

Index Structures and Algorithms for Querying Distributed RDF Repositories

Heiner Stuckenschmidt
Vrije Universiteit Amsterdam
The Netherlands
heiner@cs.vu.nl

Richard Vdovjak
Geert-Jan Houben
Eindhoven University of
Technology
The Netherlands
richardv@win.tue.nl
g.j.houben@tue.nl

Jeen Broekstra
Aduna B.V. Amerfoort
(formerly: Aidministrato)
The Netherlands
jeen.broekstra@aduna.biz

ABSTRACT

A technical infrastructure for storing, querying and managing RDF data is a key element in the current semantic web development. Systems like Jena, Sesame or the ICS-FORTH RDF Suite are widely used for building semantic web applications. Currently, none of these systems supports the integrated querying of distributed RDF repositories. We consider this a major shortcoming since the semantic web is distributed by nature. In this paper we present an architecture for querying distributed RDF repositories by extending the existing Sesame system. We discuss the implications of our architecture and propose an index structure as well as algorithms for query processing and optimization in such a distributed context.

Categories and Subject Descriptors

E.1 [Data]: DATA STRUCTURES—*Distributed Data Structures*;
H.2.4 [Information Systems]: DATABASE MANAGEMENT SYSTEMS—*Distributed Databases, Query Processing*

General Terms

Algorithms, Performance, Design

Keywords

RDF Querying, Index Structures, Optimization

1. MOTIVATION

The need for handling multiple sources of knowledge and information is quite obvious in the context of semantic web applications. First of all we have the duality of schema and information content where multiple information sources can adhere to the same schema. Further, the re-use, extension and combination of multiple schema files is considered to be common practice on the semantic web [7]. Despite the inherently distributed nature of the semantic web, most current RDF infrastructures (for example [4]) store information locally as a single knowledge repository, i.e., RDF models from remote sources are replicated locally and merged into a single model. Distribution is virtually retained through the use of namespaces to distinguish between different models. We argue that many interesting applications on the semantic web would benefit from or even require an RDF infrastructure that supports real distribution of information sources that can be accessed from a single point. Beyond

Copyright is held by the author/owner(s).
WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

the argument of conceptual adequacy, there are a number of technical reasons for real distribution in the spirit of distributed databases:

Freshness: The commonly used approach of using a local copy of a remote source suffers from the problem of changing information. Directly using the remote source frees us from the need of managing change as we are always working with the original.

Flexibility: Keeping different sources separate from each other provides us with a greater flexibility concerning the addition and removal of sources. In the distributed setting, we only have to adjust the corresponding system parameters.

In many cases, it will even be unavoidable to adopt a distributed architecture, for example in scenarios in which the data is not owned by the person querying it. In this case, it will often not be permitted to copy the data. More and more information providers, however, create interfaces that can be used to query the information. The same holds for cases where the information sources are too large to just create a single model containing all the information, but they still can be queried using a special interface (Musicbrainz is an example of this case). Further, we might want to include sources that are not available in RDF, but that can be wrapped to produce query results in RDF format. A typical example is the use of a free-text index as one source of information. Sometimes there is not even a fixed model that could be stored in RDF, because the result of a query is only calculated at runtime (Google, for instance, provides a programming interface that could be wrapped into an RDF source). In all these scenarios, we are forced to access external information sources from an RDF infrastructure without being able to create a local copy of the information we want to query. On the semantic web, we almost always want to combine such external sources with each other and with additional schema knowledge. This confirms the need to consider an RDF infrastructure that deals with information sources that are actually distributed across different locations.

In this paper, we address the problem of integrated access to distributed RDF repositories from a practical point of view. In particular, starting from a real-life use case where we are considering a number of distributed sources that contain research results in the form of publications, we take the existing RDF storage and retrieval system Sesame and describe how the architecture and the query processing methods of the system have to be extended in order to move to a distributed setting.

The paper is structured as follows. In Section 2 we present an extension of the Sesame architecture to multiple, distributed repositories and discuss basic assumptions and implications of the architecture. Section 3 presents source index hierarchies as suitable mechanisms to support the localization of relevant data during query processing. In Section 4 we introduce a cost model for processing queries in the distributed architecture, and show its use in optimizing query execution as a basis for the two-phase optimization heuristics for join ordering. Section 5 reviews previous work on index structures for object-oriented data bases. It also summarizes related work on query optimization particularly focusing on the join ordering problem. We conclude with a discussion of open problems and future work.

2. INTEGRATION ARCHITECTURE

Before discussing the technical aspects of distributed data and knowledge access, we need to put our work in context by introducing the specific integration architecture we have to deal with. This architecture limits the possible ways of accessing and processing data, and thereby provides a basis for defining some requirements for our approach. It is important to note that our work is based on an existing RDF storage and retrieval system, which more or less predefines the architectural choices we made. In this section, we describe an extension of the Sesame system [4] to distributed data sources.

The Sesame architecture is flexible enough to allow a straightforward extension to a setting where we have to deal with multiple distributed RDF repositories. In the current setting, queries, expressed in Sesame’s query language SeRQL, are directly passed from the query engine to an RDF API (SAIL) that abstracts from the specific implementation of the repository. In the distributed setting, we have several repositories that can be implemented in different ways. In order to abstract from this technical heterogeneity, it is useful to introduce RDF API implementations on top of each repository, making them accessible in the same way.

The specific problem of a distributed architecture is now that information relevant to a query might be distributed over the different sources. This requires to locate relevant information, retrieve it, and combine the individual answers. For this purpose, we introduce a new component between the query parser and the actual SAILs - the mediator SAIL (see Figure 1).

