Sparqling Kleene - Fast Property Paths in RDF-3X

Andrey Gubichev[*]
Technische Universität München
Germany
gubichev@in.tum.de

Srikanta J. Bedathur[†]
IIIT-D, New Delhi
India
bedathur@iiitd.ac.in

Stephan Seufert
Max Planck Institute for Informatics
Germany
sseufert@mpi-inf.mpg.de

ABSTRACT

As Semantic Web efforts continue to gather steam, the RDF engines are faced with graphs with millions of nodes and billions of edges. While much recent work in addressing the resulting scalability issues in processing queries over these datasets have mainly considered SPARQL 1.0, the next-generation query language recommendations have proposed the addition of regular expression restricted navigation queries into SPARQL. We address the problem of supporting efficient processing of property paths into RDF-3X – a high-performance RDF engine.

In this paper, we restrict our attention to a restricted definition of property paths that is not only tractable but also most commonly used – instead of enumerating all paths that satisfy the given query, we focus on *regular expression based reachability* queries. Based on this, we make the following three major technical contributions: first, we present a detailed account of integrating the recently proposed highly compact reachability index called FERRARI into the RDF-3X engine to support property path evaluation; second, we show how property path queries can be efficiently answered using multiple instances of this index – one instance for each distinct label in the graph; and finally, we develop a set of queries over real-world RDF data that can serve as benchmark set for evaluating the efficiency of property path queries. Our experimental results over Yago2, a large RDF-based knowledge base, show that our proposed approach is highly scalable and flexible.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications

Keywords

Graph-Structured Data, Reachability, RDF

[*]supported by the LDBC EU FP7 project
[†]supported by the DST-MPG Partner group on Large-scale Graph Mining.

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York City, NY, USA.

1. INTRODUCTION

The RDF data format, the common data representation the Semantic Web is a highly expressive graph model that is suited for the "pay-as-you-go" mode of data growth. In this model, labeled nodes are connected through (potentially multiple) labeled, directed edges representing semantic relationships between them. This graph model is queried via the SPARQL query language using graph patterns containing both bounded and unbounded variables. As both the data and the querying models and workloads are quite different from those encountered in the traditional RDBMS world, there is a lot of recent interest in efficient processing of SPARQL queries over large RDF databases.

Despite the flexible graph representation provided by the RDF model, the SPARQL 1.0 recommendation supported only graph pattern matching queries but no navigational queries. For instance, one could not express a query that can be stated in natural language as "find all descendants of a physicist who won a Nobel prize and list the awards they have won". There was no way one could specify *descendants* as a sequence of one or more edges labelled with the hasChild property which would be required for answering the above query. In the recent draft recommendation of SPARQL 1.1, this limitation has been overcome with the addition of *property path* queries, which can be seen simply as regular expressions over paths between two nodes in the RDF graph. The above query pattern can now be expressed in SPARQL 1.1 as:

```
SELECT ?descendant, ?prize WHERE {
  ?x type Physicist.
  ?x hasChild* ?descendant.
  ?x hasWonPrize Nobel_Prize.
  ?descendant hasWonPrize ?prize
}
```

One may observe the use of the familiar Kleene-star operator to specify a path consisting of zero or more edges labeled `hasChild` emanating from the node bound to the variable `?x`, and the terminal node of the path is bound to the variable `?descendant`.

While property paths add graph-navigational power to SPARQL, their efficient evaluation based on their semantics as defined in the early drafts of SPARQL 1.1 recommendation was not possible. It was initially required that all paths that satisfy the specified regular expression be enumerated and for each path the terminal nodes involved have to be reported. Based on this, if we consider a *valid path* that contains a cycle, it may lead to nonterminating results. While the recommendation circumvented this obvious problem, efficient processing of these queries is nevertheless a huge challenge theoretically as well as practically [2, 3]. Based on these results, the recommendation has been modified since then, taking the feasible alternative of enumerating paths using the traditional set semantics of results – similar to those used in XML/XPath, instead of impractical bag semantics. The queries with the simplified semantics can be evaluated easily using a fast path computation technique such as the join-based shortest path processing [4].

