

R²DF Framework for Ranked Path Queries over Weighted RDF Graphs*

Juan P. Cedeño
CIDSE Arizona State University
Tempe, AZ, 85287, USA
juan.cedeno@asu.edu

K. Selçuk Candan
CIDSE Arizona State University
Tempe, AZ, 85287, USA
candan@asu.edu

ABSTRACT

Resource Description Framework (RDF) is a semantic web specification that aims to support conceptual modeling of information about resources in the form of a triples of facts. In this paper, we note that, although RDF provides mechanisms to encode meta-information (such as source, trust, or certainty) about facts recorded in the knowledge base, existing RDF query languages and RDF stores fail to support key primitives needed in a large class of knowledge applications which associate *utilities* or *costs* on the available knowledge statements. To address this shortcoming, we propose a novel R²DF framework for utility ranked resource descriptions. We first propose a simple *ranked RDF* (R2DF) specification to enhance RDF triples with an application specific weight (e.g. cost). We then propose a SPARankQL query language specification, which includes a set of novel primitives on top of the SPARQL language to express top-*k* queries using traditional query patterns as well as novel flexible *path predicates*. An extended query processor engine, AR2Q, leverages novel index structures to support efficient ranked path search and includes query optimization strategies based on two key metrics: (a) proximity and (b) sub-result inter-arrival time. Experiments show that the use of these two metrics has significant impacts on the performance of top-*k* queries over R2DF graphs: in particular, the proximity measure helps reduce the number of path matches that need to be considered, whereas the inter-arrival measure reduces the overall execution time significantly especially when used along with proximity. The proposed strategies help obtain query plans close to optimal.

Categories and Subject Descriptors

I.2.4 [Knowledge Representation Formalisms and Methods];
H.2 [Database Management]

General Terms

Algorithms, Theory, Experimentation, Performance

1. INTRODUCTION

*Supported by the NSF Grant NSF-III1016921. "One Size Does Not Fit All: Empowering the User with User-Driven Integration."

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WIMS '11, May 25-27, 2011, Sogndal, Norway

Copyright 2011 ACM 978-1-4503-0148-0/11/05 ...\$10.00.

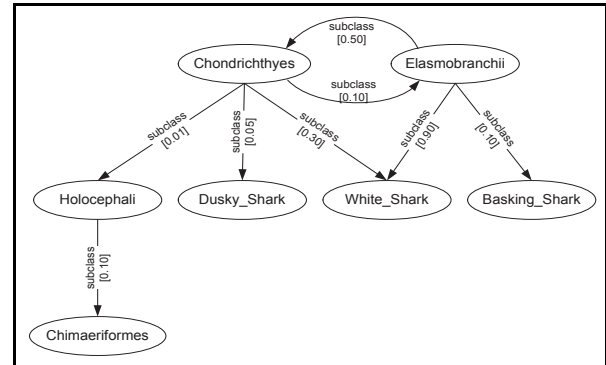


Figure 1: A sample knowledge base where the available information have varying utilities (adapted from [39, 40])

The Resource Description Framework (RDF) [31], introduced by the World Wide Web Consortium (W3C), aims to support modeling of metadata and information about web resources: in RDF, available knowledge is represented in the form of a directed graph in which the nodes represent known entities and the edges represent relationships between pairs of these entities; a pair of nodes *a* and *b* linked by an edge labeled *e* is called *triple* and states a fact about *a*, i.e. “*a* has property *e* with value *b*” or that “*a* has the relationship *e* with the entity *b*”. In RDF, the subject, *a*, does not need to be an external resource, but can also be another triple in the knowledge base. Through this –so-called *reification*– process of associating statements to other existing statements, RDF can also encode meta-information (such as source, trust, and certainty) about the statements in the knowledge base.

Most applications access RDF knowledge bases through declarative query languages, such as SPARQL [38] and SeRQL [36]. These knowledge bases often reside in general purpose RDF stores, such as Jena TDB [3], which provide indexing and optimization mechanisms to support efficient query processing. In this paper, we note that existing RDF query languages and RDF stores fail to support key primitives needed in a large class of applications which associate *utilities* or *costs* on the available knowledge statements.

EXAMPLE 1. Figure 1 shows a knowledge base (an RDF graph) where each available knowledge statements (i.e., triple or edge) has an associated value. This example is adapted from [39, 40], where the weighted graph represents integration of two conflicting shark taxonomies and each edge value indicates the amount of source disagreement on the corresponding assertion.

The ability to annotate statements in the database is especially important for applications which needs to impose additional parameters, like validity, trust, or preference. For instance, as shown in the above example, web applications using RDF data integrated

from several sources may base their preference on the level of conflict or agreement at which these data are combined with those from other sources participating in the integration [39]; software agents may prefer certain pieces of information over others based on trust values associated, not with the source itself, but with the data produced by it [28]; or web retrieval applications may model user preferences or context using weighted ontologies [15].

Note that while reification enables encoding of such meta-information, the ability to reify statements does not imply that the additional information can be effectively leveraged:

- *Flexible path expressions*: While RDF supports complex knowledge graphs, most RDF query languages provide only limited patterns for querying these. One commonly required primitive that is missing in existing frameworks is a flexible *path* primitive to allow the user to assess how two entities in the knowledge base are connected (i.e., whether there is a path in the knowledge graph from one entity to the other)¹.

For example, a user querying a web citation network of authors and publications may be interested in finding how two authors are related by means of other authors or publications or how close an author is to another one in terms of common co-authors or publications.

In existing RDF query languages, the user needs to have knowledge of the schema of the underlying RDF graph to precisely specify the number of edges, and their labels, on the path. More general path queries that can allow users to discover indirect relationships between two nodes connected by an arbitrary number of intermediate nodes and edges, without prior knowledge about the relationships, cannot be specified or evaluated using existing RDF frameworks.

Secondly, when the knowledge statements have *utilities* or *costs* associated to them, users may want to locate a few results:

- *Ranked query processing*: In many knowledge applications, the utility or cost of the elements in the knowledge base to a particular task varies and users are interested not in all the results to their queries, but the ones that are best suited for the given task. Locating such high utility or low cost results is known as top-*k* (or ranked) query processing.

For example, an archaeologist querying an online knowledge base integrating knowledge from several (potentially conflicting) sources may want to retrieve the top-10 results with the lowest amount of conflict among the data sources [39].

These type of functionalities are currently not supported by existing RDF query languages [38, 42, 37, 36]. This necessitate new language primitives and a query processing system extended in order to take advantage of the extra utility or cost information associated with the RDF statements.

1.1 Contributions: R²DF Framework

In this paper, we propose a novel R²DF framework that addresses the needs of applications that require utility/cost ranked resource descriptions. The specific contributions include the following:

- *R2DF model*: An extension of the RDF that allows application specific weights attached to the triples (Section 3).
- *SPARankQL language*: An extension of the SPARQL to enable users to express top-*k* queries using traditional patterns and operators as well as novel path predicates (Section 4). We propose extensions to the SPARQL specification [38] to

support the processing of top-*k* queries over weighted RDF graphs using both the traditional subject-predicate-object query patterns as well as advanced operators to support query patterns that contain reachability predicates.

- *AR2Q engine*: An extended query processing engine to support the SPARankQL language, including the path predicates, ranked join, and novel indexing techniques. To this end, we build on the ARQ implementation included in the Jena Semantic Framework [2, 47]. AR2Q and the novel optimization strategies to improve the performance of queries with new primitives are discussed in Section 5. Experiments show that the proposed strategies help AR2Q obtain query plans very close to optimal.

In Section 6, we experimentally evaluate the optimization strategies presented in this paper and show that they significantly improve query processing efficiency in the presence of the proposed extensions to the SPARQL specification.

1.2 Related Work

In this section, we provide overviews of the literature on reachability and proximity queries, on complex pattern (e.g. path and twig) queries on RDF, and on ranked query processing.

1.2.1 Reachability and Proximity Queries

Recent research in executing reachability queries in graphs includes [21] which proposes an approach in which queries are evaluated with the support of a join index which represents the union of a center-table and a B+ tree. The first contains the center of the hop between two nodes that are reachable from each other, whereas the second contains keys which are a pair composed by a center and a label. A different approach is presented in [29] based on the construction of a path-tree using identified paths in the graph: every path in the graph is represented by a node in the tree along with a 3-component label used to answer reachability queries.

In [14], Candan and Li presented a random walks based proximity measure for mining associations between nodes in a graph. More recent work includes [45] which uses a connection subgraph to find underlying relationships between the query nodes and nodes in the graph. This connection subgraph is constructed by giving each node a goodness score with respect to the query nodes by using random walks with restarts. Similarly, [46] proposes several algorithms to establish a proximity value between nodes in the graph that takes into account the directionality of the edges connecting them. This proximity value is computed using efficient procedures which are based on random walks and escape probability. [43] presents two novel methods that approximate a family of proximity measures, i.e Katz, rooted PageRank, and escape probability, over very large graphs. For this, the authors take a subset (sketch) of columns or rows from the graph's transition matrix and use it to approximate a proximity measure for two given nodes.

