Towards distributed processing of RDF path queries

Heiner Stuckenschmidt*

Vrije Universiteit Amsterdam, de Boelelaan 1081a,
1081HV Amsterdam, The Netherlands
E-mail: heiner@cs.vu.nl
*Corresponding author

Richard Vdovjak

Technische Universiteit Eindhoven, Den Dolech 2,
5612 AZ Eindhoven, The Netherlands
E-mail: r.vdovjak@tue.nl

Jeen Broekstra

Vrije Universiteit Amsterdam, de Boelelaan 1081a,
1081HV Amsterdam, The Netherlands

and

Aduna, Prinses Julianaplein 14 b, 3817 CS, Amersfoort,
The Netherlands
E-mail: Jeen.Broekstra@aduna.biz

Geert-Jan Houben

Technische Universiteit Eindhoven, Den Dolech 2,
5612 AZ Eindhoven, The Netherlands
E-mail: g.j.houben@tue.nl

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 support 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 optimisation in such a distributed context.

Keywords: index structures; optimisation; RDF querying.

Reference to this paper should be made as follows: Stuckenschmidt, H., Vdovjak, R., Broekstra, J. and Houben, G-J. (2005) 'Towards distributed processing of RDF path queries', *Int. J. Web Engineering and Technology*, Vol. 2, Nos. 2/3, pp.207–230.

Biographical notes: Heiner Stuckenschmidt is a Senior Researcher in the Knowledge Representation and Reasoning Group at the Vrije Universiteit, Amsterdam. His research interests include practical knowledge representation and non-standard reasoning for semantic search and information integration on the web; decentralised and context-based semantic systems and the use of semantic web technologies for the development of intelligent information systems for medical and scientific applications. For a list of publications and further information about his activities in the area look at: http://www.cs.vu.nl/~heiner/.

Richard Vdovjak holds a professional doctorate in software engineering and a PhD in computer science from Eindhoven University of Technology. Between 2004 and 2005 he was employed by the same university as assistant professor in the field of information systems and databases. From July 2005 he has worked for Philips Research Eindhoven. His research interests include ethodologies for engineering web information systems, information integration, architectures for distributed RDF(S) query processing, RDF(S) query optimisation, and model-driven software development.

Jeen Broekstra is a PhD researcher in the Knowledge Representation and Reasoning Group of the Vrije Universiteit Amsterdam. His main research topic is storage, querying and inferencing for semantic web languages. He is also a software developer at Aduna and is one of the lead designers of the Sesame RDF Framework. He has (co)written numerous publications on semantic web technology.

Geert-Jan Houben is a full professor in information systems at the Vrije Universiteit Brussel and co-director of the WISE laboratory for Web & Information System Engineering. He is part-time associated with the Computer Science department of the Technische Universiteit Eindhoven (TU/e). Geert-Jan leads in Brussels and Eindhoven the Hera research program on web information systems, with topics like web and semantic web technology (XML, RDF), semi-structured data, web query and transformation languages, web data integration and distribution, web presentation generation, adaptation and personalisation in web applications, web engineering, and WIS architecture. More information can be found at wwwis.win.tue.nl/~houben.

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 (compare Hendler, 2001). Despite the inherently distributed nature of the semantic web, most current RDF infrastructures (for example, Broekstra et al. (2002)) 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 the real distribution of information sources that can be accessed from a single point. Beyond the argument of conceptual adequacy, in many cases, it will even be unavoidable to adopt a distributed architecture, for example in scenarios in which the data are 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 just to create a single model containing all the information, but can still be queried using a special interface (Musicbrainz[1] 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, we take an existing RDF storage and retrieval system as a starting point 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. Section 2 presents the Sesame RDF storage and retrieval system and describes how its architecture can be extended to allow for distributed RDF information sources. From this architecture we derive a set of requirements for the design of access methods. In Sections 3 and 4 we present query processing techniques for distributed RDF information sources. While Section 3 introduces an index structure for locating information in the different sources, Section 4 proposes optimisation techniques for speeding up query processing based on well-known database technology. We briefly discuss the problem of object identity in query processing in Section 5 and sketch a hybrid approach to deal with this problem. We conclude with a discussion of open questions and future work.

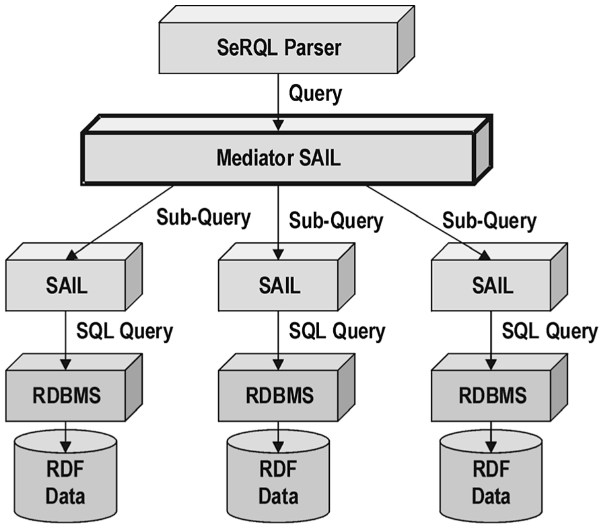
2 Distributed architecture

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 (Broekstra et al., 2002) to distributed data sources.

