Shortcut Selection in RDF Databases

Vicky Dritsou [#*1], Panos Constantopoulos [#*2], Antonios Deligiannakis [†*3], Yannis Kotidis [#*4]

[#] *Department of Informatics, Athens University of Economics and Business, Greece*
[†] *Department of Electronic and Computer Engineering, Technical University of Crete, Greece*
[*] *Digital Curation Unit, IMIS, Athena Research Centre, Greece*
[1] vdritsou@aueb.gr, [2] panosc@aueb.gr, [3] adeli@softnet.tuc.gr, [4] kotidis@aueb.gr

***Abstract*—An increasing amount of data produced nowadays is in RDF format. While significant work has been performed on view selection and query optimization algorithms in relational database systems, little attention has been paid to the problem of optimizing the performance of query workloads in RDF databases. In this paper we propose the notion of shortcuts, as a method for reducing the query processing cost. We then devise a greedy algorithm that, given a space constraint and a query workload in an RDF database, seeks to select the optimal shortcuts to materialize. Our experiments validate our approach and demonstrate that our algorithm manages to significantly reduce the query processing cost while keeping space requirements low.**

I. Introduction

Accessing information on the Web is greatly enhanced by semantic overlays that support the semantic integration of and unified access to disparate sources. This is part of the promise of the Semantic Web and is in practice effected by employing ontologies, in the role of conceptual models accepted within specific communities (which may range from small to global), expressed in representation languages recommended by W3C, namely RDF(S) and OWL.

The structural part of these ontologies is commonly seen as a directed graph with nodes representing concepts and edges representing relations between concepts ("properties" in RDF jargon). An issue that invariably arises when formulating ontology-based queries is the need to traverse particular paths (sequences of nodes and edges) repeatedly. From a practical point of view, accessing information through paths, especially long ones, raises certain problems: first, users need to be exposed to the underlying schema to formulate a query, even though they may remain indifferent to those details; second, the computation of the answer they seek can get expensive, as it requires the traversal of large data graphs containing instances of the ontology. Independently of the model used to store these graphs (RDF/OWL files, relational databases, etc.), path expressions require substantial processing; when using relational tables, this often results in multiple join expressions that can be hard to process [4], [15], [17], [22].

In the relational world, views have long been considered as a solution to the issues raised above. For example, relational views help achieve schema independence [21] by hiding certain details of the underlying schema from the end-user. Alternatively, views can be seen as "shortcuts" that conceal certain schema details enabling easier formulation of user queries and reducing the risks of errors when formulating queries over complex schemata. If a solution similar to the concept of materialized views is desired for RDF databases, we need a formalism that can help us first understand the available space of solutions, and subsequently let us design efficient algorithms that will automate the view selection process. In RDF databases, given the prevalence of path expressions, it is only logical to consider schema paths as the basic unit of materialization. In this paper we, thus, propose to augment the schema and instance graphs of an RDF database with the creation of appropriate *shortcuts* corresponding to frequently accessed paths, so that the query formulation and answering process is more easily and efficiently performed. These shortcuts have a direct analogy to relational views. Similar to a view, the instance of a shortcut does not have an independent existence but is rather composed from the instance-level data paths that evaluate the schema-path denoted by the shortcut.

Considering possible shortcuts in a large RDF database gives rise to an optimization problem in which we seek to select shortcuts that maximize the reduction of query processing cost subject to a given space constraint. Our proposed algorithm is based on estimates on the execution cost of either entire, or parts of queries. Thus, it does not assume (i) any storage representation of the RDF data (relational or vertically partitioned [4]), or (ii) the way that the execution cost estimates are obtained (i.e., using query optimizer estimates, or performing the actual queries and measuring the costs). Thus, our algorithm is applicable independently of the used storage representation or the way that query estimates are obtained.

Unfortunately, the selection of schema paths as the basic candidate unit of materialization leads to a space of possible solutions that is exponentially large with respect to the size of the RDF schema graph. In our work, we manage to prune a large number of shortcuts that cannot possibly help expedite an available workload of queries. As it turns out that the shortcut selection problem is a knapsack problem, which is known to be NP-hard, it does not admit a guaranteed efficient optimal solution. So we develop a greedy algorithm that incrementally selects shortcuts based on the notion of *per unit space benefit*, i.e. a quantification of the expected decrease in query cost that the selection of a given shortcut provides, divided by the space needed to materialize the shortcut.

