First match the neighbor of x with the query node and then match the edge from x with the query edge. This is called the *node-first* approach.
- First match the edge from x with the query edge and then match the neighbor of x with the query node. This is called the *edge-first* approach.

Recall from Section 2 the CSR representation used in PGX to store nodes and edges. In such a representation, iterating over the edges from a given node accesses consecutive elements of the arrays corresponding to edge properties, whereas iterating over the neighbors of a given node will result in random accesses to the arrays corresponding to node properties. With the edge-first approach, since we can avoid matching neighbors in case the edge matching fails, there will be lesser random accesses as compared to the node-first approach.

4. PERFORMANCE EVALUATION

We evaluate our solution on a x86 system and a SPARC system. The x86 system we use is a 2x8 core Intel Xeon E5-2660 machine with 256 GB memory where each core is 2-way hyper-threaded running at 2.2GHz. The machine has 4 250 GB SSDs and runs 64-bit Linux version 2.6.32. The SPARC system we use is a SPARC T5-8 server (8x16-cores) with 4 TB memory where each core is 8-way multithreaded running at 3.6 GHz. The server has 6 1 TB SSDs and runs Solaris 11.

We use the LUBM datasets 8K and 25K to evaluate our solution. The number of nodes and edges in our graphs generated from these datasets are shown in Table 1. Note that the number of triples in these datasets are higher than the number of nodes and edges. This is because we convert some of these triples into properties on nodes. Specifically, we convert triples with "rdf:type" as predicate into a *node-type* property. Also, other triples like those corresponding to *name*, *title*, *email*, *telephone*, etc., are converted to node properties. While evaluating an LUBM query in PGX.ISO, we load only those properties that are required for the particular query from the Oracle database.

First, we compare the performance of our solution, PGX.ISO, with Oracle's SQL based solution for SPARQL on LUBM 8K dataset for all LUBM queries. Table 2 shows the execution times of SQL (using parallel execution of SQL statements) and PGX.ISO on LUBM 8K for all LUBM queries running on our x86 system. Note that the least resolution of time measured in SQL is 0.01 seconds. Hence, any time less than 0.01 seconds is shown as 0 in Table 2. The Oracle SQL queries were tuned and run in parallel to achieve the best possible performance. These execution times are for queries that count the number of solutions for both SQL and PGX.ISO.

Table 2: Performance comparison between Oracle-SQL and PGX.ISO on LUBM 8K running on x86

LUBM Query	# Solutions	Execution Time (s)	
		SQL	PGX.ISO
Query 1	4	0	0
Query 2	2528	21.26	0.1
Query 3	6	0	0
Query 4	34	0	0
Query 5	719	0.02	0
Query 6	83557706	23.56	0.14
Query 7	67	0.01	0
Query 8	7790	0.23	0
Query 9	2178420	58	0.58
Query 10	4	0	0
Query 11	224	0.01	0
Query 12	15	0.14	0
Query 13	37118	1.15	0.03
Query 14	63400587	21.09	0.1

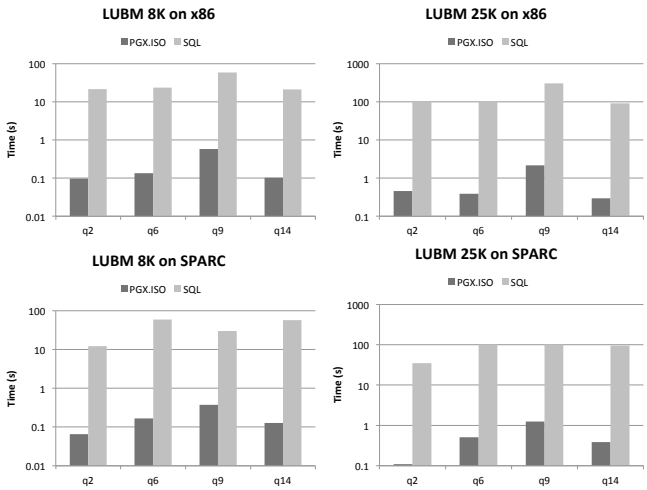

Figure 2: Comparing PGX.ISO and Oracle-SQL

The execution times of SQL in Table 2 show that the LUBM queries can be divided into two groups. The first group is the four queries, 2, 6, 9, and 14, that take a reasonable amount of time in SQL. For these queries, PGX.ISO is more than 100x faster than SQL. The second group is all the remaining queries that take 1 second or less to complete in SQL. These queries have a common feature that one of the nodes in the query graph has a unique "id" and hence it matches with exactly one node in the data graph. An index on the "id" of the nodes can be used to directly look up the matching node in the data graph. This is the reason the SQL completes very quickly for the queries in the second group. Note that PGX.ISO also uses an index and hence completes very fast for these queries. For this reason, the analysis that follows will be on the queries in the first group.