1.2.2 Path and Twig Queries on RDF Graphs

Previous work on path search over RDF graphs includes [9, 10, 12], which deal with finding complex relationships between resource entities. Specifically, authors in [9, 10] define *semantic associations* between two entities based on whether (a) there is an ordered sequence of properties (RDF predicates) between two entities, (b) whether both entities are start nodes of two different property sequences that have predicates in common, or (c) whether both entities are start nodes of two different property sequences and both sequences are *ρ -isomorphic* to each other².

¹This is similar to XPath's parent/ancestor axis [7] but within the context of a graph rather than XML trees.

²That is the predicates in both sequences have a parent-child, sibling, or equality RDFS relationship, such as *typeOf*, *subClassOf*, or *subPropertyOf*, with each other

In [8], Anyanwu *et. al* present SPARQ2L, an extension to SPARQL [38], to support path queries over large, disk-resident graphs. The authors define a *path* variable, which can be used as a regular predicate variable, and present a novel technique for path discovery based on LU decomposition, which computes partial graph fragments that are indexed and stored on disk. This work is similar to that presented in [32] for the BRAHMS system.

Research on querying graphs to find more complex structures, i.e. twigs, using the structural relationships of their nodes and edges includes [40, 26]. In [40] the authors present a novel algorithm (HR-Join) to answer ranked twig queries over weighted graphs without assuming the monotonicity property of the results, necessary for ranked join algorithms. The ranking algorithm is based on the weight of the edges of the candidate twigs rather than on the length of their underlying paths. To overcome the non-monotonicity problem the authors make use of a self-punctuating and horizon-based approach. [26] proposes an approach for the evaluation of top-*k* twig queries over disk-resident graphs based on the length of their component paths rather than on their cost. For this, the authors use a runtime graph which is a materialization of the transitive closure of the underlying graph. In a recent work [41], a path algebra, which also contains a join operator for two given sets of paths, is introduced. The authors also provide methods to traverse the graph to find the sets of paths for the join operator. However, like many other works, the authors do not consider the case for ranked path queries over weighted graphs.

1.2.3 Ranked Query Processing

When the number of candidate results is large and when most of these objects have very low scores or utilities, it is *wasteful* to rely on processing strategies that would require the system to touch all inputs and enumerate all possible candidate results. To avoid waste, data processing systems need to employ data structures and algorithms that can prune unpromising data objects from consideration without having to evaluate them. This is often referred to as ranked or top-*k* query processing [23, 11, 16, 19, 30].

Ranked join algorithms rely on weight-sorted input streams for pruning unpromising matches when identifying top matches to the user's monotonic³ query. In one of the earlier works, Fagin [23] proposed an algorithm, known commonly as the FA, which assumes the availability of both the sorted and random (i.e., index-based) accesses to the data. FA considers data from the sources in (progressively) descending order of desirability and, with the help of index-based random accesses, enumerates top-*k* join results without having to access all the data from these sources. Later improvements to FA include the threshold algorithm (TA), which pro-actively schedules random accesses when they are cheap, and the no-random access algorithm (NRA), which is applicable when random-accesses are either unavailable or extremely costly [24]. The variants of these, including [25, 35] and others mentioned earlier, mainly differ in how they *schedule* accesses to the incoming data queues and how sorted and random accesses are alternated.

2. BACKGROUND: RDF PRELIMINARIES

Resource Description Framework (RDF) is an assertional language to express propositions using formal vocabularies [4]. The abstract syntax of RDF [31] defines an RDF graph as a collection of *triples*, where each triple expresses an assertion in the form of a 3-tuple and containing a “subject,” a “predicate,” and an “object.”

Figure 2 shows an example, where two geographical facts about Arizona State University are expressed as triples. In general, a

³An object that is as good as another one in all features is also better than the other when these features are considered simultaneously.

Subject	Predicate	Object
Arizona State University	latitude	33.42
Arizona State University	longitude	-111.93

Figure 2: An example pair of triples

```
@prefix geo: <http://www.w3.org/2003/01
              /geo/wgs84_pos#>.
<http://www.asu.edu> geo:lat  "33.42".
<http://www.asu.edu> geo:long "-111.93".
```

Figure 3: Triples in Figure 2 serialized in the Turtle [6] format

triple expresses a fact (knowledge assertion) between two real-world entities described by the subject and object. A set of RDF triples can, then, be seen as the logical conjunction of the assertions of its underlying triples; a set of RDF statements can also be often viewed as a graph: the subject and object of a triple represent vertices in the RDF graph whereas the predicate represents the edge connecting them.

The W3C also introduced a standard, referred to as RDF/XML [5], for the serialization of RDF statements. In addition to this, there are two other non-standard formats that are widely used due to their ease and readability: Notation 3 (N3) [13] and its extension Turtle [6]. Figure 3 shows the serialized version of the triples of Figure 2 in Turtle format.

2.1 SPARQL Preliminaries

SPARQL is the query language introduced by the W3C to query RDF knowledge bases. It enables to users to express queries in the form of graph patterns that match and retrieve subgraphs from an RDF graph. SPARQL introduces the concept of variable to specify any node or edge in the graph; and the concept of triple pattern which is a construct identical to an RDF triple but that may contain a variable in the place of the subject, predicate, or object [38]. The basic unit of a SPARQL query is the *Basic Graph Pattern* (BGP) which is defined as a set of triple patterns. Essentially, a SPARQL query asks for subgraphs of the underlying RDF graph that *match* the corresponding BGP: a subgraph *s* is a valid match for a BGP *b* if after substituting the variables of *b* with RDF nodes (or edges) of *s*, both *b* and *s* are equal. The subgraphs resulting from substituting the variables and blank nodes of *b* with RDF terms are called a *solution mapping* and an *RDF instance mapping*, respectively.

The steps to process a given SPARQL query include parsing, algebra generation, and evaluation [17, 1, 27]. The parser converts a query string into an *abstract syntax tree* (AST) and the AST is further processed to obtain a tree-like expression containing algebraic operators. The set of operators include (a) *project* to select only a subset of the variables bound in a solution mapping; (b) *bgp* to evaluate basic graph patterns; (c) *filter* to constrain the matches for a BGP to conditions applied to their labels or numeric values; (d) *join*, *union*, and *leftjoin* to perform conjunctions, disjunctions, and optional matching of BGPs respectively; and (e) *order by*, *distinct*, *reduced*, and *slice*, to further modify the set of sub-results [38].

The evaluation of a BGP to find matching subgraphs often comprises the join of the matches for its individual triple patterns. The join operation works under the principle of *solution mapping compatibility*, which states that two mappings are compatible if all their common variables are bound to the same values in both [38].

EXAMPLE 2 (SPARQL QUERY EVALUATION). Consider the RDF graph in Figure 4 containing information about three animal species and their respective locations. The SPARQL query in Figure 5, containing a group graph pattern with two BGPs, searches for the states that have both birds and reptiles.

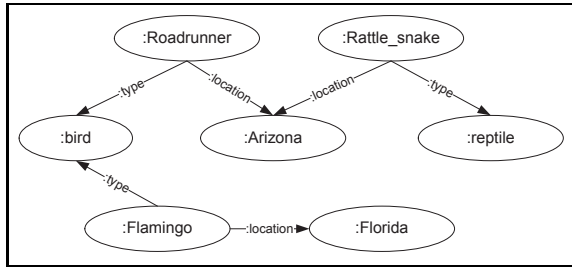


Figure 4: A sample RDF graph

```
SELECT ?state WHERE {
  {?spec1 :type :bird .
   ?spec1 :location ?state}

  {?spec2 :type :reptile .
   ?spec2 :location :?state}
}
```

Figure 5: A SPARQL query over the RDF graph in Figure 4

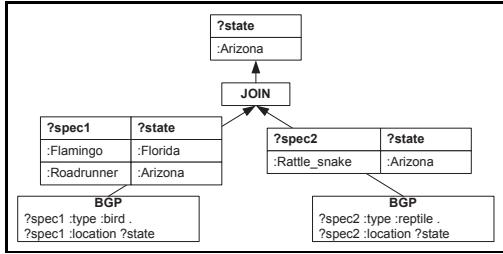


Figure 6: The algebra tree for the SPARQL query in Figure 5

After parsing, the generated algebra expression is evaluated by the query engine starting with the operators at the lowest levels of the tree, i.e. the BGP operators. Figure 6 shows the partial results (shown over the outgoing arrows) for each BGP operator and the final joined result. Note that only the solution mappings corresponding to Roadrunner and Rattle_snake are joined as they are compatible regarding their common variable ?state.

3. R2DF: WEIGHTED RDF MODEL

The first component of the proposed R²DF model is a simple weighted extension of the RDF model, which we refer to as R2DF. R2DF associates to each triple an application-specific numeric value representing a measure of desirability or lack thereof:

- **Weighted triple:** A weighted triple t_w is a 4-tuple composed of Subject (S), Predicate (P), Object (O), and a real value w between 0 and 1 representing the weight of the assertion made by the triple SPO.
- **Weighted RDF graph:** A weighted RDF graph G_w is defined as a set of weighted triples.