The contributions of this work are: (i) we formally define the notion of a shortcut as a structural augmentation of an existing RDF database; (ii) we describe a framework for decomposing a set of user queries that describe popular user requests into a set of simple path expressions in the RDF schema graph, while these paths are then combined in order to form a set of candidate shortcuts that can help during the

evaluation of the original queries; (iii) we describe the shortcut selection problem where we evaluate the expected benefit of reducing query processing cost by materializing a shortcut in the knowledge base, under a given space constraint and describe a greedy algorithm for solving this problem; and (iv) we provide an extensive set of experiments where we evaluate our proposed algorithm.

II. Related Work

Relational views have long been an important aspect of database management systems [23]. Depending on the context, relational views have been treated as pure programs [25], [26], derived data that can be further queried [18], pure data (e.g. detached from the view query) and pure index [24]. Materialized views have been recently rediscovered within the context of OLAP and data warehouses. A flurry of papers has been generated on how views can be used to accelerate frequent computations over massive datasets. Selecting and materializing a proper set of views with respect to the query workload and the available system resources has been at the core of this discussion [13], [16]. Key to the popularity and acceptance of the *view selection* problem has been the existence of the multidimensional framework of the Data Cube, which restricts the solution space to a well defined set of views with well understood containment properties [11]. In the context of RDF databases such a framework is lacking.

The vision of Semantic Web requires methods and systems able to handle efficiently query processing over large amounts of semantic data. These are mostly expressed in RDF and/or OWL and a variety of systems have been developed towards this direction, including among others Jena[1], SWKM[2] management system, relying on a backend relational DBMS, and Sesame[3]. Many RDF query languages have been proposed in the literature for querying such systems with the most popular of them being SPARQL [3], RQL [2] and RDQL [1].

Indexing techniques for RDF databases have already been explored [10], [19], [28]. In [7] the authors propose techniques for selecting which (out of all available) indices can help accelerate a given SPARQL query, and then determine how to execute and rewrite the query. Our techniques are orthogonal and can be used in conjunction with these approaches to accelerate query processing, since they target the problem of determining the proper set of shortcuts to materialize. Indexing RDF data also seems similar in concept (due to the hierarchical nature of path queries) to indexing data in object oriented (OO) and XML Databases. A good overview of indexing techniques for OO databases is presented in [5]. The work of [6] presented techniques for selecting the optimal index configuration in OO databases when only a single path is considered, without taking overlaps of subpaths into account. In [12] a uniform indexing scheme, termed U-index, for OO databases is presented. The work of [8] targets building indices over XML data. Unlike XML data, RDF data is not rooted at a single node. Moreover, the above three techniques target the creation of indices over paths, thus not tackling the problem of this paper, which is the selection of shortcuts to materialize given a workload of *overlapping* queries.

[1]Available at http://jena.sourceforge.net/

[2]Available at http://139.91.183.30:9090/SWKM/

[3]Available at http://www.openrdf.org/

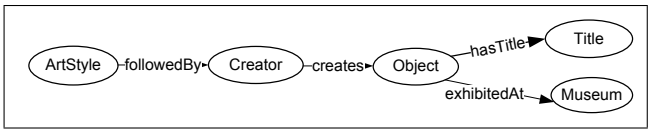

Fig. 1: Sample Schema Graph

In a recent work [4], the authors propose a novel vertically partitioned approach to storing RDF triples and consider materialized path expression joins for improving performance. While their approach is similar in spirit to the notion of shortcuts we introduce, the path expression joins they consider are hand-crafted (thus, there is no automated way to generate them in an arbitrary graph). Moreover the proposed solutions are tailored to relational backends (and more precisely column stores), while in our work we do not assume a particular storage model for the RDF database. Thus, our techniques can be applied on the model proposed in [4]. Finally, in [20] the authors propose a novel architecture for indexing and querying RDF data. While that system handles large RDF data sets in an efficient way using indices and optimizes the execution cost of small path queries, applying our technique to their system further reduces the execution cost for long path queries.

III. Shortcuts

Given a workload of user queries, we define the notion of a query *shortcut*, and then present a way to generate the set of shortcuts which, if materialized, can help speed up the execution of the queries in our workload.