The Sesame architecture is flexible enough to allow for 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 passed directly 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 that information relevant to a query might be distributed over the different sources. This requires us to locate relevant information, to retrieve it, and to 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).

Figure 1 Distributed architecture



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 Christophides et al., 2003). 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 optimisation on the SQL queries they pose to the data. Therefore we do not have to perform optimisations 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.

3 Index structures for distributed models

As discussed above, in order to be able to make use of the optimisation mechanisms of the database engines underlying the different repositories, we have to forward complete 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 which 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 complete 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.

Suppose we now want to ask for the titles of articles by employees of organisations that have projects in the area 'RDF'. The path expression of a corresponding SeRQL query would be the following:[2]

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

Now, let us assume that we have three information sources S_1, S_2 and S_3. S_1 is a publication database that contains information about articles, titles, authors and their affiliations. S_2 is a project database with information about industrial projects, topics, and organisations. 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 complete 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

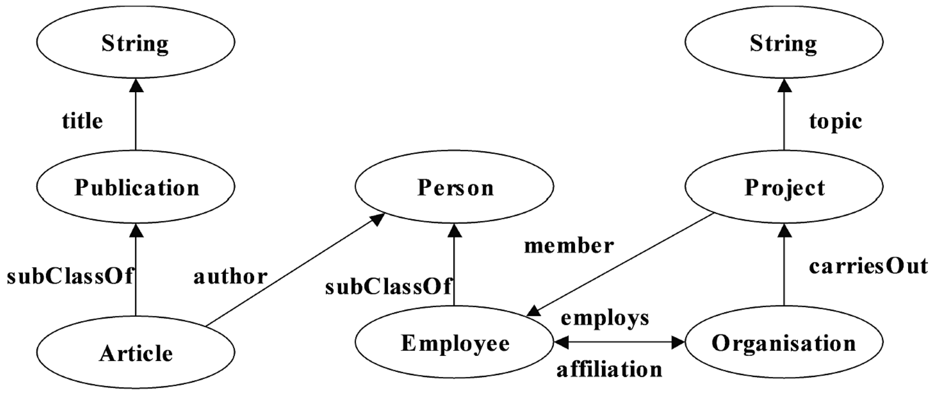
```
{A} title {T};
  author {W} affiliation {O}
```

to S_1, the fragment

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

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

Figure 2 An example RDF schema



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.

3.1 Source index hierarchies

The majority of work in the area of object orientated 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 arcs to properties linking these resources. The result of a query to such a model is a set of subgraphs corresponding to a path expression.[3] 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, 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 indexes 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 subpaths without losing the advantage of indexing complete paths. An index structure that combines these two characteristics is the join index hierarchy proposed in Xie and Han (1994). We, therefore, take their approach as a basis for defining a *source index hierarchy*.

Definition 1: Schema path

Let $G = (V, E, L, s, t, l)$ be a labeled 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 $e_0, \ldots, e_{n-1}$ such that $t(e_i) = s(e_{i+1})$ for all $i = 0, \ldots, n-2$. Let $p = e_0, \ldots, e_{n-1}$ be a path, the corresponding schema path is a list of labels $l_0, \ldots, l_{n-1}$ 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, then a source index for p is a set of pairs (s_k, n_k) where s_k are information sources (in particular RDF models) 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

3-Stuckenschmidt.qxp 14/12/2005 12:56 Page 213

get results. The information about the number of instance paths can be used to estimate communication costs and will be used for join ordering. 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 subpaths. The corresponding structure is defined as follows.

Definition 3: Source index hierarchy

Let $p = l_0, \ldots, l_{n-1}$ be a schema path. A source index hierarchy for p is an n-tuple $(P_n, \ldots, P_1)$ where

- P_n is a source index for p
- P_i are the sets of all source indices for subpaths 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 subpaths 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 subpath. 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

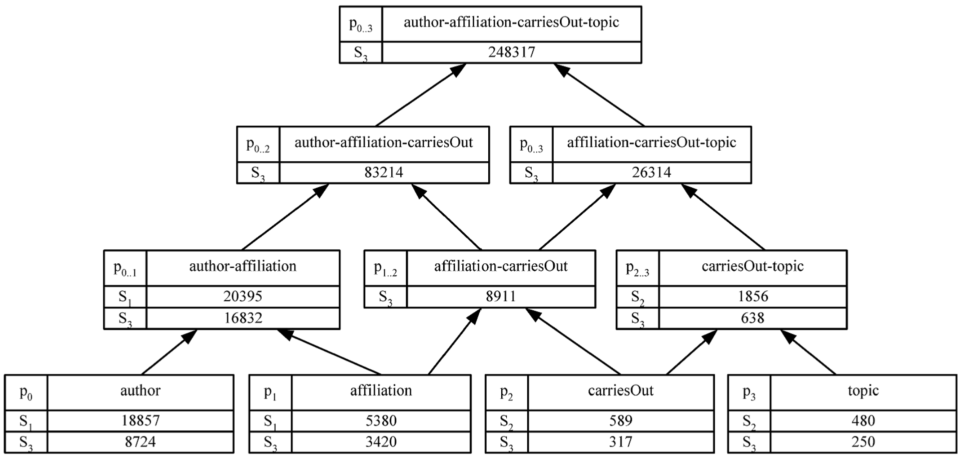
Let us reconsider the situation in Example 1. The schema path we want to index is given by the list (author, affiliation, carriesOut, topic). The source index hierarchy for this path therefore contains source indices for the paths

- $p_{0\ldots3}$: (author, affiliation, carriesOut, topic)
- $p_{0\ldots2}$: (author, affiliation, carriesOut), $p_{1\ldots3}$: (affiliation, carriesOut, topic)
- $p_{0\ldots1}$:(author, affiliation), $p_{1\ldots2}$:(affiliation, carriesOut), $p_{2\ldots3}$:(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 subpaths, we also find out that S_1 contains results for the subpath (author, affiliation) and S_2 for the subpath (carriesOut, topic) that we can join in order to compute results, because together both subpaths make up the path we are looking for.