However, in most practical uses of property paths it is sufficient to answer the *reachability* version of the query – since the variable bindings are allowed only on the path end points. Reachability queries can be answered very efficiently using the recently proposed compact in-memory FERRARI index structure [8]. We use multiple instances of the FERRARI index – one for each subgraph induced by an edge label in the graph – and implement a physical operator to evaluate the reachability variant of property paths using corresponding index instances.

Since the property paths have been added to the standard very recently, there is currently no SPARQL benchmark that fully tests the navigational capabilities of RDF systems. In this paper we identify the aspects of the system design (primarily the query optimizer's design) that are crucial for efficient support of SPARQL path queries. Informally, these are the things that systems needs to get right in order to process complex SPARQL queries over large RDF graphs. We believe that identifying such points will help formulating a comprehensive benchmark for this new SPARQL standard.

The rest of the paper is organized as follows. We start with describing the syntax and semantics of reachability queries in SPARQL 1.1 in Section 2.1 and then briefly overview the system architecture in Section 2.2. Section 2.3 presents the underlying FERRARI index structure for reachability queries. In Section 2.4 we describe our main contribution, the reachability query processing in RDF-3X, which is based on query optimization and runtime processing techniques for reachability queries. In Section 3 we study the query optimization problems that have to be reflected in benchmarks for path queries and give experimental comparison of regular path query processing in RDF-3X and Virtuoso.

2. PATH QUERY PROCESSING

2.1 Syntax & Semantics

The simpliest SPARQL query with a regular path expression has the form `SELECT ?s ?o WHERE {?s path ?o}` and retrieves the nodes `?s` and `?o` connected via the path that matches a regular expression `path` (as usual in SPARQL, the names of variables are started with ?). In this paper we consider regular expressions over constant predicates with disjunction (denoted by '|'), path concatenation ('/') and Kleene star ('*') (corresponding to zero or more occurences of a predicate) and its variant '+' (one or more occurences). The expression specifies a sequence of predicates along the path from `?s` to `?o` in the RDF graph. We call a triple pattern (`?s, path, ?o`) with regular path expression a *regular path pattern*.

As an example, Query 1 retrieves the entities that can be reached from `Berlin` by the path consisting of zero or more predicates `isLocatedIn` and then gets the types of these entities. In other words, it looks up the geographical hierarchy of Berlin (Berlin, Germany, Europe, Earth) and returns the types of the corresponding objects (i.e., state, a member of EU, a continent, a planet etc)

```
SELECT * WHERE {
 Berlin isLocatedIn*/type ?type
}
```

Figure 1: Path Query Example

The early version of the W3C SPARQL standard allowed asking how many paths between two nodes matching the given regular expression exist. However, the part of the W3C standard defining counting property paths was demonstrated to be computationally intractable. It has been shown [2, 6] that the problem lies in using bag semantics for path queries, that is, in counting all the different paths that can reach a given node via a specified sequence of predicates. Counting paths leads to returning multiple copies of the same node if it is reachable by several distinct paths. This results in a double exponential lower bound on the result set size of a single query even for very small graphs and simple regular expressions like `predicate*`, rendering the original W3C semantics infeasible. It is also known [2] that even restricting ourselves to simple paths (without repeating nodes) does not make counting easier, neither the acyclicity of the underlying graph.

For these reasons, the current W3C standard suggests (and our system supports) an intuitive existential semantics that merely checks if there exists a specified path between two nodes, without counting the number of such paths. In other words, we treat the regular path pattern (`?s, path, ?o`) as a *reachability query* that returns all pairs of nodes reachable by the given path. Naturally, `?s` and `?o` may appear in other triple patterns as well, thus restricting us to subjects with specific properties that can reach objects with other properties (of course, these properties can be expressed with arbitrary complex SPARQL subqueries)

2.2 System Architecture

We assume our system is a triple store, i. e. all triples are stored in one giant table with Subject (S), Predicate (P), and Object (O) as columns. Additionally, each subject, predicate, and object string is mapped to an integer id and stored in global dictionary. Some permutations of S, P and O (say, POS and OPS) are indexed using separate B$^+$-trees, and merge and hash joins operate directly on these indexes. In this work, we build upon the RDF-3X system that indexes all six permutations, similar extensions could be provided for a triple store applying a less aggressive indexing scheme. Finally, the query execution plan is picked by a cost-based optimizer. Again, we extend an existing RDF-3X dynamic programming approach, but our findings apply to all cost-based query engines.