EXAMPLE 3 (WEIGHTED RDF KNOWLEDGE BASE).

Figure 7 shows a set of weighted RDF triples, in Turtle format [6], containing the RDF representation of the knowledge base presented in Example 1. In this example, the weight value associated with each edge (RDF triple) represents the amount of disagreement on the corresponding assertion. The weight value extending the original Turtle format is shown in bold. The weighted RDF graph for this triple set was shown earlier in Figure 1.

As the above example shows, in this paper, without loss of generality, triples with weights are represented as quadruples, where

@prefix : <http://purl.org/NET/biol/zoology#> .			
:Chondrichthy	:subclass	:Elasmobranch	0.10 .
:Chondrichthy	:subclass	:Holocephali	0.01 .
:Chondrichthy	:subclass	:Dusky_Shark	0.05 .
:Chondrichthy	:subclass	:White_Shark	0.30 .
:Holocephali	:subclass	:Chimaeriformes	0.10 .
:Elasmobranch	:subclass	:Chondrichthy	0.50 .
:Elasmobranch	:subclass	:White_shark	0.90 .
:Elasmobranch	:subclass	:Basking_Shark	0.10 .

Figure 7: Extended, Turtle-based, representation of a sample weighted RDF knowledge base in Example 1

```
SELECT ?o ?w WHERE {
  :Chondrichthyes ?p ?o ?w .
  FILTER (?w > 0.01)
}
```

Figure 8: Sample query containing a triple pattern extended by a new variable (variable ?w) for the weights of the matches

the final value of the quadruple is the weight of the triple,⁴ instead of relying on reification – which could be used to associate weights to the triples indirectly through reified statements. The reason why we treat weights as *first class* knowledge attributes (as opposed to treating them as any other application specific attribute, which can be specified using reification statements) is that we would like the underlying engine to easily recognize and leverage these weights in indexing the knowledge statement (Section 5.2) and processing users’ ranked queries over the weighted graphs (Section 5). Note that this is not a strict requirement in that the same effect can also be achieved using a *special* reification statement recognized by the underlying indexing and query processing engine.

4. SPARANKQL QUERY LANGUAGE

In this section, we describe the extensions the proposed SPARankQL languages provides on top of SPARQL.

4.1 Language Extensions of SPARankQL

In this section, we introduce the language extensions provided by SPARankQL over the SPARQL. These extensions include operators for weight aware processing as well as operators to support more general path matching.

4.1.1 Weight Extended Triple Patterns

The first extension is the inclusion of an *optional* element, variable or literal, to the triple patterns to reflect the weight associated to the matching results of the given pattern. As a variable, it can be used freely by other patterns or operators, such as filter or select, within the query. Figure 8 presents a sample query, over the graph in Figure 1, to retrieve the objects of the triples that contain :Chondrichthyes as a subject and a weight greater than 0.01.

4.1.2 Path Query Predicates

Secondly, we propose a novel query predicate $\langle nEdges \rangle$ to represent multi-edge relationships between two nodes in an RDF graph; the predicate matches one or more edges irrespective of the labels of the edges or vertices on the path. The new predicate can also be extended to impose optional constraints over the multi-edge relationships represented by it such as label and upper and lower bounds to the weight of the matches to the path pattern.

Figure 9 presents two sample queries over the RDF data set in Figure 1: these queries aim to retrieve all the nodes reachable from

⁴If no weight is provided, the default RDF semantics is assumed.

```
SELECT * WHERE {
  :Chondrichthyes <nEdges> ?s }
(a) Path predicate with no restrictions
```

```
SELECT * WHERE {
  :Chondrichthyes
  <nEdges(subclass) (0.1) (0.3)> ?s }
(b) Path predicate with weight restrictions
```

Figure 9: Sample extended SPARQL queries containing a path predicate which allows multiple matching edges with (a) no restrictions and (b) two additional restrictions on the edge weights to specify lower and upper limits (prefixes omitted)

```
SELECT ?s WHERE {
  ?s <nEdges> :Holocephali .
  ?s ?p1 :Dusky_Shark .
  ?s ?p2 :White_Shark .
}
```

Figure 10: A sample query where path predicates are used in conjunction with other SPARQL constructs

```
SELECT @ WHERE {
  :Chondrichthyes <nEdges> ?o .
  FILTER regex(str(?o), "White_Shark", "i") }
(a) A sample "SELECT @" query
```

```
(1) [0.30]
:Chondrichthyes :subclass :White_Shark 0.30 .
:White_Shark <isVar> ``?o`` 0.0 .
(2) [1.00000]
:Chondrichthyes :subclass :Elasmobranchii 0.10 .
:Elasmobranchii :subclass :White_Shark 0.90 .
:White_Shark <isVar> ``?o`` 0.0 .
```

(b) Results for this query on the RDF graph in Figure 1

Figure 11: (a) A sample "SELECT @" query and (b) the corresponding set of results; each of which is a matching path described in the form of a set of RDF triples (since each triple has a weight, each result is also associated a weight displayed within square brackets – in this example, the overall score of a result is the sum of the corresponding triple scores)

:Chondrichthyes with a path containing *one or more* edges; (a) the first query imposes no restrictions on the paths, whereas (b) the second query with imposes restrictions on the label and on the lower and upper limits of the overall path weights.

Path predicates can be used like any regular query predicate; i.e., the path predicate can be used in conjunction with other query patterns and SPARQL clauses (see Figure 10 for an example).

4.1.3 "SELECT @" Clause

The "SELECT" clause of SPARQL allows the user select the variables that will be included in the output. For example, in Figure 10, the user states she is interested in the values of the ?s variable, whereas the values of the ?p1 and ?p2 variables are to be ignored. When the "SELECT *" clause in Figure 9 is used in a SPARQL query, then the matching values for all the variables in the query are included in the result.

One difficulty with the standard "SELECT" and "SELECT *" clauses is that they allow the query to return only named variables. However, as shown in Figure 9, the path predicate, <nEdges>, matches entire paths, consisting of one or more edges; consequently, when using the standard "SELECT" and "SELECT *" clauses, only the end points of these paths can be associated with the named variables that can be returned to the user. This prevents additional detailed filter queries to be posed on the returned paths.

In order to deal with this shortcoming of the standard SPARQL,

```
SELECT ?s ?w WHERE {
  {?s <nEdges> :Holocephali .
  ?s <nEdges> :Dusky_Shark .
  ?s ?p :White_Shark .
  } SCORE IN ?w .
  FILTER (?w > 0.01)
}
```

Figure 12: Example query showing the SCORE IN clause

```
SELECT ?s ?w WHERE {
  {?s <nEdges> :Holocephali .
  ?s <nEdges> :Dusky_Shark .
  ?s ?p :White_Shark .
  } RANK 10 DECREASING
}
```

Figure 13: Example query showing the RANK clause

we introduce a new "SELECT @" clause, which outputs the entire matching result subgraph in the form of a (serialized) RDF graph. Figure 11 shows an example: as can be seen in this figure, the results of the "SELECT @" is a sequence of matches, where each match is represented using *all* the triples that make up the match. For each result, also the total weight is computed and maintained; also, any variables on this graph are highlighted through a special "isVar" triple. Thus, the user can provide additional conditions on the resulting graph to explore any edges or nodes that have not been explicitly enumerated in the query specification.

4.1.4 "SCORE IN" Clause

Remember from Section 2.1 that the basic unit of a SPARQL query is the *Basic Graph Pattern* (BGP); i.e., a set of triple patterns. In Section 4.1.1, we have seen how triple patterns are extended to handle weights associated to the triples. In the presence of weighted RDF graphs, we also propose a new SCORE IN clause to identify the overall weights of whole solutions for a given BGP.

As shown in Figure 12, the SCORE IN clause takes as a parameter a variable name which will be included in the set of query variables and bound to the weight of each resulting match for the BGP to which the clause belongs; in other words, the variable ?w in the example reflects the weight of each resulting match for the BGP of the query. Note that the score variable can then be used to associate a filter condition on the weight of the entire BGP.

4.1.5 "RANK" Clause

The queries presented so far return all possible matches for the given query patterns. However, the weight information associated with the underlying RDF graph allows calls for the ranking the results returned by a query and possibly limiting their number based on a *k* given by the user. To support this, we propose a RANK clause which takes as a parameter an integer number *k* greater than or equal to 1, indicating the desired number of top results.

Figure 13 shows the use of the RANK clause applied to a query to return the highest scoring 10 BGP matches. Note that the scope of RANK is the same as that of the SCORE IN clause in Figure 12.