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

Figure 3 Example of a source index hierarchy



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 subpath of the indexed schema path. As the number of all subpaths of a path is $\sum_{i=1}^{n} i$, the worst-case[4] space complexity of a source index hierarchy is $O(s \bullet n^2)$. We conclude that the length of the indexed path is the significant parameter here.

3.2 *Basic 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 subpaths 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 subpaths for the 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 subpaths. 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 in Algorithm 1.

Algorithm 1 Compute answers (naive)

```
REQUIRE  A schema path $p = l_0, \ldots, l_{n-1}$
REQUIRE  A source index hierarchy $h = (P_n, \ldots, P_1)$ for $p$

FORALL  sources $s_k$ in source index $P_n$
  Answers = instances of schema path $p$ in source $s_k$
  Result = result $\cup$ answers

IF  $n \geq 2$
  FORALL  $i = 1 \ldots n-1$
    $p_{0 \ldots i-1} = l_0, \ldots l_{i-1}$
    $p_{i \ldots n-1} = l_i, \ldots l_{n-1}$
    $h_{0 \ldots i-1}$ = Sub-hierarchy of $h$ rooted at the source index for $p_{0 \ldots i-1}$
    $h_{i \ldots n-1}$ = Sub-hierarchy of $h$ rooted at the source index for $p_{i \ldots n-1}$
    $res_1$ = ComputeAnswers($p_{0 \ldots i-1}, h_{0 \ldots i-1}$)
    $res_2$ = ComputeAnswers($p_{i \ldots n-1}, h_{i \ldots n-1}$)
    $result$ = result $\cup$ join($res_1$, $res_2$)

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 neither has it selected an optimal order for joining subpaths. We can improve this situation by using knowledge about the information in the different sources and performing query optimisation. Prior to that, however, we have to establish a clear notion of identity of RDF Resources.

4 On the notion of identity in RDF

As already pointed out the notion of identity in RDF is a complex issue albeit an important one. To determine whether two resources are identical is not only important for avoiding duplicates in the result set but it is an essential requirement for being able to compute the join of two paths and thus compute an answer to a distributed query.

With respect to the identity notion, we divide RDF models into two groups:

- those where a URI comparison suffices for establishing identity
- those where the simple comparison does not work.

In the following, we focus on the second group, the more difficult one.

To illustrate the problem, we use an example from an experiment we conducted on integrating information about scientific publications from independently created bibtex files in a single RDF representation. We discovered a couple of typical problems that make it difficult to determine whether two resources describe the same object. As a result of overlaps between the bibtex files, the merged file contained a significant number of duplicates each having a unique ID, because each bibtex source had its own namespace and the resource IDs were created on the basis of the identifier given to the respective bibtex entry as shown in the example below:

```
<ow:Article rdf:about = ‘http://www.cs.vu.nl...#Berners-Lee_2001a’>
   <ow:author rdf:resource = ‘http://www.cs.vu.nl...#tim_berners-lee’ />
   <ow:author rdf:resource = ‘http://www.cs.vu.nl...#jim_hendler’ />
   <ow:author rdf:resource = ‘http://www.cs.vu.nl...#ora_lassila’ />
   <ow:title>The Semantic Web</ow:title>
   <ow:journal>Scientific American</ow:journal>
   <ow:month>May</ow:month>
   <ow:year>2001</ow:year>
   <ow:pages>34-43</ow:pages>
   <ow:volume>284</ow:volume>
   <ow:number>5</ow:number>
</ow:Article>