In this work, we assume that local repositories are implemented using database systems that translate queries posed to the RDF API into SQL queries and use the database functionality to evaluate them (compare [5]). This assumption has an important influence on the design of the distributed query processing: the database engines underlying the individual repositories have the opportunity to perform local optimization on the SQL queries they pose to the data. Therefore we do not have to perform optimizations on sub-queries that are to be forwarded to a single source, because the repository will deal with it. Our task is rather to determine which part of the overall query has to be sent to which repository.

In the remainder of this paper, we describe an approach for querying distributed RDF sources that addresses these requirements implied by the adopted architecture. We focus our attention on index structures and algorithms implemented in the mediator SAIL.

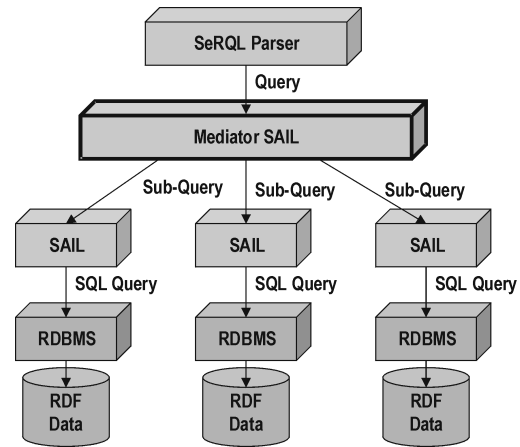


Figure 1: Distribution Architecture.

3. INDEX STRUCTURES

As discussed above, in order to be able to make use of the optimization mechanisms of the database engines underlying the different repositories, we have to forward entire queries to the different repositories. In the case of multiple external models, we can further speed up the process by only pushing down queries to information sources we can expect to contain an answer. The ultimate goal is to push down to a repository exactly that part of a more complex query for which a repository contains an answer. This part can range from a single statement template to the entire query. We can have a situation where a subset of the query result can directly be extracted from one source, and the rest has to be extracted and combined from different sources. This situation is illustrated in the following example.

EXAMPLE 1. Consider the case where we want to extract information about research results. This information is scattered across a variety of data sources containing information about publications, projects, patents, etc. In order to access these sources in a uniform way, we use the OntoWeb research ontology. Figure 2 shows parts of this ontology.

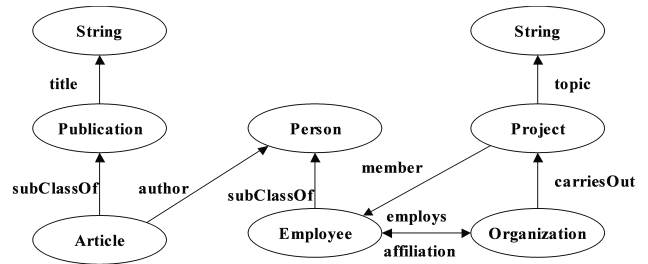


Figure 2: Part of the OntoWeb Ontology.

Suppose we now want to ask for the titles of articles by employees of organizations that have projects in the area “RDF”. The path expression of a corresponding SeRQL query would be the following¹:

¹For the sake of readability we omit namespaces whenever they do not play a technical role.

```
{A} title {T};
  author {W} affiliation {O}
  carriesOut {P} topic {'RDF'}
```

Now, let's assume that we have three information sources S_1 , S_2 , and S_3 . S_1 is a publication data base that contains information about articles, titles, authors and their affiliations. S_2 is a project data base with information about industrial projects, topics, and organizations. Finally, S_3 is a research portal that contains all of the above information for academic research.

If we want to answer the query above completely we need all three information sources. By pushing down the entire query to S_3 we get results for academic research. In order to also retrieve the information for industrial research, we need to split up the query, push the fragment

```
{A} title {T};
  author {W} affiliation {O}

to S1, the fragment

{O} carriesOut {P} topic {'RDF'}
```

to S_2 , and join the result based on the identity of the organization.

The example illustrates the need for sophisticated indexing structures for deciding which part of a query to direct to which information source. On the one hand we need to index complex query patterns in order to be able to push down larger queries to a source; on the other hand we also need to be able to identify sub-queries needed for retrieving partial results from individual sources.

In order to solve this problem we build upon existing work on indexing complex object models using join indices [14]. The idea of join indices is to create additional database tables that explicitly contain the result of a join over a specific property. At runtime, rather than computing a join, the system just accesses the join index relation which is less computationally expensive. The idea of join indices has been adapted to deal with complex object models. The resulting index structure is a join index hierarchy [21]. The most general element in the hierarchy is an index table for elements connected by a certain path $p_{0..n-1}$ of length n . Every following level contains all the paths of a particular length from 2 paths of length $n-1$ at the second level of the hierarchy to n paths of length 1 at the bottom of the hierarchy. In the following, we show how the notion of join index hierarchies can be adapted to deal with the problem of determining information sources that contain results for a particular sub-query.

3.1 Source Index Hierarchies

The majority of work in the area of object oriented databases is focused on indexing schema-based paths in complex object models. We can make use of this work by relating it to the graph-based interpretation of RDF models. More specifically, every RDF model can be seen as a graph where nodes correspond to resources and edges to properties linking these resources. The result of a query to such a model is a set of subgraphs corresponding to a path expression. While a path expression does not necessarily describe a single path, it describes a tree that can be created by joining a set of paths. Making use of this fact, we first decompose the path expression into a set of expressions describing simple paths, then forward the simpler path expressions to sources that contain the corresponding information using a path-based index structure, and join retrieved answers to create the result.