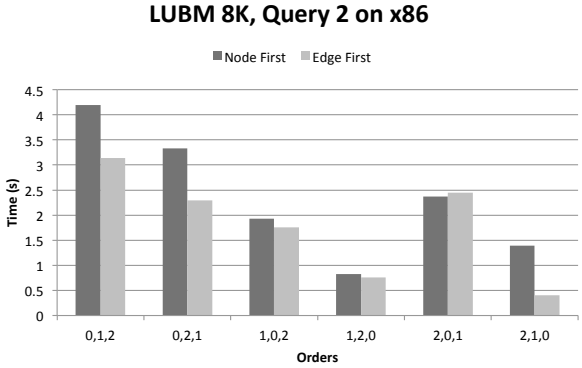

Figure 3: PGX.ISO Performance with node-first and edge-first approaches on different query node orders for LUBM Query 2 on LUBM 8K dataset

Figure 2 shows the comparison of PGX.ISO and SQL execution times on LUBM 8K and 25K datasets running on our x86 and SPARC systems for the four LUBM queries, 2, 6, 9 and 14. Note that the y-axis on those graphs show time in log scale. It is evident from these graphs that PGX.ISO performance is at least a couple of orders of magnitude better than SQL for these four queries on x86 and SPARC.

We now show the effect of different matching orders for nodes in the queries. Figure 3 shows PGX.ISO's execution time for different node orders in query 2 on LUBM 8K dataset running on the x86 system. There are three nodes in query 2 and hence there are six possible orderings as shown in Figure 3. For each order, there are two bars in Figure 3. Let us ignore the different bars within each order for the moment. This graph shows that the performance of subgraph matching varies hugely with the query node matching order. By changing the matching order alone, we could improve the performance by up to 10x according to this graph. In practice, we have observed that choosing the correct matching order could result in bigger performance improvements in some cases.

Figure 3 also shows the performance of PGX.ISO's subgraph matching when using the "node-first" approach and the "edge-first" approach. Each order in Figure 3 includes two bars, the first bar corresponds to when "node-first" matching was performed and the second bar corresponds to when "edge-first" matching was performed. It is evident from this graph that, for all the orders (except 2,0,1), the edge-first approach performs better than the node-first approach. This is because in the node-first approach, matching the nodes (neighbors) first involve accessing non-contiguous locations of an array as explained in Section 3, whereas in the edge-first approach, matching the edges first involve accessing contiguous locations.

We now evaluate the scalability of our solution on x86 and SPARC systems. Figure 4 shows PGX.ISO's scaling for query 2 on LUBM 8K and 25K running on x86 and SPARC systems. Our solution scales linearly until 16 threads on x86 and SPARC for both 8K and 25K datasets. In the case of x86, there is a dip in the scaling from 16 to 32 threads. This is because, with 32 threads, we go beyond the number of cores in the x86 machine and rely on hyper-threading to provide the scaling. On SPARC, there is a dip in scaling from 16 to 128 threads because the threads go out of socket when they are more than 16. The performance difference

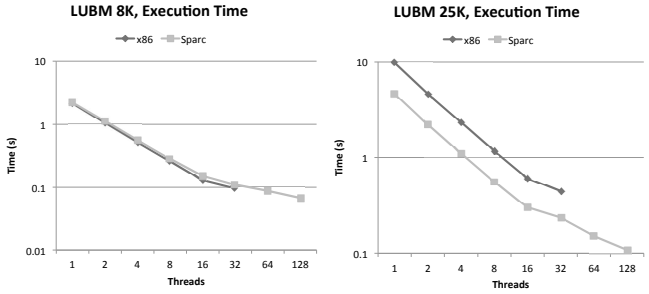

Figure 4: PGX.ISO Scalability on x86 and SPARC

between x86 and SPARC on LUBM 25K is because we were able to use the additional memory available on our SPARC system to build extra indexes to improve performance. The details regarding the indexes are beyond the scope of this paper.

5. GMQL: GRAPH-MATCHING QUERY LANGUAGE

GMQL is a declarative language designed to specify graph pattern-matching queries. It is influenced by SPARQL and Neo4J's Cypher and provides first-class constructs for nodes, edges and properties. While SPARQL is meant for querying RDF models, GMQL, like Cypher, is meant for querying PG models. Like Neo4J, we provide support for conversion from RDF models to PG models, allowing RDF data to be queried using our language as well. However, in contrast to Cypher, we additionally provide preliminary support for automated conversion from SPARQL to GMQL to allow both languages to be used in combination with our parallel and efficient in-memory engine.

5.1 Textual and Graphical Syntax

GMQL provides first-class constructs for nodes, edges and properties and provides meaningful built-in calls for PG graphs. An example of such a built-in call is 'inDegree()', which can be used, for example, to constrain the in-degree of a node (e.g. `X.inDegree()> 3`).