Database Conceptual Representation. We assume an RDF database G comprising two parts: (i) a schema graph $G_S(V_S, E_S)$, consisting of a set of nodes V_S, representing entity classes (or, concepts) and a set of edges E_S, representing relationship classes (or, properties); and (ii) a data graph $G_D(V_D, E_D)$, consisting of a set of nodes V_D which are instances of nodes in V_S and a set of edges E_D which are instances of edges in E_S. More generally, G could be any graph-structured semantic database. Figure 1 demonstrates a sample schema graph of an RDF database, where all oval shapes (e.g. "Creator") are examples of entity classes, while edges (e.g. "creates") denote the relationship classes (properties).

Queries and Query Fragments. A query workload consists of a set of queries $Q = \{q_1, q_2, q_3, \ldots, q_m\}$, where each query q_i is associated with a frequency of occurrence f_i.

In this paper we focus on data-level queries: a query q_i in Q is defined on the schema graph G_S and specifies a subset of relevant elements of the data graph G_D (thus, the queries retrieve data from the "raw" data of the instance graph). A query q can be represented as a weakly connected subgraph G_q of the schema graph G_S, where the subgraph originates

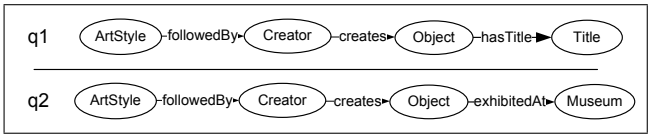

Fig. 2: Example Queries q_1 and q_2

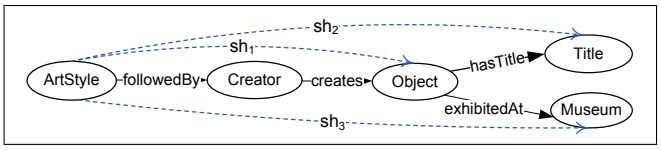

Fig. 3: Candidate Shortcuts of Example Queries q_1 and q_2.

and terminates at entity nodes. Essentially, the subgraph G_q defines at the schema level the structural properties of the data that the query seeks to retrieve from the data graph. Consider for example the graph presented in Figure 1 and the queries q_1: "For each art style that creators of objects follow, find the title of these objects"; and q_2: "For each art style that creators of objects follow, find the museums where the objects are exhibited at". Figure 2 depicts these two sample queries. Regardless of the query language used, the important issue is that the evaluation of a query requires traversing the corresponding query path (for example, $q_1 = ArtStyle \overset{followedBy}{\rightarrow} Creator \overset{creates}{\rightarrow} Object \overset{hasTitle}{\rightarrow} Title$) in order to find the matching instances of the query path in the data graph G_D.

The two aforementioned queries represent *simple path* queries. However, more complex queries are also plausible. One can consider, for example, a query that retrieves instances of $Title$s of artworks of any $ArtStyle$, along with the $Museum$s they are exhibited at. The graphic representation of such a query is exactly what is depicted in Figure 1. Note that this sample query essentially requires "joining" information from two different subpaths.

We define the *length* of a simple path query to be equal to the number of relationships (edges) that appear in it. For any query $q_i \in Q$, any subpath of q_i of length greater than 1 is called a *query fragment* in our framework. As will become clear shortly, any shortcut considered for materialization matches a query fragment of the query workload.

Candidate shortcuts. Given a query workload Q, containing $|Q|$ queries, and L the length of the query having the longest path among all queries in Q, the number of query fragments is $O(|Q| \times L^2)$. Recall that a shortcut in our framework matches a query fragment. Since the selection of the "optimal" set of shortcuts to materialize under a space constraint is NP-hard, in order to reduce the size of our search space we seek to limit the set of *candidate shortcuts* that our algorithm considers. A natural way to achieve this, is to consider only shortcuts that originate from or terminate to (user-defined) "interesting" nodes of V_S, which we shall call *candidate shortcut nodes*. In particular, we consider a node $v \in G_S$ to be a candidate shortcut node if at least one of the following conditions is true for v:

1) v is the origin or endpoint of a query q,
2) v is a starting node of two different edges e_i, e_j, $e_i \neq e_j$ traversed by one or more queries in Q,
3) v is the end of two different edges e_i, e_j, $e_i \neq e_j$ traversed by one or more queries in Q.