<ow:Article rdf:ID = ‘bernersLee01semantic-web’>
   <ow:author rdf:resource = ‘#berners-lee’ />
   <ow:author rdf:resource = ‘#hendler’ />
   <ow:author rdf:resource = ‘#lassila’ />
   <ow:title>The semantic web</ow:title>
   <ow:journal>SciAm Magazine</ow:journal>
   <ow:year>2001</ow:year>
   <ow:volume>284</ow:volume>
   <ow:number>5</ow:number>
   <ow:pages>34-43</ow:pages>
</ow:Article>
```

From the above we can clearly see that in this case comparing URIs alone does not suffice to determine the identity of real-world objects to which the URIs refer. What happens is that the same object is represented by different resources, because every information source has its own way of representing objects. In the case of document repositories, it might be the case that a certain article is only contained in one repository; authors, projects and organisations, however, will often occur in different sources. Therefore, we need mechanisms for deciding whether two resources describe the same real-world object.

In principle, there are two general ways to approach this problem:

Matching: by comparing the properties of two resources, we can determine a degree of similarity. If the similarity is above a certain threshold, we decide that the described objects are equal. This approach depends on a number of parameters like the similarity measure used, the threshold chosen and the set of properties used for determining similarity. In an extreme form, we can choose only to consider two objects identical if they share all properties. This approach, however, suffers from the same problem as the values of properties of an object, even if they are not complex objects themselves, are often represented differently across sources. In the experiment for example, we discovered three different descriptions of the Springer Verlag, that was referred to as ‘Springer’, ‘Springer Verlag’ and ‘Springer-Verlag’, respectively. While in this case it is quite obvious that the descriptions refer to the same publisher, there are also examples where it is less obvious, e.g. names of authors: compare names ‘P. Patel-Schneider’ and ‘Peter F. Patel-Schneider’. Without background knowledge it is impossible to decide whether the name refers to the same person or to different persons with the same surname. Further complications arise from the use of abbreviations, e.g. ‘LNCS’ instead of ‘Lecture Notes in Computer Science’.

Mapping: we can acquire and maintain explicit knowledge about resources describing the same objects in a separate mapping table and use it when computing the join of two paths. The problem of this approach is the effort needed to acquire the mappings. In the presence of large information sources with millions of documents, it is almost impossible to determine useful mappings.

In order to weaken the shortcomings of the two approaches described above, we consider that in many cases a combination of matching and mapping constitutes a good trade-off between the effort of defining mappings and mapping accuracy.

The basic idea of our approach is to use exact matching on a predefined set of key properties that can be assumed to uniquely determine an object. In the case of an article, such a set of properties could be the name and volume of the journal and the page numbers. This choice already frees us from the need to compare authors which might be difficult due to misspellings and the different use of surnames in different sources (see example above). For some properties like the page numbers, exact matching is appropriate, because there is little or no ambiguity in the values. For other properties relying on exact matches may introduce errors. The journal title is an example: 'Scientific American' vs. 'SciAm'.

Under certain circumstances creating an explicit mapping between objects of a certain type in different sources can be an appropriate solution for this situation. If the number of potential objects in a class is relatively low, it is beneficial to use explicit mappings instead of recursively checking equality between objects. In our example, journals are a good example of a class of objects, where a mapping approach makes sense: the journal is a key value for another class of objects and there is a relatively low number of different journals compared to the number of articles or authors (in a recent application we have developed, the ratio is below 1:1000). In many cases, we do not even need mappings that explicitly relate object in different information sources. It is often sufficient to keep a record of potential descriptive variants, like different abbreviations used for the same journal name, e.g. JAIR vs. Journal of AI Research vs. Journ. AI Research. When a list of alternative terms is available, we can easily perform normalisation to a preferred term and rely subsequently on precise matching. A task that remains is the determination of the right set of keys for the different object classes involved and the choice of classes whose members should be mapped explicitly. These choices depend on the domain of interest and are, therefore, context depended.

5 Query optimisation

Above we described a lightweight 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 where the queries are not man-entered anymore but rather generated by a portal's front-end (triggered by the user) and often exceed the size[5] which can be easily computed by using brute force. Therefore, we focus in this section on query optimisation as an important part of a distributed RDF query system. We try to avoid reinventing the wheel and once again seek for inspiration in the database field, making it applicable by 'relationising' 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 $s(p_i)$ 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 later formalised in Definition 4. This relational view over an RDF graph offers the possibility to re-use the last two decades of extensive research on join optimisation in databases, e.g. Bernstein and Chiu (1981), Hsiao et al. (1997), Ioannidis and Wong (1987), Steinbrunn et al. (1997) and Swami and Gupta (1988).

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 contribute significantly to the overall cost of a query plan. Since, in our case, the distribution is assumed to be realised via an IP network with a variable bandwidth, the communication costs are likely to contribute substantially to the overall processing costs, which makes the minimisation 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, hence must be transmitted via the network.

There can be different dependencies (both structural and extensional) on the way the data are distributed. If the information about such dependencies is available, it essentially enables the optimiser 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 necessary for the optimiser to make the right decision. In general, the more the optimiser knows about the underlying sources and data, the better optimised the query plan will be. However, taking into account the autonomy of the sources, the necessary statistics do not always have to be 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 optimiser is 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 optimiser 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 (Bernstein and Chiu, 1981). 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, we focus mainly on local optimisation 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 optimisation results from the area of relational databases.

In this context the problem of join ordering arises, when we want to compute the results for a schema path 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 of length n and let $p_0, \dots, p_{n-1}$ denote the subpaths of length 1 that p is composed from. The chain query of p is the n-join $p_0 \bowtie_{t(p_0) = s(p_1)} p_1 \bowtie_{t(p_1) = s(p_2)} p_2 \bowtie \dots \bowtie p_{n-1}$, where $s(p_i)$ and $t(p_i)$ are returning an identity (URI) of a source and target vertex of the arc p_i, 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. Note that a chain query does not include explicit joins specified in a SeRQL query in the 'Where' clause or by assigning the same variable names along the path expression. These must be considered after the path instances are computed.