4.2 SPARankQL Semantics

In this subsection, we describe how the SPARQL semantics are modified to handle SPARankQL extensions. As described in Section 2.1, the unit of representation in SPARQL [38] is the *basic graph pattern* (BGP), which is expressed as a set of RDF triples that form a graph pattern and that may contain variables as the subjects, predicates, or objects. The BGP is matched against an RDF graph to obtain a solution subgraph whose nodes are bound to the variables of the BGP. A *solution* for a BGP in SPARQL comprises

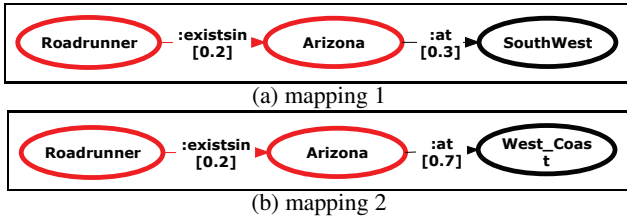


Figure 14: Two mappings sharing a triple

a subgraph along with solution and instance mappings binding the variables to the nodes of the subgraph.

Solution mappings for a BGP over a weighted RDF graph must have weight values associated with them. This calls for the definition of *weighted solution mapping*. A weighted solution mapping μ_w is a pair (μ, w) composed of a solution mapping μ and a real value w representing its weight.

4.2.1 Weight Aggregation

As defined in [38], a solution mapping that is a solution for a BGP represents a solution subgraph. This implies that the weight associated with a weighted solution mapping represents the aggregation of the weights of the triples in the subgraph. Specifically, we need to define weight aggregation functions: a *weight aggregation function*, $aggW(\mu_w, f)$, is a function that determines the aggregated weight of a weighted solution mapping μ_w by applying an aggregation function, f , to the weights of the underlying subgraph of μ_w . Note that the aggregation function, f applied to the weights of the triples of the underlying subgraph may be *sum*, *min*, *max*, *average*, or *product*⁵.

4.2.2 Weight Merging

In addition to simple BGP matching, SPARQL provides operators to allow expressing more complex queries using one or more group graph patterns [38]. They operate over multisets of solution mappings to perform operations like *join*, *union*, *project*, *filter*, *order*, and *distinct*.

Note that some of these operators create new solution mappings by *merging* two solution mappings if they are compatible. In the presence of weighted solution mappings, this merge function needs to be extended to perform the aggregation of the weight values associated with them. A *weight merge function* $mergeW(\mu_{w_1}, \mu_{w_2}, f)$ is a function that determines the aggregated weight of two weighted solution mappings μ_{w_1} and μ_{w_2} by applying a weight aggregation function, f , to the weights of the subgraph resulting from $merge(\mu_{w_1}, \mu_{w_2})$. For example, definition of the weight-aware *join* operator is as follows:

$$Join(\Omega_{w_1}, \Omega_{w_2}, f) = \{ (merge(\mu_1, \mu_2), mergeW(\mu_{w_1}, \mu_{w_2}, f)) \mid \mu_{w_1} \in \Omega_{w_1} \wedge \mu_{w_2} \in \Omega_{w_2} \wedge \mu_1 \text{ and } \mu_2 \text{ are compatible} \}$$

The redefinitions of the rest of operators follow a similar pattern.

Note that, when two mappings μ_{w_1} and μ_{w_2} being merged overlap, there are two possible interpretations of the merge operation, depending on how overlaps are handled. Let us consider the two mappings shown in Figures 14(a) and (b); these two mappings share a triple. When merging these two mappings into a one unified mapping, we can use either *multi-set* or *set* based merge semantics:

- as shown in Figure 15(a), in the multi-set semantics, we count an overlapping triple as many times it occurs;

⁵For our implementation and evaluation, we focused on the *sum* function.

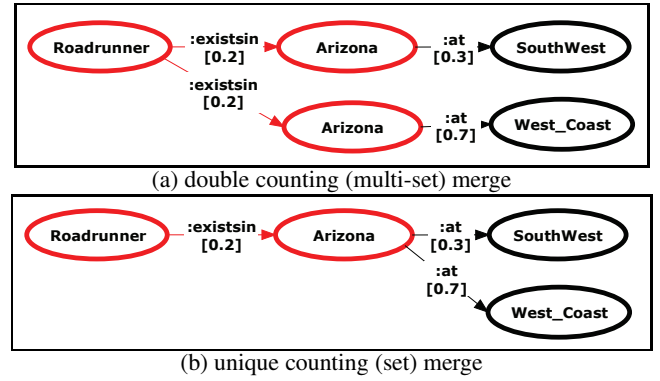


Figure 15: Alternative merge strategies for the mappings in Figure 14

- with the set semantics on the other hand, as shown in Figure 15(b), would count each unique triple only once.

Naturally, the appropriate semantics is application dependent.

5. AR2Q ENGINE FOR SPARANKQL QUERY PROCESSING

Extensions provided by the SPAR2QL language, including the extended triple patterns, path predicates, and the new clauses described in the previous section, necessitate (a) introduction of new algebraic operators, (b) implementation of the physical operators to support these extensions, and (c) reconsideration of the query processing strategies. These are discussed in this section.

5.1 Path Enumerator

Since, in addition to traditional triple patterns, the basic unit of many of the operators of SPARankQL is likely to include *path* patterns, the AR2Q query engine is extended with a *path enumerator* component which returns solution mappings that match user specified path patterns.

Note that depending on the semantics (e.g. utility vs. cost) of the edge weights, we may need to enumerate the matching paths in increasing or decreasing total path weight. Unlike the shortest path problem, which has an efficient polynomial time solution, the longest path problem is known to be NP-complete; but, it can be solved in polynomial time on acyclic graphs by simply rendering the edge weights negative. In this paper, we focus only on the query specifications that have polynomial path enumeration solutions. For this purpose, we use of a version of Yen's k-shortest simple paths algorithm, which returns k shortest simple paths between a given pair of nodes [22] (the original Yen's k-shortest simple paths algorithm is published at [48]). The *path enumerator* provides an iterator that can be used by other operators in the query plan to fetch path pattern results in a pull-based manner⁶; i.e. alternative paths are produced incrementally on-demand, through a *getNext()* call, based on the requirements of the parent operator in the query plan tree. Each path enumerated by the algorithm is processed to create a weighted solution mapping by binding the nodes, predicates (edges), and the weight to variables in the path pattern. Let us assume that the given pair of nodes are embedded within a neighborhood of the graph with N' nodes and M' edges; then, each invocation of the *getNext()* call will take,

⁶This contrasts with the use of Yen's algorithm in [40], in which Yen's algorithm is used with a push-based, streaming, approach where the paths are identified and placed into a buffer space until explicitly instructed to stop the enumeration.

$O(N'(M' + N' \log N'))$ time to execute [48]. In the rest of the paper, we refer to each invocation of the `getNext()` call to a path enumerator as an *access* and in the experiments section, we will evaluate alternative query plans in terms of the number of accesses they require, in addition to the overall query execution times.

5.2 Index Support

Next, we present the index structures used by AR2Q to support the query language extensions described in previous sections.

5.2.1 Weighed Triple (Quadruple) Index

The RDF storage framework of Jena, TDB [3], provides three composite indices, *subject-predicate-object* (SPO), *predicate-object-subject* (POS), and *object-subject-predicate* (OSP) for efficient access to triples for disk-resident RDF graphs. These increase the storage space, database creation time, and time required for updates, but also help avoid full scans of data. Thus, we extend the indexing scheme of TDB in such a way that weighted triples (or quadruples) can be returned in non-increasing order of their weight no matter what literals are available in the query pattern. Thus, the extended indexing scheme contains up to nine composite index structures, SWPO, SPWO, SPOW, PWOS, POWS, POSW, OWSP, OSWP, and OSPW, where W denotes the weight component.

5.2.2 Reachability Index

As stated above, for a given pair of nodes are embedded within a neighborhood of the graph with N' nodes and M' edges, each invocation of the `getNext()` call to a path enumerator will take $O(N'(M' + N' \log N'))$ time to execute. Thus, to reduce the path enumeration time, it would be more effective to consider as small a reachability neighborhood as possible; i.e., only those nodes that are reachable or can reach the non-variable nodes in the path pattern.

The *reachability index* allows recovering, for each node, n , in the graph, the list of nodes for which there is a path from n to them. Note that computation of the reachability neighborhood itself is expensive and this precludes any online methods to recover the reachability information. Therefore, AR2Q creates and use an all-pairs reachability index to recover the reachability information quickly in run-time, without affecting the performance of the query evaluation. Creating an all-pairs reachability index is not cheap, but its cost (for example, $O(N^3)$ for N nodes when using the well known Warshall-Floyd's algorithm or $O(N^2 \log N + NM)$ for N nodes and M edges when using Johnson's algorithm) is often acceptable) as the index structure is created *offline*. In practice cheaper approximation algorithms can also be used.

5.2.3 Proximity Index

As we present in Section 5.4, in order to reduce the number of `getNext()` calls (or *accesses*) to the path enumerators, AR2Q uses a strategy based on proximities of nodes in the R2DF graph.