The problem with using path indices to select information sources is the fact that the information that makes up a path might be distributed across different information sources (compare Example 1). We therefore have to use an index structure that also contains information about sub-paths without losing the advantage of indexing complete paths. An index structure that combines these two characteristics is the join index hierarchy proposed in [21]. We therefore take their approach as a basis for defining a *source index hierarchy*.

DEFINITION 1 (SCHEMA PATH). Let $G = \langle V, E, L, s, t, l \rangle$ be a labelled graph of an RDF model where V is a set of nodes, E a set of edges, L a set of labels, $s, t : E \rightarrow V$ and $l : E \rightarrow L$.

For every $e \in E$, we have $s(e) = r_1$, $t(e) = r_2$ and $l(e) = l_e$ if and only if the model contains the triple (r_1, l_e, r_2) . A path in G is a list of edges e_0, \dots, e_{n-1} such that $t(e_i) = s(e_{i+1})$ for all $i = 0, \dots, n-2$. Let $p = e_0, \dots, e_{n-1}$ be a path, the corresponding schema path is the list of labels l_0, \dots, l_{n-1} such that $l_i = l(e_i)$.

The definition establishes the notion of a path for RDF models. We can now use path-based index structures and adapt them to the task of locating path instances in different RDF models. The basic structure we use for this purpose is an index table of sources that contain instances of a certain path.

DEFINITION 2 (SOURCE INDEX). Let p be a schema path; a source index for p is a set of pairs (s_k, n_k) where s_k is an information source (in particular, an RDF model) and the graph of s_k contains exactly n_k paths with schema path p and $n_k > 0$.

A source index can be used to determine information sources that contain instances of a particular schema path. If our query contains the path p , the corresponding source index provides us with a list of information sources we have to forward the query to in order to get results. The information about the number of instance paths can be used to estimate communication costs and will be used for join ordering (see Section 4). So far the index satisfies the requirement of being able to list complete paths and push down the corresponding queries to external sources. In order to be able to retrieve information that is distributed across different sources, we have to extend the structure based on the idea of a hierarchy of indices for arbitrary sub-paths. The corresponding structure is defined as follows.

DEFINITION 3 (SOURCE INDEX HIERARCHY). Let $p = l_0, \dots, l_{n-1}$ be a schema path. A source index hierarchy for p is an n -tuple $\langle P_n, \dots, P_1 \rangle$ where

- P_n is a source index for p
- P_i is the set of all source indices for sub-paths of p with length i that have at least one entry.

The most suitable way to represent such index structure is a hierarchy, where the source index of the indexed path is the root element. The hierarchy is formed in such a way that the subpart rooted at the source index for a path p always contains source indices for all sub-paths of p . This property will later be used in the query answering algorithm. Forming a lattice of source indices, a source index hierarchy contains information about every possible schema sub-path. Therefore we can locate all fragments of paths that might be combined into a query result. At the same time, we can first concentrate on complete path instances and successively investigate smaller fragments using the knowledge about the existence of longer paths. We illustrate this principle in the following example.

EXAMPLE 2. Let us reconsider the situation in Example 1. The schema path we want to index is given by the list (author, affiliation, carriesOut, topic). The source index hierarchy for this path therefore contains source indices for the paths

- $p_{0..3} : (author, affiliation, carriesOut, topic)$
- $p_{0..2} : (author, affiliation, carriesOut),$
 $p_{1..3} : (affiliation, carriesOut, topic)$
- $p_{0..1} : (author, affiliation),$
 $p_{1..2} : (affiliation, carriesOut),$
 $p_{2..3} : (carriesOut, topic)$
- $p_0 : (author),$
 $p_1 : (affiliation),$
 $p_2 : (carriesOut),$
 $p_3 : (topic)$

Starting from the longest path, we compare our query expression with the index (see Figure 3 for an example of index contents). We immediately get the information that S_3 contains results. Turning to sub-paths, we also find out that S_1 contains results for the sub-path (author, affiliation) and S_2 for the sub-path (carriesOut, topic) that we can join in order to compute results, because together both sub-paths make up the path we are looking for.

The source indices also contain information about the fact that S_3 contains results for all sub-paths of our target path. We still have to take this information into account, because in combination with fragments from other sources we might get additional results. However, we do not have to consider joining sub-paths from the same source, because these results are already covered by longer paths. In the example we see that S_2 will return far less results than S_1 (because there are less projects than publications). We can use this information to optimize the process of joining results.

A key issue connected with indexing information sources is the trade-off between required storage space and computational properties of index-based query processing. Compared to index structures used to speed up query processing within an information source, a source index is relatively small as it does not encode information about individual elements in a source. Therefore, the size of the index is independent of the size of the indexed information sources. The relevant parameters in our case are the number of sources s and the lengths of the schema path n . More specifically, in the worst case a source index hierarchy contains source indices for every sub-path of the indexed schema path. As the number of all sub-path of a path is $\sum_{i=1}^n i$, the worst-case² space complexity of a source index hierarchy is $O(s \cdot n^2)$. We conclude that the length of the indexed path is the significant parameter here.

3.2 Query Answering Algorithm

Using the notion of a source index hierarchy, we can now define a basic algorithm for answering queries using multiple sources of information. The task of this algorithm is to determine all possible combinations of sub-paths of the given query path. For each of these combinations, it then has to determine the sources containing results for the path fragments, retrieve these results, and join them into a result for the complete path. The main task is to guarantee that we indeed check all possible combinations of sub-paths for the

²It is the case where all sources contain results for the complete schema path.

query path. The easiest way of guaranteeing this is to use a simple tree-recursion algorithm that retrieves results for the complete path, then splits the original path, and joins the results of recursive calls for the sub-paths. In order to capture all possible splits this has to be done for every possible split point in the original path. The corresponding semi-formal algorithm is given below (Algorithm 1).