The problem of join optimisation 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 minimised.[6]

5.1 Space complexity

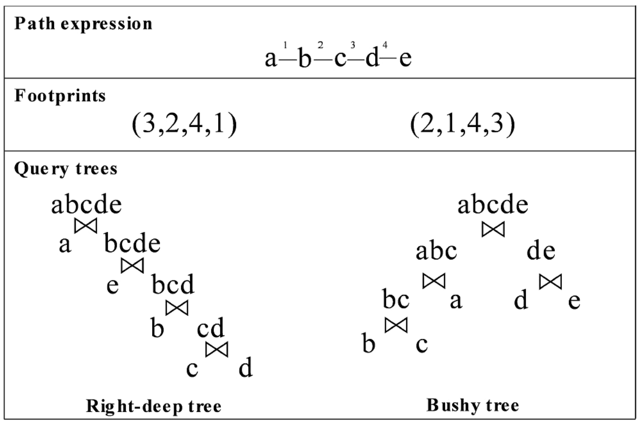
In case we disregard the solutions obtained by applying the commutativity of the join operation, each query execution plan can be identified with a sequence of numbers that represent the order in which the relations are joined. We refer to this sequence as the footprint of the execution plan.

Example

For reasons of brevity, assume the following name substitutions in the model introduced in Example 1: the concept names *Article, Employee, Organisation, 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 for the above path expression.

If the order of the join operands also matters, i.e. the commutativity law is considered, the sequence of the operands of each join is also recorded in the footprint. 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 its subset consisting of right-deep trees.

Figure 4 The problem of join ordering for chain queries



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 either a base relation[7] or a result of a previous join. For a path of length *n* there are *n*! possibilities of different query execution plans if we disregard the commutativity of joins. 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 (Yu and Meng, 1998). However, in case of memory-resident databases where all data fit in the main memory, the possibilities generated by the commutativity law can be neglected for some join methods as they mainly play a role in the cost model minimising disk-memory operations; we discuss this issue further in the next section. We adopt the memory-only strategy as, in our context, there are most of the time only two attributes per relation (two neighbouring URI references) which usually 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.[8] In this specially shaped query tree there is an execution pipeline of length *n*–1 that allows for both easier parallelising and shortening of the response time (Hsiao et al., 1997). This property is very useful in the context of the www where many applications are built in a producer-consumer paradigm.

5.2 Distribution aspects

The distribution of data often follows certain patterns or dependencies. Some of them are imposed by the geographical nature of distribution; some are introduced due to the diversity of source domains. For instance, one source may focus on a domain A whereas another source on a domain B. If there is no overlap between the instances of the two domains we can conclude that there is no overlap between the schema paths covered by those sources either. Such information, when available, can be used by the optimiser and the path expressions from the two sources can be evaluated independently from each other, using a set union operation at the end to produce the final result. This effectively reduces the number of tuples which are to be joined and thus also the overall costs. In the following we give formal definitions of data fragmentations and dependencies that may occur in our distributed context.

Definition 5: Horizontal fragmentation

Let p be a schema path, let $Ext(p)$ denote the extension of p and $X(p)$ those instances of p residing on the source X. We say that p is horizontally fragmented if there exists more than one X such that $|X(p)| > 0$.

Example 3

The schema path (author, affiliation) is horizontally fragmented because there are two different sources (S_1, S_3) that contain instances for this schema path.

Definition 6: Vertical fragmentation

Let p be a schema path and let r and s denote two not overlapping subpaths of p. Let $Ext(r) = \cup\{X_i(p), |X_i(p)| \geq 0\}$ and similarly $Ext(s) = \cup\{X_j(p), |X_j(p)| \geq 0\}$, and let I_X, J_X denote the two sets of sources satisfying the above conditions for r and s respectively. Then p is (a) vertically fragmented, (b) partially vertically fragmented on r and s if (a) $I_X \cap J_X = \varnothing$ and (b) $I_X \neq J_X$. Informally, we say that p is (partially) vertically fragmented if (some) instances of r reside on different sources than (some of) the instances of s.

Example 4

The schema path (author, affiliation, carriesOut, topic) is partially vertically fragmented since its two subpaths (author, affiliation) and (carriesOut, topic) reside on two different sets of sources, i.e. $\{S_1, S_3\}$ and $\{S_2, S_3\}$, respectively.[9]

Note that no path of length 1 is vertically fragmented. This result follows directly from Definition 6 and also from the atomicity of RDF triplets. Unlike in the case of vertical fragmentation where the 1-path relation would have to be split in attributes beyond the atomicity of an RDF triplet, the horizontal fragmentation can still occur, i.e. several sources can contain instances of the same 1-path.

Definition 7: Total placement dependency

Let r and s be two schema paths. There is a total placement dependency between r and s if $r \bowtie s = \cup X(r) \bowtie X(s)$ for each source X containing instances of r and s.

Definition 8: Partial placement dependency

Let r and s be two schema paths. There is a partial horizontal placement dependency between r and s if there exists a total placement dependency between a subset of instances of r and a subset of s. In other words, there exist sources X, Y, for which $X(r) \bowtie Y(s) = \emptyset$, even though $|X(r)| > 0$, $|Y(s)| > 0$.

If there was no overlap between the data coming from the source S_3 (representing the academic research) and the data from the other sources (representing the industrial research) there would be a partial placement dependency between all adjacent 1-paths in our example. This information can be used by the optimiser to prune those joins involving subpaths from S_3 combined with subpaths coming from the other two sources; such pruning reduces the search space and thus speeds up the optimisation.

5.3 Meta optimisation

Unlike in databases, both functionality and performance of distributed RDF systems is very context dependent. In order to be able to apply the optimisation methods devised in the database field, we must first make the implicit context (determined by the application domain and other conditions) explicit. Therefore, it is essential for the optimiser to undergo a so-called meta-optimisation phase in which it is fine-tuned to a concrete context situation. Before the optimiser starts reordering joins and searching for an optimal query plan, there are some high level optimisation decisions to be made which influence both the type and the behaviour of the actual optimisation algorithm as well as the notion of optimality and the size of the solution space.

The implicit context is given by several factors amongst which the source capabilities, the identity notion, the length of the query and the available distribution dependencies play a crucial role. The pseudo-code describing the individual steps is summarised in Algorithm 2. This algorithm can be seen as a generic optimisation framework for solving path expression queries in the context of distributed RDF (meta)data.

The algorithm starts by determining applicable sources w.r.t. to the query based on the placement dependency then follows by splitting the query path into subpaths which are covered entirely by local sources. The length of the inter source join query is then equal to the number of generated subpaths. The subpaths are subsequently pushed to the underlying sources where reducing operations (e.g. selection) take place. If the sources are capable of join operations the mediator can apply the semi-join optimisation. Subsequently, all resulting path instances are transferred to the mediator where the join ordering optimisation is performed. Depending on whether the instance identity can be determined solely by URIref comparison or a more sophisticated method is required, the join method is set to be a hash join or a nested-loop join, respectively. After that, the solution space (bushy trees or right-deep trees) and the ordering optimisation algorithm (the exact search or the heuristic search) is chosen depending on the length of the inter-source join query. These decisions are made based on several threshold values which are hardware dependent and must be fine-tuned for a concrete system.

Algorithm 2: Meta-optimisation