Intuitively, a node is said to have a *high proximity* to another, if the paths connecting the first node to the second in the graph are large in number, short in length, and have low weights [14, 46, 45]. We argue (and experimentally validate in Section 6) that a query node which has high proximity to the other query nodes has a higher likelihood of producing many desirable matches with the remaining query nodes. Note that, the intuition that high proximity between two nodes in a graph implies the existence of many short and low-cost paths between them is *partly* related to the *betweenness* concept in graph and social network analysis. In particular, if two nodes are close to each other in terms of proximity, the nodes belonging to the paths connecting these nodes are likely to be highly connected to each other and the rest of the nodes (i.e., will have high betweenness).

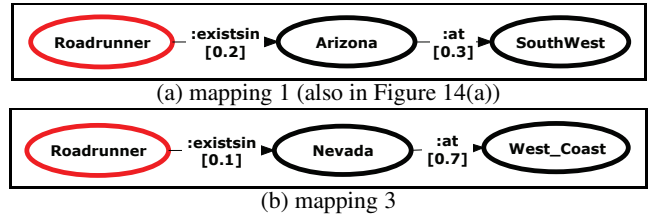


Figure 16: Two mappings sharing a single node

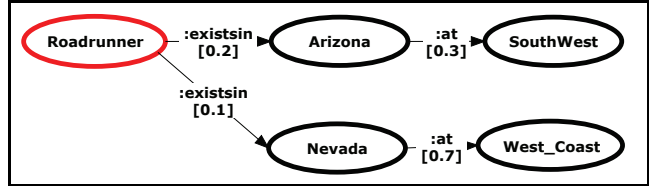


Figure 17: Merge of the mappings in Figure 16

Proximity score, however, is not defined (as in *betweenness centrality*) for the individual nodes of the graph, but (as in [14, 46, 45]) for given pairs of nodes. Thus, the *proximity index* allows recovering, for each node n in the graph, the list of nodes that are very close to it, in terms of the number of paths between them and the length and costs of these paths. Once again, in order to avoid runtime computation of proximity scores, AR2Q uses a variant of the all-pairs proximity computation algorithm presented in [46]. As is the case for reachabilities, for better run-time performance, the all-pairs proximity information is computed *offline* in $O(N^3)$ time, where N is the number of nodes, and stored as an index structure.

5.2.4 Path Cache Index

In addition to all the above index structures, which are computed off-line during data registration, AR2Q can also maintain a path cache index which stores the shortest paths returned by the path enumerator for different path patterns. This path cache is used when a path pattern is also included in future queries.

5.3 Join Operators

The evaluation of a BGP involves (a) matching of its triple and path patterns using available index structures and path enumerators and then (b) joining of the corresponding solution mappings. A join operator working with ranked streams of weighted solution mappings (accessed through `getNext()` calls) also needs to return a ranked sequence of joined weighted solution mappings. As discussed in Section 1.2, there are various ranked join algorithms, such as [34, 24, 20], differing based on various assumptions, such as whether index structures supporting random accesses are available or available or not. Despite these differences, almost all ranked join algorithms assume that the aggregation function being used to combine the ranked inputs is monotonic⁷.

Unfortunately, in SPARankQL, whether the function is monotonic or not depends on whether *set* or *multi-set* semantics is used to combine the mappings. As discussed in Section 4.2.2, in multi-set semantics, we count an overlapping triple as many times it occurs, whereas with the set semantics, we would count each unique triple only once. While the multi-set semantics would ensure that the merge will be monotonic (as long as the underlying aggregation function, f , itself is monotonic), this is not always the case

⁷In this case, monotonicity would imply that a result which is as good as another one in all contributing patterns is also better than the other result when these features are considered simultaneously.

for the set semantics. To see this consider the two mappings in Figure 16 and their corresponding merge in Figure 17 and compare these with the two mappings in Figure 14 and their set-based merge in Figure 15(b):

- In Figure 16, we are given two paths with overall weights 0.5 and 0.8; in Figure 17, these are combined with overall weight 1.3 assuming *sum* as the aggregation function.
- In Figure 14, we have two paths with overall weights 0.5 and 0.9. Again using *sum*, in Figure 15(b), these combine to a graph with overall weight of only 1.2.

In other words, as the above example shows, when set semantics is used to combine mappings, a pair of mappings with larger overall weight may lead to a merged graph with a smaller total weight. This violates monotonicity requirement of most common ranked join algorithms. On the other hand, it can be shown that a more lax monotonicity requirement (referred to as *sum-max monotonicity* in [40]) can still be shown to hold: Let r_1 and r_2 be two results and let R_1 and R_2 be the corresponding sets of sub-results that give r_1 and r_2 respectively; then, the following is true:

$$\left(\sum_{sr_i \in R_1} \text{cost}(sr_i) \leq \max_{sr_j \in R_2} (\text{cost}(sr_j)) \right) \rightarrow \text{cost}(r_1) \leq \text{cost}(r_2).$$

Therefore, in the case of set semantics, AR2Q relies on a modified version of the *horizon-based rank join* (HR-Join) algorithm presented in [40]. HR-Join is designed to handle ranked joined joins when sum-max monotonicity holds. Our extension of HR-Join allows the operator to (a) work with general structures not limited to twigs, (b) access its input streams more efficiently using a pull-based approach, (c) match two sub-results on one or more nodes not limited to a root node, (d) be fully compatible with other SPARQL algebra operators occurring in an algebra expression.

5.4 Join Ordering

In order to process a complex, multi-join SPARQL query, like many SPARQL engines, AR2Q creates a left-deep join plan, where the query patterns are added to the partial plan one at a time. Note that, in general, there are exponentially many ways to order the input pattern streams in a left-deep join plan, each with a different *getNext()* access cost and overall execution time. Thus selecting a good join plan requires the ability to predict which order will result in a smaller number of *getNext()* accesses. Since for different input patterns the result *inter-arrival times* (i.e., the time needed for the execution of a *getNext()* call) may be different, the differences in inter-arrival times may impact the total query execution time and may need to be considered in finding a join-order.

Existing SPARQL optimizers, such as [3, 44], collect and use statistics about the selectivities of the triple patterns. [44], for example, adds BGP edges to the partial execution plan in the non-decreasing order of selectivity. The optimization strategies presented in this subsection use two characteristics inherent to each pattern of a BGP: the result inter-arrival times for each query pattern and the *proximity* of a pattern to the other patterns in the query.

5.4.1 Join Orders based on Inter-Arrival Times

As we experimentally verify in Section 6, it is often better to perform a high number of accesses for those patterns with very low result inter-arrival times (i.e., access costs) and only a few accesses for those with high inter-arrival times. To see why, consider the left-deep plan in Figure 18: one single *getNext()* access to the left stream of *Op3* will produce a series of accesses to its input streams, which will in turn cause many accesses to the input streams of (*Op1*

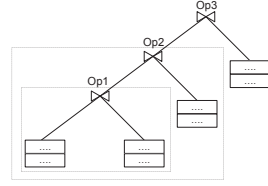


Figure 18: A sample left-deep join plan

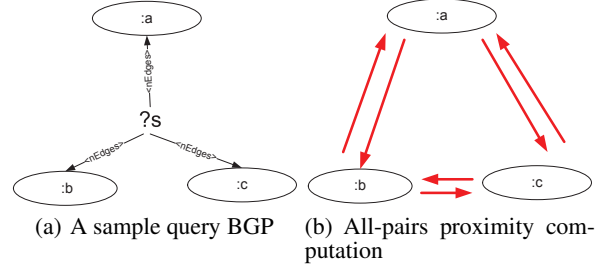


Figure 19: (a) A sample query BGP. The first step of proximity based join ordering involves computing the proximity score between all pairs of nodes in the query

and *Op2*). This implies that the two patterns under *Op1* tend to be accessed more frequently than those directly under *Op2* and *Op3*. Therefore, it is intuitive to try to place under *Op1* those patterns with the lowest access costs (or result inter-arrival times).

Note that result inter-arrival times (i.e., *getNext()* access costs) for path query patterns can either be computed through sampling or, as described earlier, be estimated based on the sizes of the reachability graphs involved in the path enumeration.

5.4.2 Join Orders based on Proximity Scores

As we discussed in Section 5.2.3, a node is said to have a *high proximity* to another, if the paths connecting the first node to the second in the graph are large in number, short in length, and have low weights [14, 46, 45]. A key intuition in this paper, which we will experimentally verify in Section 6, is that a query node which has high proximity to the other query nodes has a higher likelihood of producing many (due to the multitude of the paths) short (due to the low weights) matches with the remaining query nodes and therefore should be placed as left in the query plan tree as possible.

Therefore, as shown in Figure 19, given a query BGP, we first identify the proximity scores among all pairs of patterns in the query⁸ and then compute the overall proximity score for each pattern, p , as the *average* of the individual proximities between p and the other patterns in the BGP. Once a proximity score has been assigned to each pattern in the query, the join order is established by sorting the list of patterns in *non-increasing* order of proximity score and using this list to order patterns left to right on the left deep query plan tree. Since the proximity computation itself is a costly process, we use the proximity index structure described in Section 5.2.3 to identify proximities between the query patterns.