Algorithm 1 Compute Answers.

Require: A schema path $p = l_0, \dots, l_{n-1}$

Require: A source index hierarchy $h = (P_n, \dots, P_1)$ for p

for all sources s_k in source index P_n **do**

ANSWERS := instances of schema path p in source s_k

RESULT := result \cup answers

end for

if $n \geq 2$ **then**

for all $i = 1 \dots n - 1$ **do**

$p_{0..i-1} := l_0, \dots, l_{i-1}$

$p_{i..n-1} := l_i, \dots, l_{n-1}$

$h_{0..i-1} :=$ Sub-hierarchy of h rooted at the source index for

$p_{0..i-1}$

$h_{i..n-1} :=$ Sub-hierarchy of h rooted at the source index for

$p_{i..n-1}$

$res_1 := \text{ComputeAnswers}(p_{0..i-1}, h_{0..i-1})$

$res_2 := \text{ComputeAnswers}(p_{i..n-1}, h_{i..n-1})$

RESULT := result \cup join(res_1, res_2)

end for

end if

return result

Note that Algorithm 1 is far from being optimal with respect to runtime performance. The straightforward recursion scheme does not take specific actions to prevent unnecessary work and it neither selects an optimal order for joining sub-paths. We can improve this situation by using knowledge about the information in the different sources and performing query optimization.

4. QUERY OPTIMIZATION

In the previous section we described a light-weight index structure for distributed RDF querying. Its main task is to index schema paths w.r.t. underlying sources that contain them. Compared to instance-level indexing, our approach does not require creating and maintaining oversized indices since there are far fewer sources than there are instances. Instance indexing would not scale in the web environment and as mentioned above in many cases it would not even be applicable, e.g., when sources do not allow replication of their data (which is what instance indices essentially do). The downside of our approach, however, is that query answering without the index support at the instance level is much more computationally intensive. Moreover, in the context of semantic web portal applications the queries are not man-entered anymore but rather generated by a portal's front-end (triggered by the user) and often exceed the size³ which can be easily computed by using brute force. Therefore we focus in this section on query optimization as an important part of a distributed RDF query system. We try to avoid re-inventing the wheel and once again seek for inspiration in the database field, making it applicable by "relationizing" the RDF model.

Each single schema path p_i of length 1 (also called 1-*path*) can be perceived as a relation with two attributes: the source vertex

³Especially, the length of the path expression.

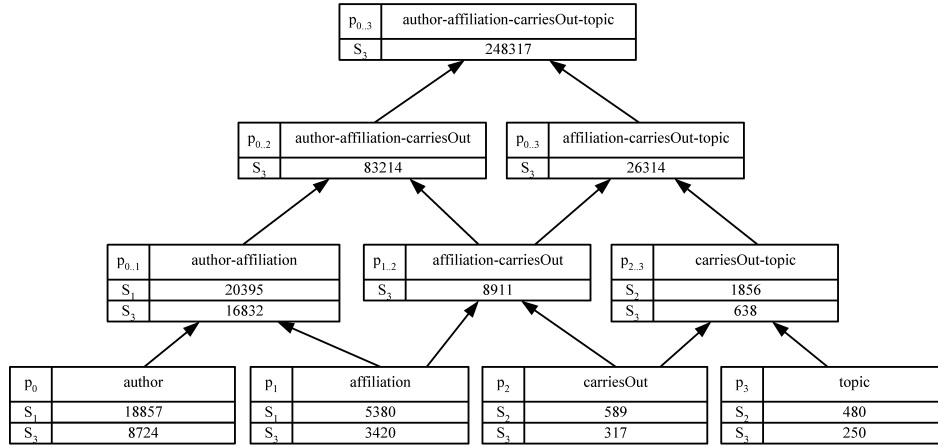


Figure 3: Source index hierarchy for the given query path.

$s(p_i)$ and the target vertex $t(p_i)$. A schema path of length more than 1 is modelled as a set of relations joined together by the identity of the adjacent vertices, essentially representing a chain query of joins as defined in Definition 4. This relational view over an RDF graph offers the possibility to re-use the extensive research on join optimization in databases, e.g. [1, 8, 9, 17, 20].

Taking into account the (distributed) RDF context of the join ordering problem there are several specifics to note when devising a good query plan. As in distributed databases, communication costs significantly contribute to the overall cost of a query plan. Since in our case the distribution is assumed to be realized via an IP network with a variable bandwidth, the communications costs are likely to contribute substantially to the overall processing costs, which makes the minimization of data transmission across the network very important. Unless the underlying sources provide join capabilities, the data transmission cannot be largely reduced: all (selected) bits of data from the sources are joined by the mediator and hence must be transmitted via the network.

There may exist different dependencies (both structural and extensional) on the way the data is distributed. If the information about such dependencies is available, it essentially enables the optimizer to prune join combinations which cannot yield any results. The existence of such dependencies can be (to some extent) computed/discovered prior to querying, during the initial integration phase. Human insight is, however, often needed in order to avoid false dependency conclusions, which could potentially influence the completeness of query answering.