In order to support the reachability queries in a triple store, we add the capability to check whether two nodes are connected (this is done by a special reachability index), as well as describe the query optimization and selectivity estimation methods for such queries. Finally, we describe the runtime technique to speed up the plan execution.

2.3 Physical Operator

The physical operator we employ for answering reachability queries is based on the recently proposed FERRARI index structure [8]. In this section, we describe this index structure together with the necessary graph-theoretical background and briefly review related.

2.3.1 Graph Reachability

The problem of reachability in graphs, or – more precisely – quickly processing reachability queries of the form (G, u, v), where G denotes a directed graph and u, v a pair of vertices, has been actively studied in recent years. The goal in this setting is to determine in near-constant time whether the input graph G contains a (directed) path originating in node u and ending in node v. In terms of computational complexity, the reachability problem is rather lightweight, in fact, a simple graph traversal operation (BFS/DFS) originating from the source vertex u is sufficient to determine reachability in time $O(m + n)$. However, in many scenarios, especially for the case of web-scale graphs, query processing in sublinear time is desirable.

In some cases, a complete precomputation of the transitive closure of the graph is feasible. In this setting, for every node $v \in V$ a directed edge is added to each other vertex $w \in V$ that is reachable from v in the original graph via a path of arbitrary length. By storing the resulting graph in an ajdacency matrix representation, a single lookup is sufficient in order to answer a reachability query, thus enabling query processing in constant time. Given the worst-case time and space complexity of $O(mn)$ to compute and $O(n^2)$ to store the transitive closure can render this approach infeasible for many large graphs. Therefore, in order to achieve rapid query processing on these instances, so-called *reachability index structures* have been proposed in the past. In the next section we briefly review the basic approach of interval labeling for reachability indexing.

2.3.2 Reachability Index Structures

An important observation regarding the reachability relationship in directed graphs is, that vertices from the same strongly connected component are identical with respect to reachability. In other words, a pair of mutually reachable vertices can reach exactly the same set of other nodes in the graph. For this reason, all proposed reachability index structures operate on the acyclic (DAG) structure obtained after collapsing the maximal strongly components into supernodes, commonly referred to as *condensed graph*. Then, in order to answer reachability queries on the original graph, the query nodes are first mapped to their respecitve supernode. If the supernode (corresponding to a strongly connected component) is identical, the query can be immediately terminated with a positive answer. Otherwise, the index is probed to determine the reachability of the respective supernodes in the DAG structure.

The classical work of Agrawal et al. [1] can be regarded as the first reachability index structure. As a first step, the algorithm determines a spanning tree of the input graph and proceeds with a depth-first traversal in order to assign numeric postorder identifiers. More precisely, during the tree traversal every node is assigned an id in ascending order as soon as all the children of the node have received their respective id. This postorder identifier for a node $v \in V$ is denoted by $\pi(v) \in \{1, 2, \ldots, n\}$. What makes postorder labeling interesting for reachability indexing is, that for each (complete) subtree $T[v]$ of V, rooted at vertex an arbitrary vertex v, the postorder ids of the nodes contained in the subtree form a contiguous sequence of integers. Therefore, the set of vertices reachable from a node v in the tree T can be expressed by a single interval

$$I_T(v) = \left[\min_{w \in T[v]} \pi(w), \pi(v) \right] \tag{1}$$

and a reachability query for the pair of nodes (v, w) on this tree can be answered in constant by time simply by determining wheter the postorder id of the target node is contained in the interval of the source node:

$$v \sim w \quad \iff \pi(w) \in I_T(v), \tag{2}$$

where $v \sim w$ denotes that w is reachable from v via a sequence of directed edges. In order to generalize this approach to general DAGs, Agrawal et al. propose to label each vertex v in the graph with a set of intervals, $\mathcal{I}(v)$, since a single interval is only sufficient to represent reachability in tree structures. Their algorithm works by first computing a spanning tree of the graph and assigning the tree intervals $I_T(v)$ and initializing the interval sets as $\mathcal{I}(v) = \{I_T(v)\}$. Then, vertices are visited in reverse topological order and for the currently considered vertex v, the interval sets assigned to the children are *merged* into the interval set $\mathcal{I}(v)$, where subsumed intervals are discarded. Afterwards, reachability queries can be answered by checking whether the postorder id of the target node is contained in *one* of the intervals contained in the set assigned to the source node. The time complexity for query processing amounts to $O(\log(n))$, assuming that the intervals are maintained in sorted order in each set.

2.3.3 The FERRARI Index Structure

Since above interval representation can be regarded as a concise representation of the transitive closure of the graph, it shares the worst-case time complexity of construction and representation with the basic approach of complete precomputation. Therefore, in many settings this interval labeling approach can become infeasible. In order to maintain the benefits of the compact interval representation while overcoming the scalability problems of the original algorithm, the recently proposed FERRARI index structure [8] assigns a mixture of exact and approximate intervals to the nodes of the graph. The main idea of the algorithm is to restrict the total number of intervals that can be assigned to the nodes (local restriction) or overall to all nodes in the graph (global restriction). This restriction is generally specified by the user and offers a direct control over the query processing time vs. index size tradeoff. The basic principle of the index structure is to cover the intervals that would be assinged to the nodes with a smaller number of approximate and exact id ranges with the requirement that all reachable ids are covered and false positives are possible. More precisley, the algorithm repeatedly merges adjacent interval into one longer, approximate intervals (that contains as false positive entries the ids originally located in the gap between intervals). At query processing time, a query can terminated directly with (i) a positive answer if the target id is contained in an exact interval of the source node or (ii) a negative answer if the target id is located outside the intervals assigned to the source. If the id falls into an approximate id range, the FERRARI index structure employs a online search procedure (DFS with additional heuristics) in order to determine the actual reachability. In our setting, we employ the FERRARI index structure in order to precompute the reachability relationships for the subgraph induced by restricting the edges to a certain (RDF) property. By adaptively compressing the transitive closure using exact and approximate intervals the FERRARI index structure can also provide the set of ids of reachable nodes rather than just providing a boolean answer for a pair of query nodes. For this purpose, the algorithm performs a BFS traversal from the source node up to the point where all the expansion fringe consists of vertices labeled exclusively with exact intervals.

2.3.4 Indexing RDF datasets

The reachability index described above does not take into account different predicate labels. Therefore, for every distinct predicate in the RDF dataset, we create a separate index for the subgraph induced by edges with that predicate label. In order to reduce the overhead, we only consider predicates that label both incoming and outgoing edges from the same node, since these are the only predicates that can form paths in the graph. In reality, the YAGO2S dataset (100 million triples) contains 15 such predicates, the sizes of the corresponding induced subgraphs vary from few hundred nodes to 5 million nodes and the total indexing time amounts to roughly 90 seconds on an off-the-shelf laptop computer. The total size of all FERRARI indexes amounts to 210 MB, permitting to maintain them in main memory.

2.4 Query Processing

2.4.1 Query Graph and Logical Operators

The initial step in getting an optimal query plan for the query is to transform it into a calculus representation (query graph) for further optimizations. Prior to introducing property paths, each SPARQL triple pattern would be translated into a node of the query graph, and edges would be formed by shared variables between different triple patterns. We now map new regular path triples onto query graph nodes and edges in the following way. First, if the regular expression on the path contains a sequence of steps (i. e., the path concatenation '/' is used), we replace such a triple pattern with a sequence of patterns, introducing temporary variables. This way, the pattern `?s isLocatedIn⋆/type ?o` expands into two patterns `?s isLocatedIn⋆ ?tmp . ?tmp type ?o`, where the second pattern does not require any path processing. From now on, we can assume that every regular path expression simply requires matching a single predicate zero or more times (or one or more types for the '+' operator). Then, every such path triple pattern is expressing the reachability requirement on its subject and object, that is, the subject should reach the object via the given path. Such a requirement is encoded as a special *reachability edge* between all the patterns containing subject and object of the given path reachability triple.

```
SELECT ?city ?p ?type WHERE {
 ?city hasName "Berlin".
 ?city hasPopulation ?p.
 ?city isLocatedIn+/type ?type.
}
```