5.4.3 Hybrid Strategies for Join Ordering

To leverage the respective advantages of both inter-arrival time and proximity-based join ordering strategies, AR2Q uses hybrid strategies that combine the join ordering recommendations provided by these two approaches.

Direct Combination based Hybrid Strategies.

⁸Note that the proximity score is direction sensitive.

1. Compute the variances of the proximity scores and proceed if they are greater than the cut-off variance value for the dataset (otherwise, use only the inter-arrival times for join ordering)
2. Normalize all scores in each set by dividing each score by the largest score in the set; also subtract the normalized proximity scores from 1 to make the proximity order compatible with the inter-arrival time order
3. Compare the orders recommended by proximity-only and ascending inter-arrival time (starting with the right-most position) and check their differences; for each position (right-to-left) do
 - (a) If the current position at the two list contains different base patterns to be evaluated, use the recommendation given by inter-arrival time *only if* the normalized proximity score of the pattern is larger than a threshold; if the inter-arrival time order is selected, then
 - i. Find the index of the corresponding pattern in the recommendation list by inter-arrival times
 - ii. Reorder the elements in the proximity list based on the inter-arrival time based selection
4. Return the chosen mapping from the leaves of the join tree to the base query patterns

Figure 20: Pseudocode depicting the PVSH strategy

One way to achieve this is to combine the two ordering scores (i.e., inter-arrival time in one case and average proximity in the other) into a single score using an aggregation function, such as, min, max, average, or product. To render inter-arrival times and proximity scores compatible, each score is first normalized by dividing it with the highest inter-arrival time or proximity score for the query, respectively. Since the query planner needs to place to the further left positions in the join plan those patterns with lower result inter-arrival times but higher proximity scores, the normalized proximity scores are subtracted from 1.0 in order to establish a proximity-based order that is compatible to that established by inter-arrival times.

Proximity-Variance Sensitive Hybrid (PVSH) Strategies.

As we experimentally verify in the next section, reducing the number of accesses by considering proximity scores cannot provide great benefits if the differences among the proximity scores of the query patterns are not significant. In contrast, as the experiments will also show, when the proximity score variance is high, proximity-score based decisions become highly effective in reducing the number of `getNext()` accesses.

Therefore, AR2Q also uses a hybrid strategy, referred to as the *proximity-variance sensitive hybrid* (PVSH) strategy, which considers the proximity scores only if their variance is sufficiently large. Moreover, if the proximity scores are larger than a threshold, then they overwrite the inter-arrival time (or per-access cost) based recommendations. This is achieved as follows: Given the recommended left-to-right orders by inter-arrival time and proximity-based strategies, PVSH compares the lists starting with the right-most positions. The process is outlined in Figure 20.

6. EXPERIMENTS

In this section, we evaluate the performances of the query processing strategies by considering queries over two weighted knowledge bases. All experiments included in this section were run on an Intel Core i3-330M (2.13 GHz) with 4GB memory, running Windows 7.

6.1 Knowledge Bases

The first knowledge base we consider is the CoSeNa [18]; a weighted graph based on the bag of words from articles of the

New York Times. The nodes of this graph represent words and the weighted edges represent the co-occurrence of two words in the same article. For our experiments we used a subset graph containing 9K nodes and 24K edges.

Secondly, we use a knowledge base, with 10K nodes and 25K edges, containing author relationships based on a subset of the online DBLP database (<http://www.informatik.uni-trier.de/~ley/db/>) [33]. The nodes in this graph represent authors and the edges indicate the participation of two authors in the same publication. The weights of the edges in the knowledge base we constructed indicate the number of publications in which two authors co-participated. These values are normalized to the maximum number of shared publications included in the subset and subtracted from 1.0; i.e., the higher the relationship between two authors, the smaller is the edge weight.

For both knowledge bases the weights are converted cost by subtracting from 1.0. The proximity index structures for the knowledge bases used in this experiment were created off-line, in approximately three hours.

6.2 Evaluation Metric: Relative Distance from the Best Plan

Since join ordering is in general an NP-complete problem, the goal of optimization strategies is often to help avoid the worst plans. Of course, the closer to the execution time of the selected plan to that of the best plan, the better. Therefore, to assess the quality of a plan, we consider its relative distance from the best (and worst) plans; more specifically, the *relative distance* $RD(p)$ of the query plan, p , from the best plan is defined as

$$RD(p) = \frac{cost(p) - cost_{best}}{cost_{worst} - cost_{best}},$$

where $cost_{best}$ and $cost_{worst}$ are the costs (i.e., the number of accesses or the total execution time) for the best and the worst query plans and $cost(p)$ is the cost of the plan p . Note that the best plan will have a relative distance of 0.0 to the best plan, whereas the worst plan will have an RD value of 1.0.

Therefore, the smaller the RD score of a plan, the better it is.

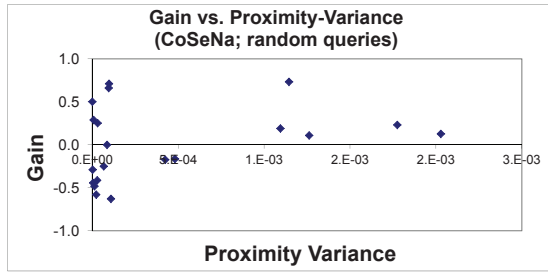
Since, obtaining the optimal plan is in general NP-complete, in practice, our goal is to have RD value as close to 0.0 as possible. Of course, in general, we would hope to perform better than a strategy which selected the plans randomly. Therefore, in addition to evaluating RD values for the query plans identified using alternative optimization strategies, we also consider the expected cost, RD_{rand} , of a randomly selected plan.

All the reported experiments were run for sets of 20 queries with different number of patterns (3, 4, and 5) and returning different numbers of top results (5 through 30). The queries are selected randomly, except when explicitly stated otherwise. Also, unless otherwise specified all queries run with 20 target number results.

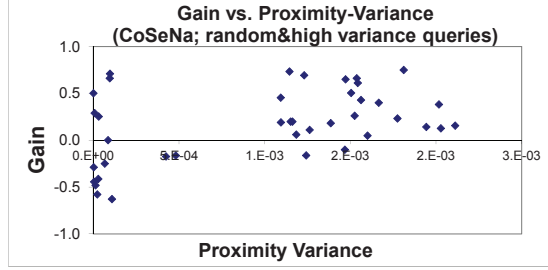
6.3 Impacts of the Proximity Scores

In this subsection, we first study the importance of the proximity scores in query optimization and the impact of their variances on the gain in the number of accesses provided by proximity-based query optimization. Here, we define gain as the reduction in the RD value provided by the proximity-based strategy relative to the expected cost, RD_{rand} , of a randomly selected plan. In other words, the better the optimized plan is, the higher the gain.

Figure 21(a) plots the gains obtained by proximity-based optimization against the proximity-variances for 20 random queries on the CoSeNa data set:



(a) 20 randomly selected queries



(b) 20 random queries + 20 high variance queries

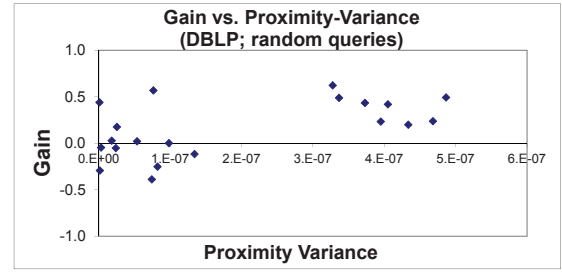
Figure 21: Gain provided by proximity vs. the variance in the proximity scores for the CoSeNa knowledge base

- For those queries with a very low variance in their proximity scores, the gains by the proximity-based strategy is essentially random; it can be positive (the selected plan is better than random) or negative (the selected plan is worse than random).
- On the other hand, for those queries with a high proximity variance, the gain provided by the proximity-based strategy is all better than random, with no occurrences of queries with negative gains.

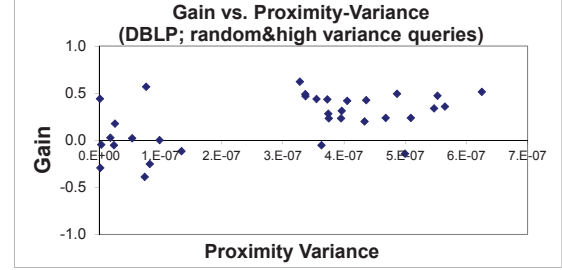
To confirm this observation, we extended the initial query sets with more queries specifically selected to have high proximity-variance: if our hypothesis is true, then we would expect that the new queries would tend to have positive gains. The results of this experiment are shown in Figure 21(b): As expected, almost all the new points in the figure, corresponding to the queries with high proximity variance, are optimized with positive gains. We further verified these results using the DBLP knowledge base and obtained very similar results (Figure 22).