Having defined the set of candidate nodes, we then develop the set of candidate shortcuts SH by considering all valid combinations between candidate shortcut nodes, so that they are connected through an existing path of the ontology and this path is traversed by at least one query.

Example: Figure 3 shows all the candidate shortcuts for the queries of Figure 2. The node $ArtStyle$ is a candidate shortcut node, since it is the origin of the two queries. Similarly, nodes $Title$ and $Museum$ are also candidate shortcut nodes, since they are the last nodes traversed in the queries. Finally, node $Object$ is a candidate shortcut node, since two different edges ($hasTitle$ and $exhibitedAt$) that appear in the two queries originate from it.

Recall that any candidate shortcut $sh_i \in SH$ maps to a query fragment, and that the corresponding query fragment may be contained in more that one queries, which will be called *related queries of* sh_i and denoted as RQ_i. In the example of Figure 3, the set of related queries of the candidate shortcut sh_1 is $RQ_1 = \{q_1, q_2\}$. Two query shortcuts are *disjoint* if the query fragments that they map to do not share any edge. Two shortcuts that are not disjoint have one of the following two *containment relationships*: *full containment* or *overlap*. A shortcut sh_i is *fully contained* in sh_j iff the query fragment that sh_i maps to is a subpath of the corresponding query fragment of sh_j, denoted hereafter as $sh_i \prec sh_j$. A shortcut sh_i *overlaps* sh_j if they are not disjoint, and none of them fully contains the other.

IV. Benefit of Shortcuts and Problem Definiton

We now turn to defining the benefit of introducing a shortcut and describing shortcut selection as a benefit maximization problem. Assume a candidate shortcut sh_i with underlying query fragment qf_i and its set of related queries RQ_i. Let c_i denote the estimated cost of retrieving the answer to qf_i. As explained in Section I, this cost depends on the storage representation of the RDF data (i.e., relational or vertically partitioned). Our algorithm is based on estimates of the c_i values, and is applicable independently of how these costs are obtained (i.e., using query optimizer estimates, or performing the actual queries and measuring their costs).

Now suppose that we augment the database by shortcut sh_i. This involves inserting one edge in the schema graph G_S, while in the data graph G_D one edge (i.e., RDF triple) is inserted for each result of qf_i[4] . Let c_i' denote the new cost of answering qf_i given the materialization of shortcut sh_i.

We define the benefit obtained by introducing shortcut sh_i in order to answer qf_i to be equal to the difference between the initial cost minus the new cost: $c_i - c_i'$. Since qf_i is used in answering each of its related queries, the benefit for query $q_k \in RQ_i$ can be estimated by multiplying the fragment benefit by the frequency of the query: $f_k \times (c_i - c_i')$. The total benefit obtained by introducing shortcut sh_i is then equal to the sum of the benefits obtained for each related query.

If the query fragments underlying the candidate shortcuts are disjoint then the aggregate benefit is the sum of the benefits of all candidate shortcuts. If, however, there are containment relationships between fragments, things get more complicated. We illustrate this through an example. Consider again the aforementioned queries q_1 and q_2 of Figure 2 and candidate

[4]By inserting one instance of sh_i for each result of qf_i and allowing duplicate instances of shortcuts (triples) using RDF Bags, we obtain the same query results, since the result cardinalities remain unchanged.

shortcuts sh_1 and sh_2 of Figure 3. Assume now that shortcut sh_1 has been implemented first and that adding sh_2 is being considered. The benefit of adding sh_2 with regard to query q_1 will be smaller than what it would have been without sh_1 in place, since sh_1 reduces the cost of a part of q_1. We denote as d_{ij} the difference in the cost required to answer qf_i due to the existence of a shortcut induced by qf_j. This difference is positive for all $qf_j \prec qf_i$. Finally, shortcuts induced by overlapping query fragments do not interfere in the above sense: since the starting node of one of them is internal to the other, they cannot have common related queries.

Let us now turn to the space consumption entailed by using shortcuts. The augmentation of the RDF database by a shortcut sh_i has a space footprint denoted as sp_i. This space can be evaluated exactly by issuing the corresponding query and observing the size of the result (number of RDF triples), or by exploiting statistics of the query optimizer. The total space consumption of all shortcuts actually introduced should not exceed some given space allocation expressed as a budget B.