```
REQUIRE  Query Q
REQUIRE  Source Hierarchy Index H
REQUIRE  Source join capability JoinCapable
REQUIRE  URIref identity URIrefIdentifiable
REQUIRE  Placement dependency information PDI

   Using PDI determine applicable sources
   Using H split Q into subpaths covered by sources
   Length  = number of generated subpaths
   Push Query fragments to the sources
   Apply reducing operations (e.g. selections)

IF  JoinCapable
   Distribution of joins among the sources
   Use semijoins where beneficial
   Transfer all path instances to the mediator

IF  URIrefIdentifiable
   joinMethod = 'Hash join'
   comparMethod = 'URIref comparison'
ELSE
   joinMethod = Nested-loop join
   comparMethod = Mapping + Matching + URIref comparison

IF  length ≤ treshold1
   solutionSpace = 'bushy trees'
   bestSolution = EXACTSEARCH(Q)
ELSIF  length ≤ treshold2
   solutionSpace = 'right-deep trees'
   bestSolution = EXACTSEARCH(Q)
ELSIF  length ≤ treshold3
   solutionSpace = 'bushy trees'
   bestSolution = HEURISTICSEARCH(Q)
ELSIF  length ≤ treshold4
   solutionSpace = 'right-deep trees'
   bestSolution = HEURISTICSEARCH(Q)
RETURN  bestSolution.
```

5.4 Cost model

The main goal of query optimisation 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 determine the cost of a query execution plan exactly 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 optimisation algorithm that is discussed below.

As mentioned above, we adopt the memory-resident paradigm, and the cost we are trying to minimise is equivalent to minimising 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 9: Transmission cost

The transmission cost of path instances of the schema path p from a source X to the mediator is modelled as $TC_p = Cinit_X + |p|*Lngth_p*||URI||_X*C_X$ where $Cinit_X$ 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,[10] 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 utilised. 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 10: Nested loop join cost

The processing cost of a nested loop join of two relations p,r is defined as $NJC_{p,r} = |p|*|r|*K$, where $|x|$ denotes the cardinality of the relation x and K represents the comparison cost of two objects.

Note that the nested loop join allows for a more sophisticated definition of instance identity than a common URI comparison. In particular, if necessary the basic URI comparison can be complemented by methods discussed in the section about object identity.

Definition 11: 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 instance identity 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 12: 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.

$$QPC_\theta = \sum_{i=1}^{n} TCp_i + PC_\theta$$

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 10 or 11 depending on which join method is utilised. To compute the cardinality of non-base join arguments, join selectivity is used. The join selectivity σ is defined as a ratio between the tuples retained

$$\sigma = \frac{|p \rhd\lhd r|}{|p \times r|}$$

by the join and those created by the Cartesian product:

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.

5.5 Heuristics for join ordering

Looking at the space complexity, 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.

As we have shown, our problem of evaluating path expressions corresponds exactly to evaluation of chain queries. This allows us to directly adopt the results of comparison of different join ordering heuristics in (Steinbrunn et al., 1997). Inspired from this survey, we chose to apply the two-phase optimisation consisting of the iterative improvement (II) algorithm followed by the simulated annealing (SA) algorithm (Swami and Gupta, 1988). This combination performs very well on the class of chain join 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 descend rapidly into several local minima, assuring the 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 'neighbourhood' of a prosperous solution more thoroughly, hopefully lowering the cost.

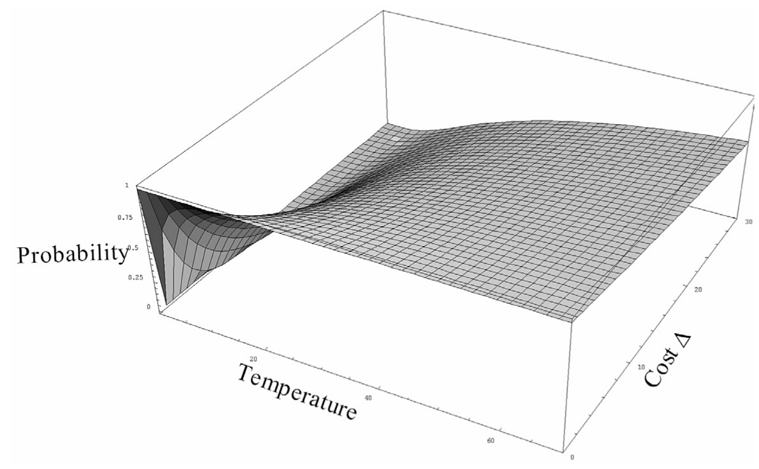
The pseudo-code of the SA phase is presented in Algorithm 3. It takes a starting point/solution from the II phase, and similarly to II performs random moves from a predefined set accepting all cost improvements. However, unlike the II, the SA algorithm can also accept with a certain probability those 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 more easily 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 getting 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 Swami and Gupta (1988). The same set of moves is used both for the II and SA phases.

Algorithm 3: Simulated annealing