The above observations are further verified by studying the number of `getNext()` accesses caused by the query plans. When we compare Figures 23(a) and 23(b) for the CoSeNa graph, we see that when focusing on queries with high variance in proximity scores, proximity-based optimization is able to reduce the number of accesses more effectively. Figures 23(c) and 23(d) verifies this for queries on the DBLP graph. These results confirm that proximity plays a role in optimization and the variance in the proximity scores is an important indicator of the benefits provided by the proximity-based strategy.

Note however that proximity-only optimization may not be effective when the result inter-arrival times (i.e., per-access costs) of the query patterns are highly heterogeneous. Figure 24 illustrates this experimentally: here the two curves are for the average and proximity-based optimized plans for queries, with heterogeneous inter-arrival times (varying between ~ 0 –indicating cached

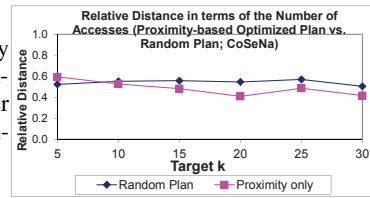


(a) 20 randomly selected queries

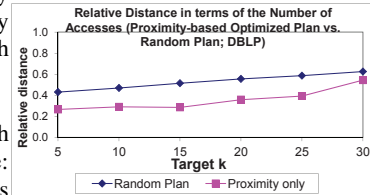


(b) 20 random queries + 20 high variance queries

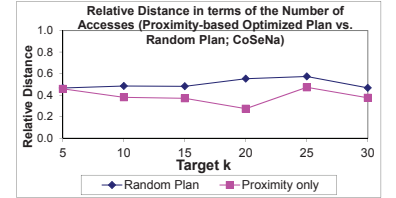
Figure 22: Gain provided by proximity vs. the variance in the proximity scores for the DBLP knowledge base



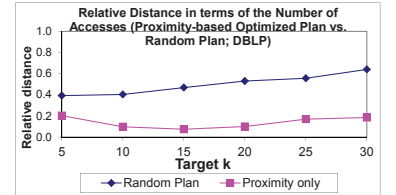
(a) random queries (CoSeNa)



(c) random queries (DBLP)



(b) high- variance queries (CoSeNa)



(d) high- variance queries (DBLP)

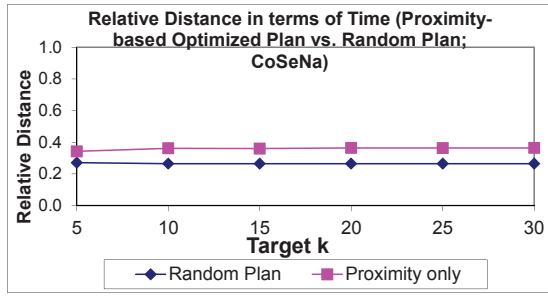
Figure 23: Relative distances (in terms of accesses) for the average and the proximity-based optimized plans

results– and 10 ms), over the CoSeNa and DBLP graphs. As Figure 24(a) shows, the heterogeneity of the inter-arrival times in the query patterns may sometime negatively effect the results provided by the proximity-only optimization strategy.

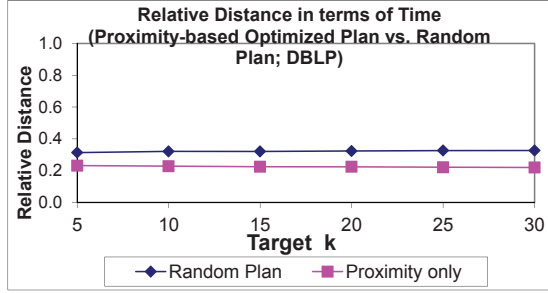
6.4 Impact of the Result Inter-Arrival Times

In this subsection, we study the impact of the result inter-arrival times of the query patterns on query plan selection. As before we consider 20 random queries with 4 patterns. The inter-arrival times of the patterns varied from ~ 0 ms to 10ms.

Figures 25(a) and 25(b) show the relative distances (in terms of number of accesses and the total execution time, respectively) of the inter-arrival time based optimized plans. Figure 25(a) shows that, as predicted by the discussion in Section 5.4.1, the strategy based on ascending order of inter-arrival times provides a plan close to the optimal plan in terms of the total execution time, whereas the opposite of this strategy (i.e., the strategy based on descending or-



(a) CoSeNa knowledge base



(b) DBLP knowledge base

Figure 24: Relative distances (in terms of time) for the average and the proximity-based optimized plans for queries with heterogeneous inter-arrival times

(a) Relative Distance in terms of Number of Accesses (Cosena)								
top-k	Rand.	Prox.	Inter-arr.	Prod.	Avg.	Max	Min	PVSH
5	0.52	0.59	0.45	0.39	0.41	0.45	0.39	0.43
10	0.55	0.53	0.59	0.55	0.56	0.58	0.55	0.57
15	0.56	0.48	0.55	0.56	0.47	0.55	0.56	0.51
20	0.55	0.41	0.49	0.53	0.43	0.49	0.53	0.48
25	0.57	0.49	0.56	0.51	0.51	0.56	0.51	0.54
30	0.50	0.42	0.49	0.44	0.44	0.49	0.44	0.49
avg	0.54	0.49	0.52	0.50	0.47	0.52	0.50	0.50
(b) Relative Distance in terms of Exec. Time (Cosena)								
top-k	Rand.	Prox.	Inter-arr.	Prod.	Avg.	Max	Min	PVSH
5	0.27	0.34	0.08	0.1	0.15	0.09	0.10	0.07
10	0.26	0.36	0.09	0.12	0.18	0.09	0.12	0.07
15	0.26	0.36	0.09	0.13	0.18	0.09	0.13	0.08
20	0.26	0.36	0.09	0.13	0.18	0.09	0.13	0.08
25	0.26	0.36	0.09	0.13	0.19	0.09	0.13	0.08
30	0.26	0.36	0.09	0.13	0.19	0.09	0.13	0.08
avg	0.26	0.36	0.09	0.12	0.18	0.09	0.12	0.07

Table 1: Comparison of the optimization strategies in terms of (a) the number of accesses and (b) execution times (for the CoSeNa knowledge base; queries joining 4 patterns)

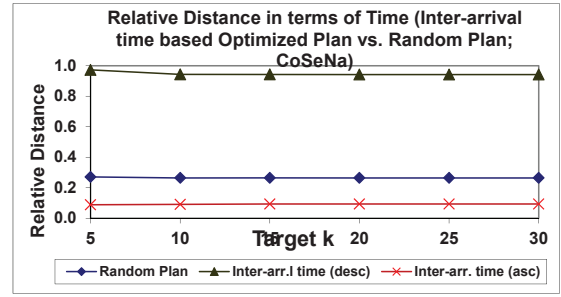
der of inter-arrival times) provides a plan close to worse plan. This confirms that, if used properly, inter-arrival time based strategy can be very effective in optimizing the overall query execution time.

On the other hand, as Figure 25(b) shows, the inter-arrival based strategy has almost no positive or negative impact on reducing the number of accesses. This suggests that using inter-arrival time based strategy along with a proximity-based strategy (which would help reduce the number of accesses) may help further reduce the overall execution time. best approach.

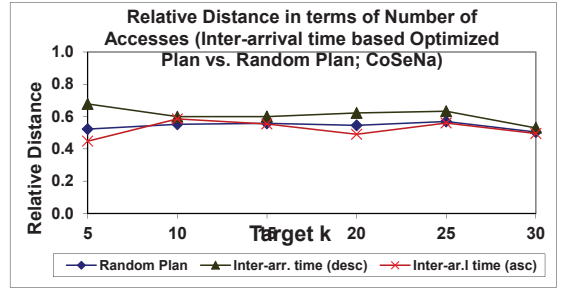
Figure 26 confirms the above observations for the DBLP knowledge base.

6.5 Impact of Hybrid Optimization Strategies

In this subsection, we compare the hybrid strategies discussed in Section 5.4.3 to the proximity-score only and inter-arrival time only strategies. As before, we consider 20 random queries each



(a) Relative dist. in terms of execution time



(b) Relative dist. in terms of number of accesses

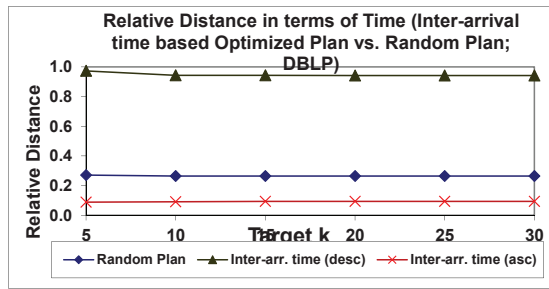
Figure 25: Relative distances (in terms of (a) the number of accesses and (b) the execution time) for the average and the inter-arrival time based optimized plans (for the CoSeNa knowledge base); the figure considers both descending and ascending inter-arrival time orders