Figure 2: Query for geographical hierarchy and population of Berlin

For example, the Query 2 will have its second triple pattern rewritten into

```
?city isLocatedIn+ ?tmp.  ?tmp type ?type
```

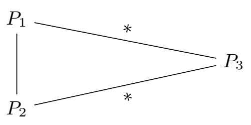
The corresponding query graph, depicted in Figure 3a, therefore has three nodes. The edge between P_1 and P_2 expresses the fact that these two patterns share the variable `?c`, while two reachability edges (depicted with *-label) mean that there should exist a path between `?c` defined in P_1 and P_2 and `?tmp` in P_3.

Nodes and edges in the query graph correspond to logical operators in the query plan. As usual, nodes are mapped to scans over the entire database (or the corresponding index) with selections induced by literals. Edges are transformed into joins, such that regular edges become equi-joins and reachability edges turn into *reachability joins*. A reachability join $\bowtie_R (?s, ?o)$ is conceptually a join with the condition *?s reaches ?o* as a join predicate (as opposed to $?s = ?o$ condition in an equi-join). For Query 2, the two reachability joins and an equi-join are given in Figure 3b. Out of them, the join tree in Figure 3c is constructed. Note that since the query graph is a clique, only two of the three potential joins are used. Indeed, the equi-join between P_1 and P_2 and the reachability join between their result and P_3 guarantees that all three join conditions are satisfied.

There are two special cases of reachability joins:

- Either subject or object in the corresponding reachability triple pattern is constant (e.g., in triple `Berlin locatedIn⋆ ?place`). Then the reachability join $\bowtie_R (?s, ?o)$ turns into a reachability selection $\sigma_R(const, ?o)$.
- Either subject or object is unbound, i.e. does not occur in any other triple pattern. In this case $\bowtie_R$ becomes a reachability scan: for a bounded variable it looks up all the reachable nodes via the specified path.

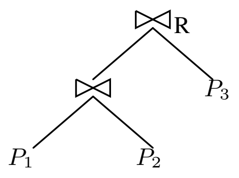
$P_1 = (?c, hasName, Berlin)$
$P_2 = (?c, hasPopulation, ?p)$
$P_3 = (?tmp, type, ?type)$



(a) Triple Patterns and Query Graph

$P_1 \bowtie P_2$
$P_1 \bowtie_R P_3$
$P_2 \bowtie_R P_3$

(b) Potential Joins



(c) Join Tree

Figure 3: Translating the query into the join tree

2.4.2 Query Optimization and Physical Operators

After we interpreted the query graph in terms of table scans, projections and joins, we can run one of the classical join ordering algorithms to obtain the optimal query plan. In order to enable the cost-based query optimization, we need to extend the cost model to account for reachability joins and selections. More specifically, the optimizer has to estimate the result sizes (cardinalities) of reachability selections and reachability joins.

To estimate the cardinality of $\sigma_R(const, ?o)$ recall that the FERRARI index for the $const$ element contains sequence of exact and approximate reachability intervals corresponding to the set of nodes reachable from a particular vertex. The intervals are approximate in the sense that all reachable nodes are covered (i. e., nodes outside the intervals are not reachable from $const$), while some nodes inside the approximate intervals can yield a false positive answer to the reachability query. In practice, we could observe that, for the YAGO2S dataset, the structure of the subgraphs induced by the individual predicates permits a labeling of exact intervals to all of the respective nodes with a modest size budget. Thus, it is not necessary to compress the interval labeling by introducing approximate intervals and therefore the intervals correspond exactly to the nodes reachable from $const$. The total size of these intervals therefore serves as a fast and accurate approximation for the cardinality of the $\sigma_R(const, ?o)$ operator.