```
REQUIRE start solution sSolution
REQUIRE start temperature sTempr

  Solution = sSolution
  bestSolution = solution
  tempr = sTempr
  cost = Cost(bestSolution)
  minCost = cost
REPEAT
  REPEAT
    newSolution = NEW(solution)
    newCost = Cost(newSolution)
    IF newCost ≤ cost
      Solution = newSolution
      Cost = newCost
    ELSIF $e^{-(newCost-cost)/tempr}$ ≥ RAND(0. . .1)
      Solution = newSolution
      Cost = newCost

    IF cost<minCost
      bestSolution = solution
      minCost = cost
  UNTIL equilibrium reached
  DECREASE(tempr)
UNTIL frozen

RETURN bestSolution.
```

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 Steinbrunn et al. (1997); due to lack of space we omit the detailed performance analysis and the interested reader is referred to the aforementioned survey.

Figure 5 Distribution of acceptance probability for the simulated annealing method



6 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 (Halevy, 2001) because we are not currently considering the problem of integrating heterogeneous data.

6.1 Index structures for object models

There has been quite a lot of research on indexing object-orientated 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 (Shidlovsky and Bertino, 1996) This view directly corresponds to RDF data, that is also often 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 organise 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 (Bertino, 1991). In RDF as well as in object-orientated databases, the inheritance relationship 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 (Christophides et al., 2003). In the area of object-orientated database systems, these two kinds of indexing structures have been combined, resulting in the so-called nested inheritance indices (Bertino and Foscoli, 1995) and generalised nested inheritance indices (Shidlovsky and Bertino, 1996). These index structures directly represent the implications of inheritance reasoning, an approach that is equivalent to indexing the deductive closure of the model.

6.2 Query optimisation

There is a long tradition of work on distributed databases in general (Ozsu and Valduriez, 1991) and distributed query processing in particular (Kossmann, 2000). The dominant problem is the generation of an optimal query plan that reduces execution costs as much as possible while guaranteeing the completeness of the result. As described by Kossmann (2000), 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 optimisation which can be performed in a centralised manner by the mediator. While some restricted cases of this problem can be solved in a polynomial time (Moerkotte, 2003; Ono and Lohman, 1990), the general problem of finding an optimal plan for evaluating join queries has been proven to be NP-hard (Scheufele and Moerkotte, 1996). The approaches to tackle this problem can be split into several categories (Steinbrunn et al., 1997): deterministic algorithms, randomised algorithms, and genetic algorithms. Deterministic algorithms often use techniques of dynamic programming (e.g., Ono and Lohman (1990)), however, due to the complexity of the problem they introduce simplifications, which render them as heuristics. Randomised algorithms (e.g. (Swami and Gupta, 1988), 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. Stillger and Spiliopoulos, 1996) 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.

7 Discussion

In this paper, we laid the foundations for querying distributed RDF repositories by proposing a mediation architecture as well as index structures and algorithms for optimised querying distributed repositories without having to retrieve and store the data in a single model. We argued that besides being more natural w.r.t. semantic web applications, this model has advantages over a centralised approach in terms of flexibility, freshness and independence of data.

The querying in such RDF repositories often boils down to the computation of path expressions over the different RDF sources involved. An effective approach to distribution implies the need for optimisation techniques for the querying using these path expressions. In a way this is similar to certain approaches and techniques that we know from query optimisation in the context of databases. However, we have to take into account the specific context that we have with distributed RDF repositories. We have successfully attempted to apply the results from this research in the database area. An essential prerequisite to that is the meta-optimisation phase which fine-tunes the optimiser for concrete application specifics. We have defined our context by considering the issue of instance identity in RDF, the capabilities of RDF sources and the schema/query size. The characteristic aspect of our approach is the focus on optimising the order in which the path expressions are evaluated and in doing so we have relied on specific heuristics to tackle this inherently complex problem.