The goal is to select the set of shortcuts to materialize, such that (i) the selected shortcuts fit within the space budget B; (ii) the benefit of the selected shortcuts is the maximum among all the sets that fit within the space budget B. This is a knapsack problem, which is known to be NP-hard [14].

V. Greedy Algorithm for Shortcut Selection

In order to approximately solve this specific variation of knapsack, we developed a greedy algorithm (presented in Algorithm 1). This algorithm takes as input a space budget b for the materialization of shortcuts and the set of candidate shortcuts, which are obtained as explained in Section III. Each candidate shortcut sh_i is associated: (i) **space**: the actual space for materializing sh_i; (ii) **benefit**: the estimated benefit of materializing sh_i; (iii) **L1**: a list of shortcuts in which sh_i is fully contained; (iv) **L2**: a list of shortcuts which sh_i fully contains; and (v) **QP**: a list of query paths that "traverse" sh_i. The use of the lists L1, L2 and QP will become evident when we discuss the adjustment of the benefits of each shortcut after our algorithm selects a candidate for materialization.

The candidate shortcuts are stored in an AVL-tree, where the sorting key of its elements is their per unit space benefit. This structure allows us to perform insert(), remove() and removeMax() operations in logarithmic time with respect to the number of candidate shortcuts. For each shortcut in $sh_i.L1$ and $sh_i.L2$, the algorithm maintains pointers to other nodes in the AVL-tree, in order to efficiently be able to locate the nodes in these lists.

The algorithm operates in an iterative manner, always maintaining the amount of budget *SpaceLeft* that remains to be allocated to shortcuts. Our greedy algorithm repeatedly selects the candidate shortcut $candSh$ with the largest per-space benefit and removes it from the AVL-tree. If the shortcut requires more space than the available space *SpaceLeft*, then this shortcut is ignored. Otherwise, the selected shortcut $candSh$ is chosen for materialization (at a later step). Before proceeding to the next candidate shortcut, the algorithm needs to update certain benefit estimates as a result of the most recent

Algorithm 1 GreedyAddShortcuts

Require: Space budget B, AVL-tree $candShortcuts$ containing the candidate shortcuts {
Entries in AVL-tree sorted by per-space benefit
Each candidate shortcut sh_i contains the following info:
space: Estimated space for materializing sh_i
benefit: Estimated benefit of materializing sh_i
L1: list of shortcuts in which sh_i is fully contained
L2: list of shortcuts which sh_i fully contains
QP: list of query paths that "traverse" sh_i
L1 and L2 lists contain pointers to other nodes in AVL-tree}

$SpaceLeft = B$
while $SpaceLeft > 0$ AND $|candShortcuts| > 0$ **do**
  $candSh = candShortcuts.removeMax()$ {Remove shortcut with maximum per-space benefit}
  if $candSh.space > SpaceLeft$ **then**
    continue {cannot materialize shortcut}
  end if
  Mark $candSh$ for materialization
  $SpaceLeft- = candSh.space$
  for all $biggerSh \in candSh.L1$ **do**
    {Cost of $biggerSh$ reduced due to $candSh$}
    candShortcuts.remove($biggerSh$)
    commonQueries = $biggerSh.QP \bigcap candSh.QP$ {Queries where both $biggerSh$ and $candSh$ are used}
    for all $q_j \in commonQueries$ **do**
      Update $biggerSh$.benefit given addition of $candSh$, utilizing frequency f_j of q_j in calculation
    end for
    Remove $candSh$ from the L2 list of $biggerSh$
    if $biggerSh.benefit > 0$ **then**
      $candShortcuts.insert(biggerSh)$ {Re-insert}
    end if
  end for
  for all $smallerSh \in candSh.L2$ **do**
    {Shortcuts fully contained in $candSh$ may become useless for some queries. Update their benefit}
    candShortcuts.remove($smallerSh$)
    commonQueries = $smallerSh.QP \bigcap candSh.QP$ {Queries where both $smallerSh$ and $candSh$ are used}
    for all $q_j \in commonQueries$ **do**
      Update $smallerSh$.benefit given addition of $candSh$, utilizing frequency f_j of q_j in calculation
    end for
    Remove $candSh$ from the L1 list of $smallerSh$
    if $smallerSh.benefit > 0$ **then**
      $candShortcuts.insert(smallerSh)$ {Re-insert}
    end if
  end for
end while

shortcut insertion. In particular:

Shortcuts that fully contain $candSh$. The algorithm iterates over all candidate shortcuts $biggerSh$ that appear in $candSh.L1$ and removes them from the AVL-tree one-by-one. The estimated cost of each query that contained $biggerSh$ is now reduced given the materialization of $candSh$. Thus, the expected benefit of $biggerSh$ is also reduced, since part of its computation has been made easier, due to $candSh$. In order to obtain the benefit of $biggerSh$, we first need to observe that $biggerSh$ and $candSh$ are not necessarily useful for the same set of queries. Some queries may contain both shortcuts. For such queries, the benefit of $biggerSh$ is reduced, based on the frequency of these queries in the query workload. For queries where only $candSh$ is useful, no action needs to be taken. A final note is that, given the removal of $candSh$ from the AVL-tree, we also need to remove $candSh$ from the $L2$ list of $biggerSh$ (please note that $biggerSh$ was in the $L1$ list of $candSh$). Finally, if $biggerSh$ is still useful (i.e., has a benefit larger than 0), it is re-inserted in the AVL-tree.

Shortcuts fully contained in $candSh$. The operation of the algorithm for shortcuts fully contained in $candSh$ is similar. The algorithm iterates over all such candidate shortcuts $smallerSh$ that appear in $candSh.L2$ and starts by removing

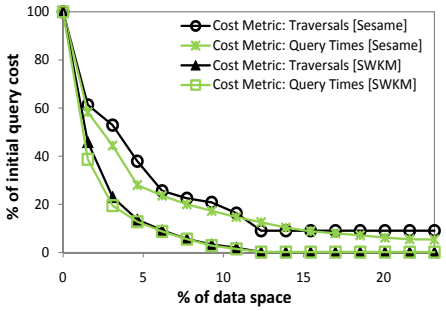

Fig. 4: Using the Yago Data Set.

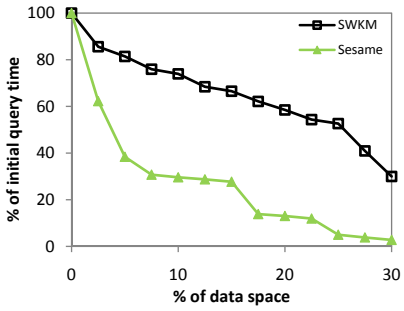

Fig. 5: Using the CIDOC Ontology.

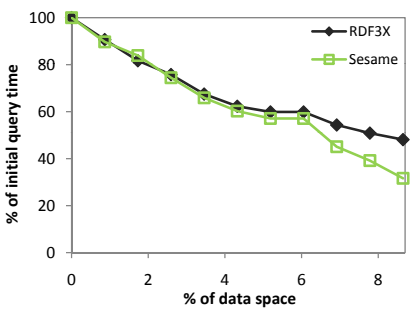

Fig. 6: Using Long Path Queries.

them from the AVL-tree one-by-one. The estimated cost of each query that contained $smallerSh$ is now reduced given the materialization of $candSh$, as $smallerSh$ may not be useful for some queries where $candSh$ is also useful. Thus, the expected benefit of $smallerSh$ is reduced. In order to obtain the benefit of $smallerSh$, for the set of queries where both $smallerSh$ and $candSh$ are useful, the benefit of $smallerSh$ is reduced, based on the frequency of these queries in the query workload. We then remove $candSh$ from the $L1$ list of $smallerSh$. Finally, if $biggerSh$ is still useful (i.e., has a benefit larger than 0), it is re-inserted in the AVL-tree.

VI. Evaluation

In order to evaluate our technique, we implemented our algorithm in C++ and compared the query execution times before and after the augmentation of the database in question with the selected shortcuts. We used for this purpose three systems with different underlying architecture: (i) the SWKM Semantic management system, which relies on a backend relational DBMS, (ii) the native Sesame repository and (iii) the RDF3X system, that optimizes the query performance by creating a specific set of indices. Thus, we tested whether our technique can be beneficial with regard to different types of RDF stores. We also used real and synthetic data sets derived from uniform and non-uniform schema graphs. All reported experiments were executed on a Intel Core 2 Duo 2.33GHz PC with 4GB RAM running 32bit Windows Vista. Our greedy algorithm required at most 2193 secs to execute for the largest data set of 4 million RDF triples that we tested.