In order to estimate the cardinality of $\bowtie_R (?s, ?o)$, we precompute and materialize for every predicate the average number of nodes contained in the set of intervals of a vertex in the FERRARI index, i.e. the total size of intervals for all nodes divided by the number of nodes. This value provides a rough approximation of the number of nodes reachable from the fixed node `?s`. Then, to estimate the result size of $\bowtie_R (?s, ?o)$ we multiply this number by the cardinality of the subplan yielding the `?s` values (that is, by the expected number of start nodes).

Once the optimal logical plan has been found, the physical operators are constructed. In the pipeline model, the reachability join operator is implemented similarly to a hash join: in the build part, the more selective input subplan is executed, and then in the probe part the output of the second subplan is checked against the build part and FERRARI index, to find the nodes that are reachable from any of the nodes in the build part. The reachability scan and reachability filter simply probe the FERRARI index for every incoming node to filter those that satisfy reachability constraints.

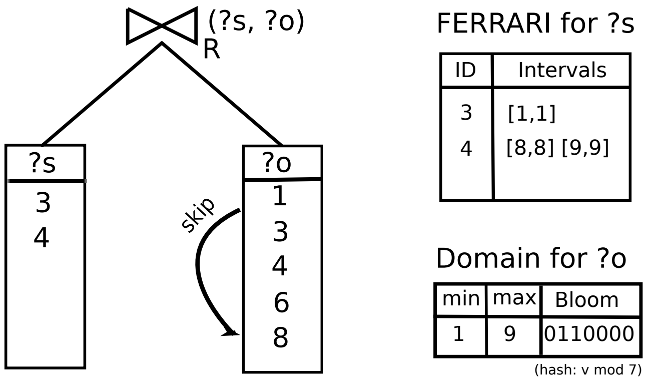


Figure 4: Sideways Information Passing

2.4.3 Sideways Information Passing for Path Queries

Even after the optimal query plan has beed identified by the system, its execution suffers from the fact that individual triple patterns may be surprisingly unselective, while their combinations (i. e., joins) rule out most of the scanned data. In order to deal with these phenomena, RDF-3X employs the Sideways Information Passing (SIP) mechanism [7]. The system keeps the information about significant gaps occured in scans and merges and about domains of hash joins. All this information is passed between different index scans via shared-memory variables.

Unlike merge joins and index scans, our reachability join operator does not keep the order of the input join variables values, so we can not identify potential gaps in its output and pass it to the next operators. However, we can still leverage the SIP mechanism to speed up the reachability join itself. Recall that the intervals of the FERRARI index contain all the reachable nodes for all the `?s` values of the left side. By keeping these reachability intervals in the shared memory for the `?o` variable of the right side, we can notify the right side about the potential gaps in the values of `?o`. Intuitively, during the build time we identify for every `?s` value the nodes that *can not* be reached from s, thus allowing to skip these values in the right side subplan execution.

Instead of keeping in the shared memory the intervals themselves, we encode these potentially reachable nodes in a Bloom filter. As the Bloom filter will be probed for absence of certain ranges of values, we use the range-preserving hash function of the form $h(x) = ax \mod m$ [7]. We also keep the minimum and maximum values of potentially reachable nodes, since they can also guide the underlying index scans of the right side to skip some fractions of the data.

The SIP mechanism for the reachability join is illustrated in Figure 4. Suppose that the inputs to the reachability join are two index scans, and during the probe time we identify that the left scan yields the values 3 and 4. Then, by looking up the corresponding intervals in the reachability index, we figure that the potential domain for `?o` variable has values 1, 8, 9. We encode these values in a Bloom filter, and now during the right scan we know that we can skip all values from 1 to 8, potentially also skipping several disk pages between these values.

As a concrete example, consider again Query 2 and its optimal plan in Figure 3c. The left side (build part) of the reachability join gets a very selective subplan (effectively, its one tuple lookup), whereas the right side (probe part) entails a very expensive index scan that touches a large portion of the database. The SIP mechanism provides the right side the hints to skip the most part of the scanned data, thus significantly improving the performance: on a commodity laptop the running time has decreased from 2193 ms to just 75 ms. We describe the details of our experimental evaluation in the following Section 3.3.