There are a couple of directions for future work. Firstly, we assume that the source index hierarchies are based on expanded RDF models. This is in line with the current design of Sesame where the expansion of a model is computed as soon as new information is stored using the expansion rules from Hayes (2002). The index structure can then be generated from this expanded model. Unfortunately, this is only possible if the different sources used for querying are actually RDF repositories. If we also want to use sources that do not have RDF inference capabilities, we have to perform reasoning in the mediator SAIL in order to make sure the implied results are also found. Another limitation of our approach is the restriction of the index to paths. As RDF queries are often tree-shaped, we sometimes need more than one source index hierarchy to locate the information relevant for a query. In future we will address the problem of coordinating different index structures, or alternatively of developing structures capable of indexing tree-shaped queries.

In terms of optimisation we want to experiment with join-capable sources, which would allow for true distributed optimisation including query execution parallelising and data transfer minimisation. We also envision a need for a general RDF (meta) optimisation framework where, depending on the context (hybrid source capabilities, heterogeneous bandwidth parameters, differences in URI referencing etc.), would choose from a pre-compiled optimisation library a sequence of optimisation procedures setting their parameters, e.g. the solution space and the appropriate join methods.

References

Bernstein, P.A. and Chiu, D.W. (1981) 'Using semi-joins to solve relational queries', *Journal of the ACM*, Vol. 28, pp.25–40.

Bertino, E. (1991) 'An indexing technique for object-oriented databases', in *Proceedings of the 7th International Conference on Data Engineering*, April 8–12, Kobe, Japan, IEEE Computer Society, pp.160–170.

Bertino, E. and Foscoli, P. (1995) 'Index organizations for object-oriented database systems', *TKDE*, Vol. 7, No. 2, pp.193–209.

Broekstra, J., Kampman, A. and van Harmelen, F. (2002) 'Sesame: a generic architecture for storing and querying rdf and rdf schema', in *The Semantic Web – ISWC 2002*, *Vol. 2342 of Lecture Notes in Computer Science*, Springer, pp.54–68.

Christophides, V., Plexousakisa, D., Scholl, M. and Tourtounis, S. (2003) 'On labeling schemes for the semantic web', *Proceedings of the 13th World Wide Web Conference*, pp.544–555.

Halevy (2001) 'A. answering queries using views – a survey', *The VLDB Journal*, Vol. 10, No. 4, pp.270–294.

Hayes, P. (2002) 'RDF model theory', *Working Draft, W3C*.

Hendler, J. (2001) 'Agents and the semantic web', *IEEE Intelligent Systems*, Vol. 2.

Hsiao, H., Chen, M. and Yu, P. (1997) 'Parallel execution of hash joins in parallel databases', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, pp.872–883.

Ioannidis, Y.E. and Wong, E. (1987) 'Query optimization by simulated annealing', *ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database Systems Archive*, ACM Press, pp.9–25.

Kossmann, D. (2000) 'The state of the art in distributed query processing', *ACM Computing Surveys*, Vol. 32, No. 4, pp.422–469.

Moerkotte, G. (2003) 'Constructing optimal bushy trees possibly containing cross products for order preserving joins is in p, tr-03-012', *Technical Report*, University of Mannheim.

Ono, K. and Lohman, G.M. (1990) 'Measuring the complexity of join enumeration in query optimization', *16th International Conference on Very Large Databases*, Morgan Kaufmann, pp.314–325.

Ozsu, M. and Valduriez, P. (1991) *Principles of Distributed Database Systems*, Prentice Hall.

Scheufele, W. and Moerkotte, G. (1996) 'Constructing optimal bushy processing trees for join queries is np-hard, tr-96-011', *Technical Report*, University of Mannheim.

Shidlovsky, B. and Bertino, E. (1996) 'A graphtheoretic approach to indexing in object-oriented databases', in S.Y.W. Su (Ed.) *Proceedings of the 12th International Conference on Data Engineering*, February 26–March 1, New Orleans, Louisiana, IEEE Computer Society, pp.230–237.

Steinbrunn, M., Moerkotte, G., et al. (1997) 'Heuristic and randomized optimization for join ordering problem', *The VLDB Journal*, Vol. 6, pp.191–208.

Stillger, M. and Spiliopoulou, M. (1996) 'Genetic programming in database query optimization', in J.R. Koza, D.E. Goldberg, D.B. Fogel and R.L. Riolo (Eds) *Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press, pp.388–393.

Swami, A. and Gupta, A. (1988) 'Optimization of large join queries', *ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database System*, ACM Press, pp.8–17.

Xie, Z. and Han, J. (1994) 'Join index hierarchies for supporting efficient navigations in object-oriented databases', in *Proceedings of the International Conference on Very Large Databases*, pp.522–533.

Yu, C. and Meng, W. (1998) *Principles of Database Query Processing for Advanced Applications*, Morgan Kaufmann Publishers.

Notes

[1] http://www.musicbrainz.org.

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

[3] At this point we ignore additional constraints such as equality and inequality constraints that are supported by some query languages.

[4] It is the case where all sources contain results for the complete schema path.

[5] Especially, the length of the path expression.

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

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

[8] The number corresponds to the sum of the n–1-th line in the Pascal triangle.

[9] The fragmentation is considered only partial as the two sets have a common element (S_3).

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