In an attempt to provide a fair and unbiased evaluation of the algorithm, instead of obtaining approximate statistics from the storage engine, we computed in a preprocessing step exact estimates of the costs and sizes of the queries and shortcuts, by probing the underlying repositories. These statistics were then fed as input to our implemented algorithm. We considered two approaches to determining query cost: the first takes as query cost the number of edges required to be traversed in order to retrieve the answer of the query, while the second takes the actual time required to execute a query.

In this spirit, we first tested the reduction in query time achieved by our algorithm using the Yago data set ([27]) consisting of 1.62 million triples, while the query workload consisted of the 233 distinct query paths of length 3 contained in the Yago schema. We experimented with two different RDF store systems, namely SWKM and Sesame, that do not share common architecture features; SWKM relies on a relational DBMS, while Sesame relies on a native triple store system. These two storage systems, along with the two different types of input statistics (traversals and actual query costs) yield four possible combinations that are presented in Figure 4. The lines in this Figure depict the reduction in the execution time for the entire workload after the materialization of the proposed shortcuts (y-axis) w.r.t. the space consumption these shortcuts require (x-axis).

The results deriving from these experiments are twofold; first, it is shown that the materialization of shortcuts significantly reduces the query time for small space consumptions. Indeed, in all cases depicted we achieve a reduction of at least 40% of the initial query time by consuming space equal to only 1,5% of the data set. Moreover, materialized shortcuts that require 23% of the data size reduce the query time by at least 90% in both systems and for all cost metrics. On the other hand, it is obvious that both cost metrics yield almost identical behaviors with each system. Therefore, the choice between the two query cost metrics does not affect the performance of our algorithm.

In order to evaluate our algorithm with different kinds of schema and query workload, we performed the experiment presented in Figure 5. Here we test our greedy algorithm using the popular ISO 21127 CIDOC CRM ontology [9], which contains 74 nodes and 111 properties at the schema level. This ontology has a different schema type than Yago. Its schema graph forms a "constellation graph" structure with non-uniform density. More precisely, it contains a small number of "star" nodes, each connected with a large number of of "planet" nodes and sparsely connected with other star nodes. We generated synthetic data for this graph by using a Perl script that picks a schema node at random, generates one data node from it, and then produces all the data edges emanating from the latter according to the schema graph. This process is then repeated on the newly created ending data nodes of the produced edges. A parameter p defining the probability of breaking a path (stop following the edges of the current node and continue with a new one) is also used, which in this experiment is set to 0.4. The data set generated for this experiment comprises of 1 million triples. Regarding the query workload, we first extracted from the schema graph all possible paths of length at least 2 and then reduced this set to include only path queries that are strongly correlated. By this we mean queries that share all or almost all of their edges with many of the other queries. Our objective here was to test whether our algorithm is also able to capture the dependencies

among such query fragments. The final set of queries used in this experiment contains 65 path queries with path lengths varying from 2 to 4. We obtained these path queries by using a Perl script that, given a schema graph, identifies all possible path queries of a specific length provided as input parameter. Figure 5 shows the relative reduction achieved in the total query time of the workload w.r.t. the fraction of data space that needs to be consumed when using SWKM and Sesame respectively. Since we have showed that the query cost metric does not affect the performance of the algorithm, we have used here as query cost the number of traversals required. Again our algorithm significantly reduces the query time, while the best performance is achieved when using Sesame: by consuming 30% of the data space it reduces the query time by 95%.