3. EVALUATION

In this section we discuss the challenges inherent to SPARQL 1.1 path queries that the system needs to cope with in order to efficiently support complex queries over real-world data. These potential issues help identify interesting queries for benchmarking systems. We then compare our RDF-3X with path support against the open-source Virtuoso RDF store. We start our discussion with briefly describing the dataset in use and our experimental setup.

3.1 Dataset and Experimental Setup

In our experiments we formulate queries against YAGO2S [5], a knowledge base harvested from Wikipedia. It contains around 100 million facts about 10 million entities and therefore represents a prime example of a large real world dataset.

We run the modified RDF-3X system on a dual-core laptop equipped with 4 GB of main memory using 64-bit Linux (2.6.35 kernel). The Virtuoso system is run on a server with two quad-core Intel Xeon CPUs (2.93GHz) and with 64GB of main memory using Redhat Enterprise Linux 5.4.

3.2 Choosing the queries

Here we discuss the technical problems that could prevent the system from finding the optimal plan for the reachability query in SPARQL. All these issues are related to the cardinality estimation, since it is at the core of any cost-based query optimizer. The discussion here is independent of our actual implementation, and may serve as a foundation for benchmark proposal. We provide small query examples to illustrate the issues, and also use these ideas in our test queries (see Appendix A).

Choosing the right build part.

Since the reachability join operator is quite similar to a hash join, it is important that the more selective subplan is used during the build phase. This requires the system to provide accurate cardinality estimations for both subplans (which in turn may be arbitrarily complex subqueries). Naturally, a good benchmark query should allow to influence the cardinality of subplans by proper choice of parameters, so that depending on the parameter one of the subplans is much more selective. Query 5 illustrates this idea. If `X` is set to be `Host_cities_of_the_Summer_Olympic_Games` and `Y` is `yagoGeoEntity` (a general type describing any geographical entity), then clearly the first pattern has to serve as a build part. On the other hand, `X = yagoGeoEntity` and `Y= Continent` reverse the situation.

```
SELECT * WHERE {
  ?entity1 type %X%.
  ?entity1 linksTo* ?entity2.
  ?entity2 type %Y%.
}
```

Figure 5: Build Part Selection

The `linksTo` property is used in YAGO to represent the links between Wikipedia pages corresponding to the entities.

Comparing cardinalities of different property paths.

Another situation in which it is important to accurately etimate cardinalities occurs when a query contains several reachability triple patterns, and the optimizer needs to order them. The naive heuristics that the triple pattern with a constant in it is more selective, does not always work here. Consider Query 6, where the pattern with `isMarriedTo*` is more selective than the pattern with `isLocatedIn*`, since YAGO contains way fewer entities with the `isMarriedTo` predicate than entities located in the United States. The optimal plan would, therefore, start with the reachability scan on `isMarriedTo*` joined with the `owns` triple, and finally filter the entities situated in the US.

```
SELECT * WHERE {
 ?person isMarriedTo* ?spouse.
 ?spouse owns ?entity.
 ?entity isLocatedIn* United_States.
}
```

Figure 6: Two Reachability Triple Patterns

Cardinality of property path vs index scan .

Although the path queries typically touch a significant fraction of data (as in the previous example), sometimes they can actually be very selective. This calls for a very accurate cardinality estimation of property path triples. As an example, Query 7 entails the join between an index scan on the `type` predicate, and a reachability scan on the `hasAcademicAdvisor` predicate. Since very few triples in YAGO are incident with latter predicate, the reachability scan has much smaller cardinality than the index scan. The optimal plan would therefore start building the hash table from the reachability scan output, and then probe the index scan.

```
SELECT * WHERE {
 ?person hasAcademicAdvisor* ?scientist.
 ?scientist type ?t
```

Figure 7: Selective Reachability Triple Pattern

3.3 Experimental Evaluation

In this section we briefly report on the experimental comparison between RDF-3X (with FERRARI) and Virtuoso. The table below reports the warm cache results of seven queries against YAGO2S. As we see, our approach outperforms Virtuoso by large margine, providing the runtime that is 3-5 times faster than the competitor's. Virtuoso could not execute the last query, the query execution proccess has reported "out of memory" exception. Note also that the Virtuoso instance has allocated 20 Gb of main memory, whereas RDF-3X used less around 1 Gb of main memory for query execution.