GMQL has a textual and a graphical syntax as shown in Figure 5. This query finds all combinations of graduate students, their universities and the departments from which they obtained their degrees. The textual syntax has an `IN` clause defining the name of the data graph D and a `MATCH` clause defining the subgraph Q in the form of a set of paths consisting of nodes and edges. A graph pattern is formed by using the same node in multiple paths (e.g. nodes `X` and `Y` in the example). Edges may have a type or a variable. In the example, all edges have a type, but a variable is needed in case additional constraints need to be defined on an edge (e.g. `X -[e1]-> Z, e1.type = ub :memberOf, e1.numericProp1 < 12`). The query also has a `SELECT` clause, defining what data should be returned and how it should be formatted. Tabular and JSON format is supported.

Figure 5 furthermore shows GMQL's graphical syntax. Graphically, a graph Q takes the form of an actual graph with constraints placed next to the nodes and edges to which they belong. Cross-constraints that involve multiple nodes and edges (e.g. `X.age < Y.age`) are placed in a separate table to avoid having to visualize them multiple times.

```
SELECT ?X ?Y ?Z
WHERE {
 ?X rdf:type ub:GraduateStudent .
 ?Y rdf:type ub:University .
 ?Z rdf:type ub:Department .
 ?X ub:memberOf ?Z .`
 ?Z ub:subOrganizationOf ?Y .
 ?X ub:undergraduateDegreeFrom ?Y
}
```

```
IN lubm50
MATCH
 X -[ub:memberOf]-> Z -[ub:subOrganizationOf]-> Y,
 X -[ub:undergraduateDegreeFrom]-> Y,
 X.type = ub:GraduateStudent,
 Y.type = ub:University,
 Z.type = ub:Department
SELECT AS TABLE X, Y, Z
```

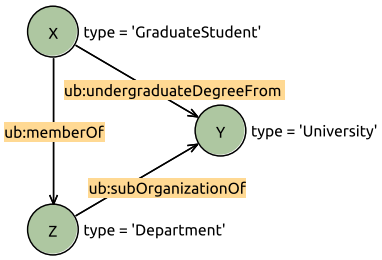

Figure 5: LUBM query 2 in SPARQL (top), textual GMQL (center) and graphical GMQL (bottom)

5.2 Conversion from SPARQL to GMQL

We provide automated conversion from SPARQL to GMQL to be able to efficiently process SPARQL queries using our in-memory and parallel solution. Besides being able to process SPARQL queries efficiently, the conversion also provides a way for existing SPARQL users to easily get started with GMQL by being able to see how their SPARQL queries translate to GMQL. Just like for GMQL, we provide full-featured IDE support for SPARQL. However, although we provide full *parsing* support for SPARQL 1.1 [17], our *conversion* currently only supports a subset of SPARQL. We therefore apply a hybrid approach when processing SPARQL queries: we first try to convert such a query to GMQL and process it using our in-memory and parallel solution, but we fall back to processing SPARQL queries using Oracle's SQL-based solution in case conversion fails.

5.3 Implementation

We provide a (textual) SPARQL editor and textual and graphical GMQL editors that we created using the Spoofax Language Workbench [7]. The editors are capable of synchronizing in real-time [15] by means of transformations defined in Spoofax's transformation language Stratego [2]. Synchronization from SPARQL to GMQL is unidirectional while synchronization between textual and graphical GMQL editors is bidirectional. Queries are translated into C++ programs, which we compile and execute at runtime. Our IDE provides support for presenting query results in graphical and tabular form.

6. CONCLUSIONS

In this paper, we presented a parallel and efficient in-memory solution to the problem of subgraph isomorphism that can be used to implement SPARQL queries on RDF data. This solution is implemented as part of the graph pattern matching component, PGX.ISO, in the graph analysis framework, PGX. We evaluated our solution on the LUBM benchmark, which is a standard benchmark for RDF and showed that our solution handles graphs with billions of edges quite efficiently. We also showed that our in-memory solution performs about two orders of magnitude better than disk based solutions to RDF. In addition, we presented an intuitive query language, GMQL, that can be used to perform graph matching queries at a high level.

7. REFERENCES

[1] Property graph model. `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`. [Online; accessed 5/19/2014].

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008.

[3] O. Erling and I. Mikhailov. In *Conference on Social Semantic Web*, 2007.

[4] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Semantic Web Journal*, 2005.

[5] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. SIGMOD, 2013.

[6] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[7] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, 2010.

[8] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB, 2013.

[9] C. Murray. Oracle spatial and graph - rdf semantic graph developer's guide. `http://docs.oracle.com/cd/E16655_01/appdev.121/e17895.pdf`, 2014. [Online; accessed 5/19/2014].

[10] Neo4j. Neo4j, the world's leading graph database. http://www.neo4j.org/, 2013.

[11] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2004.

[12] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 2008.

[13] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 2012.

[14] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 1976.

[15] O. van Rest, G. Wachsmuth, J. R. H. Steel, J. G. Süß, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *ICMT*, 2013.

[16] W3C. RDF Primer. `http://www.w3.org/TR/rdf-primer/`, 2004. [Online; accessed 5/19/2014].

[17] W3C. SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query/`, Mar. 2013. [Online; accessed 5/19/2014].