The results obtained from experiments with the RDF3X system are different depending on the length of the path queries. RDF3X creates a set of indices when loading the data, so that it can efficiently reduce query execution time, and requires for these indices, according to the authors, space size that is less than the data size. Indeed, adding shortcuts to this system does not improve query execution time when path queries have length up to 4. However, in Figure 6 we demonstrate the results obtained by using a query workload of 26 queries with lengths varying from 10 to 15 edges, obtained again by the same Perl script mentioned in the previous experiment. We generated a synthetic schema graph containing 190 nodes and 467 edges from which a data set of 4 million triples was generated using the above Perl script with a probability of breaking a path of 0.2. The query cost metric used here was again the number of traversals. The results show that our technique still significantly reduces the query execution time for small consumption of space: by consuming approx. 8% of the data set space we reduce the total query time by 50%. Note here that the authors in [20] state that the space required for the indices they generate and which reduce the query execution time is less than the space of the data set. However, the consumption presented in the experiment by our approach is far less than this initial space. We also depict in Figure 6 the reduction in query time when using Sesame, where the results are slightly better: by consuming 8% of the data space we achieve a reduction of 65% in query time.

In summary, our experimental study shows that the augmentation of RDF databases with shortcuts reduces significantly the query time of the workload. We observed this reduction for different types of RDF stores, using both uniform and non-uniform schema graphs, for real and synthetic data sets, for real and synthetic ontologies, with different types of query workloads and for different database sizes.

VII. Conclusions

The popularity of the Semantic Web and the adaptation of ontologies by many disciplines results in large RDF databases that are being developed in different domains. While the data management community has significant experience in optimizing queries via the use of materialized views in relational databases, there is considerable lack of similar techniques on the emerging problem of optimizing path queries over large RDF instance graphs. In this work, we first introduced the notion of a shortcut, as the basic structural augmentation unit of an existing RDF database, and then we introduced the shortcut selection problem: how to select the best set of shortcuts that reduce query processing cost under a given space constraint. We then proposed a greedy algorithm that seeks to maximize the benefit of the selected shortcuts and our experiments demonstrate that the proposed technique significantly reduces the cost of processing common queries, using a small fraction of the space required for storing the complete data graph.

References

[1] RDQL - A Query language for RDF. *W3C Member*. Available at http://www.w3.org/Submission/RDQL/.

[2] RQL Query Language. Available at http://139.91.183.30:9090/RDF/RQL/.

[3] SPARQL Query Language for RDF. *W3C Recommendation*. Available at http://www.w3.org/TR/rdf-sparql-query/.

[4] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, 2007.

[5] E. Bertino. A Survey of Indexing Techniques for Object-Oriented Database Management Systems. *Query Processing for Advanced Database Systems*, 1994.

[6] E. Bertino. Index Configuration in Object-Oriented Databases. *The VLDB Journal*, 3(3), 1994.

[7] R. Castillo, U. Leser, and C. Rothe. RDFMatView: Indexing RDF Data for SPARQL Queries. Technical report, Humboldt University, 2010.

[8] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *VLDB*, 2001.

[9] N. Crofts, M. Doerr, T. Gill, S. Stead, and M. S. (editors). Definition of the cidoc conceptual reference model, January 2010.

[10] G. H. L. Fletcher and P. W. Beck. Indexing social semantic data. In *ISWC'08 (Posters & Demos)*, 2008.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE*, 1996.

[12] E. Gudes. A Uniform Indexing Scheme for Object-Oriented Databases. *Information Systems*, 22(4), 1997.

[13] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In *SIGMOD Conference*, 1996.

[14] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[15] Y. Kotidis. Extending the Data Warehouse for Service Provisioning Data. *Data Knowl. Eng.*, 59(3), 2006.

[16] Y. Kotidis and N. Roussopoulos. A Case for Dynamic View Management. *ACM Trans. Database Syst.*, 26(4), 2001.

[17] P. Larson and V. Deshpande. A File Structure Supporting Traversal Recursion. In *SIGMOD Conference*, 1989.

[18] P. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *VLDB*, 1985.

[19] B. Liu and B. Hu. Path Queries Based RDF Index. In *SKG*, Washington, DC, USA, 2005.

[20] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1), 2010.

[21] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2nd edition, 2000.

[22] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal Recursion: A Practical Approach to Supporting Recursive Applications. In *SIGMOD Conference*, 1986.

[23] N. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Record*, 27, 1997.

[24] N. Roussopoulos, C.-M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS Project: View R Us. *IEEE Data Eng. Bull.*, 18(2), 1995.

[25] T. K. Sellis. Efficiently Supporting Procedures in Relational Database Systems. In *SIGMOD Conference*, 1987.

[26] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *SIGMOD Conference*, 1975.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *WWW*, New York, NY, USA, 2007. ACM Press.

[28] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, 2007.