The performance and data statistics are both necessary for the optimizer to make the right decision. In general, the more the optimizer knows about the underlying sources and data, the better optimized the query plan is. However, taking into account the autonomy of the sources, the necessary statistics do not have to be always available. We design our mediator to cope with incomplete statistical information in such a way that the missing parameters are estimated as being worse than those that are known (pessimistic approach). Naturally, the performance of the optimizer is then lower but it increases steadily when the estimations are made more realistic based on the actual response from the underlying sources; this is also known as optimizer calibration.

As indicated above, the computational capabilities of the underlying sources may vary considerably. We distinguish between those sources that can only retrieve the selected local data (pull up strategy) and those that can perform joins of their local and incoming external data (push down strategy), thus offering computational services that could be used to achieve both a higher degree of parallelism and smaller data transmission over the network, e.g., by applying semi-join reductions [1]. At present, however, most sources are capable only of selecting the desired data within their extent, i.e., they do not offer the join capability. Therefore, further we focus mainly on local optimization at the mediator's side.

For this purpose we need to perceive an RDF model as a set of relations on which we can apply optimization results from the area of relational databases. In this context the problem of join ordering arises, when we want to compute the results for schema paths from partial results obtained from different sources. Creating the result for a schema corresponds to the problem of computing the result of a chain query as defined below:

DEFINITION 4 (CHAIN QUERY). *Let p be a schema path composed from the 1-paths p_1, \dots, p_n . The chain query of p is the n -join $p_1 \bowtie_{t(p_1)=s(p_2)} p_2 \bowtie_{t(p_2)=s(p_3)} p_3 \bowtie \dots p_n$, where $s(p_i)$ and $t(p_i)$ are returning an identity of a source and target node, respectively. As the join condition and attributes follow the same pattern for all joins in the chain query, we omit them whenever they are clear from the context.*

In other words, to follow a path p of length 2 means performing a join between the two paths of length 1 which p is composed from. The problem of join optimization is to determine the right order in which the joins should be computed, such that the overall response time for computing the path instances is minimized.⁴ Note that a chain query in Definition 4 does not include explicit joins, i.e., those specified in the *Where* clause, or by assigning the same variable names along the path expression. When we append these explicit joins, the shape of the query usually changes from a linear chain to a query graph containing a circle or a star, making the join ordering problem NP-hard [15].

⁴In case the sources offer also join capabilities the problem is not only in which order but also where the joins should take place.

4.1 Space Complexity

Disregarding the solutions obtained by the commutativity of joins, each query execution plan can be associated with a sequence of numbers that represents the order in which the relations are joined. We refer to this sequence as footprint of the execution plan.

EXAMPLE 3. For brevity reasons, assume the following name substitutions in the model introduced in Example 1: the concept names *Article*, *Employee*, *Organization*, *Project*, *ResearchTopic* become *a*, *b*, *c*, *d*, *e*, respectively; the property names *author*, *affiliation*, *carriesOut*, *topic* are substituted with *1*, *2*, *3*, *4*, respectively. Figure 4 presents two possible execution plans and their footprints.

Path expression	$a^1-b^2-c^3-d^4-e$	
Footprints	(3,2,4,1)	(2,1,4,3)
Query trees	<p style="text-align: center;">Right-deep tree</p>	<p style="text-align: center;">Bushy tree</p>

Figure 4: Two possible query executions and their footprints.

If also the order of the join operands matters, i.e., the commutativity law is considered, the sequence of the operands of each join is recorded in the footprint as well. The solution space consists of query plans (their footprints) which can be generated. We distinguish two cases: first the larger solution space of bushy trees and then its subset consisting of right-deep trees.

If we allow for an arbitrary order of joins the resulting query plans are so-called bushy trees where the operands of a join can be both a base relation⁵ or a result of a previous join. For a query with n joins there are $n!$ possibilities of different query execution plans if we disregard the commutativity of joins and cross products. Note that in the case of bushy trees, there might be several footprints associated with one query tree. For instance, the bushy tree in Example 3 can be evaluated in different order yielding two more footprints: (2, 4, 1, 3) or (4, 2, 1, 3). In our current approach, these footprints would be equivalent w.r.t. the cost they represent. However, treating them independently allows us to consider in the future also semi-join optimization [1] where their cost might differ considerably.

If the commutativity of join is taken into account, there are $\binom{2n}{n} \frac{n!}{2^n}$ different possibilities of ordering joins and their individual constituents [22]. However, in case of memory-resident databases where all data fits in main memory, the possibilities generated by the commutativity law can be for some join methods neglected as they mainly play a role in the cost model minimizing disk-memory operations; we discuss this issue further in Subsection 4.2. We adopt the memory-only strategy as in our context there are always only two

⁵A base relation is that part of the path which can be retrieved directly from one source.

attributes per relation, both of them being URI references which, when the namespace prefix is stored separately, yield a very small size. Of course, the assumption we make here is that the Sesame server is equipped with a sufficient amount of memory to accommodate all intermediate tuples of relations appearing in the query.

A special case of a general execution plan is a so-called right-deep tree which has the left-hand join operands consisting only of base relations. For a footprint that starts with the r -th join there are $\binom{n}{r}$ possibilities of finishing the joining sequence. Thus there are in total $\sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1}$ possibilities of different query execution plans.⁶ In this specially shaped query tree exists an execution pipeline of length $n-1$ that allows both for easier parallelizing and for shortening the response time [8] This property is very useful in the context of the WWW where many applications are built in a producer-consumer paradigm.

4.2 Cost Model

The main goal of query optimization is to reduce the computational cost of processing the query both in terms of the transmission cost and the cost of performing join operations on the retrieved result fragments. In order to determine a good strategy for processing a query, we have to be able to exactly determine the cost of a query execution plan and to compare it to costs of alternative plans. For this purpose, we capture the computational costs of alternative query plans in a cost model that provides the basis for the optimization algorithm that is discussed later.