The full text of all queries can be found in Appendix A.

	Q1	Q2	Q3	Q4	Q5	Q6
RDF-3x	1	188	1	75	350	253
Virtuoso	8	452	4	418	946	–

Table 1: Running times, ms

4. CONCLUSIONS

Efficient querying and processing of graph structured data remains a challenging problem with manifold application scenarios in contemporary systems. The RDF data representation with its schema-less approach has found widespread use, stirred by its origins in the Semantic Web community. While recent research addresses a multitude of problems in querying and storing RDF data, providing more expressive querying mechanisms can be regarded as one of the most promising directions. To this end, in this paper we have presented the first steps towards extending existing RDF query processing systems to solve a subset of the newly introduced Property Path standard for SPARQL 1.1. More precisely, we have integrated the recently proposed graph reachability index, FERRARI, into the state-of-the-art RDF processor, RDF-3X. Multiple instances of the reachability index, one for each relevant property of the underlying RDF graph are used to efficiently process the subset of new SPARQL 1.1 Property Path queries that are arguably most relevant in practical use. Our experimental evaluation over the well-known RDF knowledge base, YAGO2, underpins that our combination of a reachability index as physical operator together with the RDF-3X engine improve the previous approaches for query processing over this kind of queries by a large margin. Finally we describe the query optimization challenges and propose a set of benchmark queries that reflect these challenges with the goal of enabling an insightful comparison of more expressive RDF processing engines that will be proposed in the future.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *SIGMOD'89*, pages 253–262. ACM, 1989.

[2] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *WWW'12*, pages 629–638, New York, NY, USA, 2012. ACM.

[3] M. Arenas, C. Gutierrez, D. P. Miranker, J. Pérez, and J. F. Sequeda. Querying semantic data on the web? *SIGMOD Rec.*, 41(4):6–17, Jan. 2013.

[4] A. Gubichev and T. Neumann. Path query processing on very large rdf graphs. In *WebDB*, 2011.

[5] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artificial Intelligence*, 2013.

[6] K. Losemann and W. Martens. The complexity of evaluating path expressions in sparql. In *PODS'12*, pages 101–112, New York, NY, USA, 2012. ACM.

[7] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD'09*, pages 627–640, New York, NY, USA, 2009. ACM.

[8] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing. In *ICDE'2013*. IEEE, 2013.

APPENDIX

A. QUERIES

All queries use the following prefix: yago:<http://yago-knowledge.org/resource/>

Q1 select ?country ?area where { yago:Berlin yago:isLocatedIn* ?country. ?country yago:dealsWith ?area. ?area rdf:type yago:wikicategory_Member_states_of_NATO }

Q2 select ?city ?b ?area where { ?city rdf:type yago:wikicategory_Capitals_in_Europe . ?city yago:isLocatedIn* ?b. ?b yago:dealsWith ?area }

Q3 select ?a ?b ?area where { ?a yago:isLocatedIn* ?b. ?b yago:dealsWith ?area. ?a yago:isPreferredMeaningOf "Berlin"@eng}

Q4 select ?a ?b ?type where { ?a yago:isPreferredMeaningOf "Berlin"@eng. ?a yago:isLocatedIn* ?b. ?b type ?type. }

Q5 select * where { ?person yago:isMarriedTo* ?spouse. ?spouse yago:owns ?entity. ?entity yago:isLocatedIn* yago:United_States } limit 100

Q6 select * where { ?airport1 a yago:wikicategory_Airports_in_the_Netherlands. ?airport1 yago:hasLongitude ?long. ?airport1 yago:hasLatitude ?lat. ?airport1 yago:isConnectedTo* ?place. ?place a yago:wikicategory_Mediterranean_port_cities_and_towns_in_Spain. ?place yago:wasCreatedOnDate ?date. ?place yago:hasNumberOfPeople ?people. } limit 10