(a) Relative Distance in terms of Number of Accesses (DBLP)								
top-k	Rand.	Prox.	Inter-arr.	Prod.	Avg.	Max	Min	PVSH
5	0.43	0.27	0.42	0.48	0.46	0.46	0.40	0.42
10	0.47	0.29	0.48	0.53	0.52	0.53	0.55	0.48
15	0.51	0.28	0.49	0.54	0.53	0.54	0.57	0.49
20	0.56	0.36	0.49	0.50	0.53	0.54	0.52	0.49
25	0.59	0.39	0.48	0.52	0.52	0.53	0.53	0.48
30	0.62	0.54	0.61	0.74	0.66	0.65	0.56	0.49
avg	0.53	0.36	0.49	0.55	0.53	0.54	0.56	0.49
(b) Relative Distance in terms of Exec. Time (DBLP)								
top-k	Rand.	Prox.	Inter-arr.	Prod.	Avg.	Max	Min	PVSH
5	0.31	0.23	0.05	0.07	0.05	0.05	0.16	0.04
10	0.32	0.23	0.05	0.08	0.05	0.05	0.17	0.04
15	0.32	0.22	0.05	0.08	0.05	0.05	0.17	0.04
20	0.32	0.22	0.05	0.08	0.05	0.05	0.17	0.05
25	0.33	0.22	0.05	0.08	0.05	0.05	0.17	0.04
30	0.33	0.22	0.05	0.08	0.05	0.05	0.18	0.04
avg	0.32	0.22	0.05	0.08	0.05	0.05	0.17	0.04

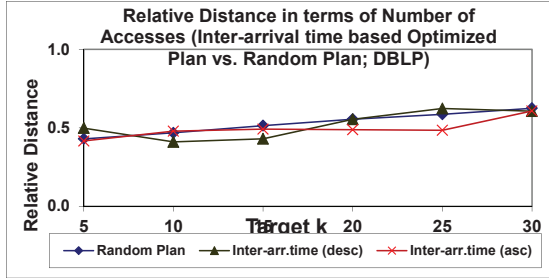
Table 2: Comparison of the optimization strategies in terms of (a) the number of accesses and (b) execution times (for the DBLP knowledge base; queries joining 4 patterns)

with 4 patterns. The inter-arrival times of the patterns varied from ~ 0 ms to 10ms. The variance cut-off for the PVSH approach are selected as 10^{-3} for the CoSeNa knowledge base, and 2×10^{-7} for the DBLP knowledge base. Similarly, the proximity thresholds to determine the use of the recommendation given by inter-arrival time are selected as 3×10^{-1} and 10^{-7} for the CoSeNa and DBLP knowledge bases respectively. To present the results more precisely, we present the relative distance results in a tabular form in Tables 1 and 2, for CoSeNa and DBLP knowledge bases, respectively. The last row in each table averages the relative distance values obtained for different target number of results.

When studying these tables, we can see that, in terms of the number of accesses, the hybrid approaches are not always the best



(a) Relative dist. in terms of execution time



(b) Relative dist. in terms of number of accesses

Figure 26: Relative distances (in terms of (a) the number of accesses and (b) the execution time) for the average and the inter-arrival time based optimized plans (for DBLP)

and pure proximity-based approach is always competitive, if not the best. In terms of the execution time, however, the proposed *proximity-variance sensitive hybrid* (PVSH) approach always provides the best relative distance scores among all the alternatives.

7. CONCLUSIONS

In this paper, we presented the R²DF for weighted RDF knowledge bases. As part of this framework, we first introduced R2DF, a simple extension to the RDF specification that allows RDF triples to include weight values. Since current RDF query languages would fail to make use of this proposed extension, we next presented SPARankQL, an extension of the SPARQL specification to support, among other features, (a) new path predicates to express advanced relationships (reachability) between nodes and (b) the ability to express ranked queries. In order to support efficient execution of SPARankQL queries, we also presented indexing, ranked query processing, and optimization strategies. Experiments showed that the so-called *proximity-based* optimization reduces the number of `getNext()` calls to the base pattern matching operators significantly when the variance in the proximity scores of the patterns in the query is large. Experiments also showed that a result inter-arrival time (or per-access cost) based strategy, which considers the differences among the costs of evaluating query patterns, is able to reduce the overall execution time very effectively, providing close to optimal query plans. This further motivated a *proximity-variance sensitive hybrid* strategy which combines the two optimization strategies to push the overall execution time further down.

8. REFERENCES

- [1] Arq - a sparql processor for jena.
- [2] Jena - a semantic web framework for java. <http://jena.sourceforge.net/>.
- [3] Tdb - jena semantic fram. <http://openjena.org/wiki/TDB>.
- [4] Rdf semantics, February 2004.
- [5] RDF/XML syntax specification (Revised). Tech. report, W3C, February 2004.
- [6] Turtle - terse RDF triple language, W3C team submission, 2008.
- [7] W3C xml path language. In *Encyclopedia of Database Systems*, 2009.
- [8] K. Anyanwu, A. Maduko, and A. P. Sheth. Sparql2l: towards support for subgraph extraction queries in rdf databases. In *WWW*, pages 797–806, 2007.
- [9] K. Anyanwu and A. Sheth. The p operator: Discovering and ranking associations on the semantic web. *SIGMOD Record*, 31, 2002.
- [10] K. Anyanwu and A. Sheth. p-queries: enabling querying for semantic associations on the semantic web. In *WWW*, pages 690 – 699, 2003.
- [11] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *VLDB*, pages 914–925, 2007.
- [12] S. Barton. Designing indexing structure for discovering relationships in rdf graphs. In *DATES*, pages 7–17, 2004.
- [13] T. Berners-Lee and D. Connolly. Notation3 (n3): A readable rdf syntax. Technical report, W3C, January 2008.
- [14] K. S. Candan and W.-S. Li. Using random walks for mining web document associations. In *PAKDD*, pp. 294–305, 2000.
- [15] I. Cantador, et al. A multi-purpose ontology-based approach for personalised content filtering and retrieval. *Advances in Semantic Media Adaptation and Personalization*, 2008.
- [16] M. J. Carey and D. Kossmann. Processing top n and bottom n queries. *IEEE Data Eng. Bull.*, 20(3):12–19, 1997.
- [17] J. Carroll et al. Jena: Implementing the semantic web recommendations. Technical report, HP Laboratories, 2003.
- [18] M. Cataldi, C. Schifanella, K. S. Candan, M. L. Sapino, and L. Di Caro. Cosenza: a context-based search and navigation system. In *MEDES* 2009.
- [19] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects by exploiting relationships: Computing top-k over aggregation. In *SIGMOD*, 2006.
- [20] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE TKDE*, 16(8):992–1009, 2004.
- [21] J. Cheng, J. X. Yu, and N. Tang. Fast reachability query processing. In *DASFAA*, pages 674–688, 2006.
- [22] E. de Queiros Vieira Martins and M. M. B. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR*, 1(2):121–133, 2003.
- [23] R. Fagin. Fuzzy queries in multimedia database systems. In *PODS*, 1998.
- [24] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [25] U. Guntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.
- [26] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, pages 581–594, 2008.
- [27] O. Hartig and R. Heese. The sparql query graph model for query optimization. In *ESWC* 2007.
- [28] O. Hartig. Querying trust in RDF Data with tSPARQL. In *ESWC*, 2009.
- [29] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.
- [30] J. W. Kim and K. S. Candan. Skip-and-prune: cosine-based top-k query processing for efficient context-sensitive document retrieval. In *SIGMOD*, 2009.
- [31] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation, Feb. 2004.
- [32] K. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *ESWC*, pages 145–159, 2007.
- [33] M. Ley. The dblp computer science bibliography: Evolution, research issues, perspectives. In *SPIRE*, pages 1–10, 2002.
- [34] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.
- [35] C. Li, M. A. Soliman, K. C.-C. Chang, and I. F. Ilyas. Ranksql: Supporting ranking queries in relational database management systems. In *VLDB*, 2005.
- [36] openRDF.org. Sesame, 2007.
- [37] Msulgara Project. Mulgara semantic store.
- [38] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.
- [39] Y. Qi, K. S. Candan, and M. L. Sapino. Ficsr: feedback-based inconsistency resolution and query processing on misaligned data sources. In *SIGMOD*, 2007.
- [40] Y. Qi, K. S. Candan, and M. L. Sapino. Sum-max monotonic ranked joins for evaluating top-k twig queries on weighted data graphs. In *VLDB*, 2007.
- [41] M.A. Rodriguez, and P. Neubauer. A path algebra for multi-relational graphs. In *Int. Workshop on Graph Data Mgmt. (GDM)* 2011.
- [42] A. Seaborne. Rdfql – a query language for rdf. W3C Member Submission, 2004.
- [43] H. H. Song, et al. Scalable proximity estimation and link prediction in online social networks. In *Internet Measurement Conference*, pages 322–335, 2009.
- [44] M. Stocker et al. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW* 2008.
- [45] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD* pages 404–413, 2006.
- [46] H. Tong, C. Faloutsos, and Y. Koren. Fast direction-aware proximity for graph mining. In *KDD*, pages 747–756, 2007.
- [47] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient rdf storage and retrieval in jena2. Technical Report HPL-2003-266, December 2003.
- [48] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.