As mentioned earlier, we adopt the memory-resident paradigm, and the cost we are trying to minimize is equivalent to minimizing the total execution time. There are two main factors that influence the resulting cost in our model. First is the cost of data transmission to the mediator, and second is the data processing cost.

DEFINITION 5 (TRANSMISSION COST). *The transmission cost of path instances of the schema path p from a source X to the mediator is modelled as $TC_p = C_{init_X} + |p| * Lngth_p * ||URI||_X * C_X$ where C_{init_X} represents the cost of initiating the data transmission, $|p|$ denotes the cardinality, $Lngth_p$ stands for the length of the schema path p , $||URI||_X$ is the size of a URI at the source X ⁷ and C_X represents transmission cost per data unit from X to the mediator.*

Since we apply all reducing operations (e.g., selections and projections) prior to the data transmission phase, the data processing mainly consists of join costs. The cost of a join operation is influenced by the cardinality of the two operands and the join-method which is utilized. As we already pointed out, there are no instance indices at the mediator side that would allow us to use some join “shortcuts”. In the following we consider two join methods: a nested loop join and a hash join both without additional indexing support.

DEFINITION 6 (NESTED LOOP JOIN COST). *The processing cost of a nested loop join of two relations p, r is defined as $NJC_{p,r} = |p| * |r| * K(p, r)$, where $|x|$ denotes the cardinality of the relation x and $K(p, r)$ represents the cost of the identity comparison.*

⁶The number corresponds to a sum of the $n-1$ -th line in the Pascal triangle.

⁷Different sources may model URIs differently, however, we assume that at the mediator all URIs are represented in the same way.

Note that the nested loop join allows for a more sophisticated definition of object equality than a common URI comparison. In particular, if necessary, the basic URI comparison can be complemented by (recursive) comparisons of property values or mapping look-ups. This offers room to address the issue of URI diversity also known as the designation problem, when two different URIs refer to the same real-life object.

DEFINITION 7 (HASH JOIN COST). *The processing cost of a hash join of two relations p, r is defined as $HJC_{p,r} = I * |p| + R * |r| * B$, where $|x|$ denotes the cardinality of the relation x , I represents the cost of inserting a path instance in the hash table (the building factor), R models the cost of retrieving a bucket from the hash table, and B stands for the average number of path instances in the bucket.*

Unlike the previous join method, the hash join algorithm assumes that the object equality can be determined by a simple URI comparison, in other words that the URI references are consistent across the sources. Another difference is that in the case of the nested loop join for in-memory relations the join commutativity can be neglected, as the query plan produced from another query plan by the commutativity law will have exactly the same cost. However, in the case of the hash join method the order of operands influences the cost and thus the solution space must also include those solutions produced by the commutativity law.

DEFINITION 8 (QUERY PLAN COST). *The overall cost of a query plan θ consists of the sum of all communication costs and all join processing costs of the query tree. $QPC_{\theta} = \sum_{i=1}^n TC_{p_i} + PC_{\theta}$, where PC_{θ} represents the join processing cost of the query tree θ and it is computed as a sum of recurrent applications of the formula in Definition 6 or 7 depending on which join method is utilized. To compute the cardinality of non-base join arguments, a join selectivity is used. The join selectivity σ is defined as a ratio between the tuples retained by the join and those created by the Cartesian product: $\sigma = \frac{|p \bowtie r|}{|p \times r|}$.*

As it is not possible to determine the precise join selectivity before the query is evaluated, σ for each sub-path join is assumed to be estimated and available in the source index hierarchy. After the evaluation of each query initial σ estimates are improved and made more realistic.

4.3 Heuristics for join ordering

While the join ordering problem in the context of a linear/chain query can be solved in a polynomial time [12], we have to take into account the more complex problem when also the explicit joins are involved which is proven to be NP-hard [15]. It is apparent that evaluating all possible join strategies for achieving the global optimum becomes quickly unfeasible for a larger n . In these cases we have to rely on heuristics that compute a “good-enough” solution given the constraints. In fact, this is a common approach for optimizers in interactive systems. There, optimization is often about avoiding bad query plans in very short time, rather than devoting a lot of the precious CPU time to find the optimal plan, especially, when it is not so uncommon that the optimal plan improves the heuristically obtained solutions only marginally.

Heuristics for the join ordering problem have been studied extensively in the database community. In this work we adopt the results of comparing different join ordering heuristics from [17]. Inspired from this survey, we chose to apply the two-phase optimization

consisting of the iterative improvement (II) algorithm followed by the simulated annealing (SA) algorithm [20]. This combination performs very well on the class of queries we are interested in, both in the bushy and the right-deep tree solution space, and degrades gracefully under time constraints.

The II algorithm is a simple greedy heuristics which accepts any improvement on the cost function. The II randomly generates several initial solutions, taking them as starting points for a walk in the chosen solution space. The actual traversal is performed by applying a series of random moves from a predefined set. The cost function is evaluated for every such move, remembering the best solution so far. The main idea of this phase is to descent rapidly into several local minima assuring aforementioned graceful degradation. For each of the sub-optimal solutions, the second phase of the SA algorithm is applied. The task of the SA phase is to explore the “neighborhood” of a prosperous solution more thoroughly, hopefully lowering the cost.

Algorithm 2 Simulated annealing algorithm

Require: start solution $sSolution$
Require: start temperature $sTemp$
 $solution := sSolution$
 $bestSolution := solution$
 $temp := sTemp$
 $cost := Cost(bestSolution)$
 $minCost := cost$
repeat
 repeat
 $newSolution := NEW(solution)$
 $newCost := Cost(newSolution)$
 if $newCost \leq cost$ **then**
 $solution := newSolution$
 $cost := newCost$
 else if $e^{-\frac{(newCost - cost)}{temp}} \geq RAND(0..1)$ **then**
 $solution := newSolution$
 $cost := newCost$
 end if
 if $cost < minCost$ **then**
 $bestSolution := solution$
 $minCost := cost$
 end if
 until equilibrium reached
 DECREASE($temp$)
until frozen
return $bestSolution$

The pseudo-code of the SA phase is presented in Algorithm 2. It takes a starting point/solution from the II phase, and similarly to II performs random moves from a predefined set accepting all cost improvements. However, unlike the II, the SA algorithm can accept with a certain probability also those moves that result in a solution with a higher cost than the current best solution. The probability of such acceptance depends on the temperature of the system and the cost difference. The idea is that at the beginning the system is hot and accepts easier the moves yielding even solutions with higher costs. However, as the temperature decreases the system is becoming more stable, strongly preferring those solutions with lower costs. The SA algorithm improves on the II heuristics by making the stop condition less prone to get trapped in a local minimum; SA stops when the temperature drops below a certain threshold or if the best solution so far was not improved in a number of consecutive

temperature decrements, the system is considered frozen. There are two sets of moves: one for the bushy solution space and one for the right-deep solution space; for details we refer the reader to [20].

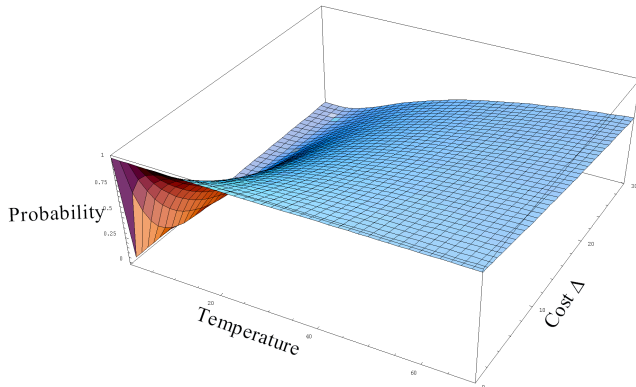


Figure 5: Acceptance probability with respect to the temperature and the cost difference.

Figure 5 shows the acceptance probability dependency in the SA phase computed for the range of parameters that we used in our experiments. As we adopted the two-phase algorithm our simulations were able to reproduce the trends in results presented in [17]; due to the lack of space we omit the detail performance analysis and the interested reader is referred to the aforementioned survey.

5. RELATED WORK

In this paper we focused mainly on basic techniques such as indexing and join ordering. Relevant related work is described in the remainder of this section. More advanced techniques such as site selection and dynamic data placement are not considered, because they are not supported by the current architecture of the system. We also do not consider techniques that involve view-based query answering techniques [6] because we are currently not considering the problem of integrating heterogeneous data.

5.1 Index Structures for Object Models

There has been quite a lot of research on indexing object oriented databases. The aim of this work was to speed up querying and navigation in large object databases. The underlying idea of many existing approaches is to regard an object base as a directed graph, where objects correspond to nodes, and object properties to links [16]. This view directly corresponds to RDF data, that is often also regarded as a directed graph. Indices over such graph structures now describe paths in the graph based on a certain pattern normally provided by the schema. Different indexing techniques vary on the kind of path patterns they describe and on the structure of the index. Simple index structures only refer to a single property and organize objects according to the value of that property. Nested indices and path indices cover a complete path in the model that might contain a number of objects and properties [2]. In RDF as well as in object oriented databases, the inheritance relation plays a special role as it is connected with a predefined semantics. Special index structures have been developed to speed up queries about such hierarchies and have recently been rediscovered for indexing RDF data [5]. In the area of object-oriented database systems, these two kinds of indexing structures have been combined resulting in the so-called nested inheritance indices [3] and generalized nested inheritance indices

[16]. These index structures directly represent implications of inheritance reasoning, an approach that is equivalent to indexing the deductive closure of the model.

5.2 Query Optimization

There is a long tradition of work on distributed databases in general [13] and distributed query processing in particular [10]. The dominant problem is the generation of an optimal query plan that reduces execution costs as much as possible while guaranteeing completeness of the result. As described by Kossmann in [10], the choice of techniques for query plan generation depends on the architecture of the distributed system. He discusses basic techniques as well as methods for client-server architectures and for heterogeneous databases. Due to our architectural limitations (e.g., limited source capabilities) we focused on join-ordering optimization which can be performed in a centralized manner by the mediator. While some restricted cases of this problem can be solved in a polynomial time [12, 11], the general problem of finding an optimal plan for evaluating join queries has been proven to be NP-hard [15]. The approaches to tackle this problem can be split into several categories [17]: deterministic algorithms, randomized algorithms, and genetic algorithms. Deterministic algorithms often use techniques of dynamic programming (e.g. [12]), however, due to the complexity of the problem they introduce simplifications, which render them as heuristics. Randomized algorithms (e.g. [20, 19]), perform a random walk in the solution space according to certain rules. After the stop-condition is fulfilled, the best solution found so far is declared as the result. Genetic algorithms (e.g. [18]) perceive the problem as biological evolution; they usually start with a random population (set of solutions) and generate offspring by applying a crossover and mutation. Subsequently, the selection phase eliminates weak members of the new population.

6. LIMITATIONS AND FUTURE WORK

The work reported in this paper can be seen as a very first step towards a solution for the problem of distributed processing of RDF queries. We motivated the overall problem and proposed some data structures and algorithms that deal with the most fundamental problems of distributed querying in a predefined setting. We identified a number of limitations of the current proposal with respect to the generality of the approach and assumptions made. These limitations also set the agenda for future work to be done on distributed RDF querying and its support in Sesame.

Implementation Currently, our work on distributed query processing is of a purely theoretical nature. The design and evaluation of the methods described are based on previous work reported in the literature and on worst-case complexity estimations. The next step is to come up with a test implementation of a distributed RDF storage system. The implementation will follow the architecture introduced in the beginning of the paper and will be built on top of the Sesame storage and retrieval engine. The implementation will provide the basis for a more practical evaluation of our approach and will allow us to make assertions about the real system behavior in the presence of different data sets and different ways they are distributed. Such a practical evaluation will be the basis for further optimization of the methods.

Schema-Awareness One of the limitations of the approach described in this paper concerns schema aware querying in a distributed setting. Even if every single repository is capable of computing the deductive closure of the model it contains, the overall

result is not necessarily complete, as schema information in one repository can have an influence on information in other repositories. This information could lead to additional conclusions if taken into account during query processing. In order to be able to deal with this situation, we need to do some additional reasoning within the mediator in order to detect and process dependencies between the different models.

Object Identity One of the basic operations of query processing is the computation of joins of relations that correspond to individual properties. The basic assumption we make at this point is that we are able to uniquely determine object identity. Identity is essential because it is the main criterion that determines whether to connect two paths or not. From a pragmatic point of view, the URI of an RDF resource provides us with an identity criterion. While this may be the case in a single repository, it is not clear at all whether we can make this assumption in a distributed setting as different repositories can contain information about the same real world object (e.g., a paper) and assign different URIs to it. To deal with this situation we have to develop heuristics capable of deciding whether two resources describe the same real world object.

Query Model In order to be able to design efficient index structures we restricted ourselves to path queries as a query model that is directly supported. We argued above that tree-shaped queries can be easily split into a number of path queries that have to be joined afterwards. Nevertheless, this simplification does not apply to the optimization part which is capable of processing also different query shapes. An important aspect of future work is to extend our indexing approach to more expressive query models that also include tree and graph shaped queries which can be found in existing RDF query languages. It remains to be seen whether the same kind of structures and algorithms can be used for more complex queries or whether we have to find alternatives.

Architecture The starting point of our investigation was a particular architecture, namely a distributed repository where the data is accessed at a single point but stored in different repositories. We further made the assumption that these repositories are read-only, i.e., they only provide answers to path queries that they are known to contain some information about.

An interesting question is how more flexible architectures can be supported. We think of architectures where information is accessed from multiple points and repositories are able to forward queries. Further we can imagine grid-based architectures where components can perform local query processing on data received from other repositories. A prominent example of such more flexible architectures are peer-to-peer systems. This would also bring a new potential for optimization as peers may collaborate on query evaluation which in turn may help in reducing both the communication and processing costs.

7. REFERENCES

- [1] P. Bernstein and D. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28:25–40, 1981.
- [2] E. Bertino. An indexing technique for object-oriented databases. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 160–170. IEEE Computer Society, 1991.
- [3] E. Bertino and P. Foscoli. Index organizations for object-oriented database systems. *TKDE*, 7(2):193–209, 1995.
- [4] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Web - ISWC 2002*, volume 2342 of *LNCIS*, pages 54–68. Springer, 2002.
- [5] V. Christophides, D. Plexousakisa, M. Scholl, and S. Tourounis. On labeling schemes for the semantic web. In *Proceedings of the 13th World Wide Web Conference*, pages 544–555, 2003.
- [6] A. Halevy. Answering queries using views - a survey. *The VLDB Journal*, 10(4):270–294, 2001.
- [7] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, (2), 2001.
- [8] H. Hsiao, M. Chen, and P. Yu. Parallel execution of hash joins in parallel databases. *IEEE Transactions on Parallel and Distributed Systems*, 8:872–883, 1997.
- [9] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *ACM SIGMOD International Conference on Management of Data*, pages 9–22. ACM:Press, 1987.
- [10] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
- [11] G. Moerkotte. Constructing optimal bushy trees possibly containing cross products for order preserving joins is in P, tr-03-012. Technical report, University of Mannheim, 2003.
- [12] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *16th International Conference on Very Large Data Bases*, pages 314–325. Morgan Kaufmann, 1990.
- [13] M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [14] D. Rotem. Spatial join indices. In *Proceedings of International Conference on Data Engineering*, 1991.
- [15] W. Scheufele and G. Moerkotte. Constructing optimal bushy processing trees for join queries is NP-hard, tr-96-011. Technical report, University of Mannheim, 1996.
- [16] B. Shidlovsky and E. Bertino. A graph-theoretic approach to indexing in object-oriented databases. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 230–237. IEEE Computer Society, 1996.
- [17] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for join ordering problem. *The VLDB Journal*, 6:191–208, 1997.
- [18] M. Stillger and M. Spiliopoulou. Genetic programming in database query optimization. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 388–393. MIT Press, 1996.
- [19] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *ACM SIGMOD International Conference on Management of Data*, pages 367–376. ACM:Press, 1989.
- [20] A. Swami and A. Gupta. Optimization of large join queries. In *ACM SIGMOD International Conference on Management of Data*, pages 8–17. ACM:Press, 1988.
- [21] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 522–533, 1994.
- [22] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, 1998.