Adaptive Integration of Distributed Semantic Web Data

Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura

Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
{steven.lynden,a.matono,yusuke.tanimura}@aist.go.jp,
kojima@ni.aist.go.jp

Abstract. The use of RDF (Resource Description Framework) data is a cornerstone of the Semantic Web. RDF data embedded in Web pages may be indexed using semantic search engines, however, RDF data is often stored in databases, accessible via Web Services using the SPARQL query language for RDF, which form part of the Deep Web which is not accessible using search engines. This paper addresses the problem of effectively integrating RDF data stored in separate Web-accessible databases. An approach based on distributed query processing is described, where data from multiple repositories are used to construct partitioned tables that are integrated using an adaptive query processing technique supporting join reordering, which limits any reliance on statistics and metadata about SPARQL endpoints, as such information is often inaccurate or unavailable, but is required by existing systems supporting federated SPARQL queries. The approach presented extends existing approaches in this area by allowing tables to be added to the query plan while it is executing, and shows how an approach currently used within relational query processing can be applied to distributed SPARQL query processing. The approach is evaluated using a prototype implementation and potential applications are discussed.

1 Introduction

The Resource Description Framework (RDF) [1], published by the World Wide Web Consortium (W3C), is used to model information in order to support the exchange of knowledge on the Web. The Semantic Web [2] effort is expected to lead to an increasing amount of data being published using RDF. In some cases RDF is embedded in Web pages and sometimes it may be held privately, but often, RDF data is published in Web-accessible databases which clients can access using a query interface accepting SPARQL [3], a W3C Recommendation RDF query language, currently the most widely used method of querying RDF data. The term "Deep Web" [4] was coined to refer to the part of the web hidden from search engine indexes behind dynamically generated pages or query interfaces. Given the abundance of RDF repositories with querying interfaces,

S. Kikuchi, S. Sachdeva, and S. Bhalla (Eds.): DNIS 2010, LNCS 5999, pp. 174–193, 2010.

the Semantic Web also constitutes a Deep Web and there is a need to develop tools for accessing and integrating such data, especially in dynamic application domains such as news and weather reporting, social networks etc. As the number of RDF repositories and the volume of data they contain is increasing, it is anticipated that various applications can benefit from the integration of RDF data from multiple distributed RDF repositories.

The SPARQL language allows sets of triple patterns to be matched against RDF graphs, supported by various features such as conjunctions, disjunctions, filter expressions, optional triple patterns and multiple ways of representing query results. The SPARQL query language has an associated protocol, also a W3C Recommendation, the SPARQL Protocol for RDF [5], which defines an interface by which queries may be executed on a SPARQL data resource along with bindings for HTTP and SOAP. The purpose of the SPARQL protocol is to promote interoperability, where clients can interact with SPARQL data resources in a consistent way. For query results, another W3C Recommendation, SPARQL Query Results for XML [6], provides a way of encoding results from SPARQL queries. Many RDF data repositories use the SPARQL query language, SPARQL Protocol and XML results format in conjunction to provide an interface for clients. Examples include DBPedia [7], an extraction of structured information from Wikipedia, and the RDF-based instantiation of the DBLP computer science publication bibliography database [8]. The use of the SPARQL query language, protocol and result format in conjunction eliminates syntactic heterogeneity between different data sources allowing them to be accessed consistently regardless of the underlying RDF database implementation. Alternatives to the SPARQL protocol exist in the form of the Open Grid Forum (OGF) Data Access and Integration Service (DAIS) Working Group's specifications for accessing RDF data resources [9], which may eventually provide an alternative to the SPARQL protocol, and middleware support such as OGSA-DAI-RDF [10], however at present the SPARQL protocol is far more widely used.

Federated queries across multiple SPARQL endpoints allow data from such endpoints to be integrated, and is also of potential benefit in heterogeneous information systems where individual components may use SPARQL wrappers to expose data. Data integration in this context is made particularly appealing by the Linked Open Data Project [11], which aims to promote widespread usage of URI-based representations to allow RDF terms to be consistently defined. The use of consistent URIs to represent the same terms in different databases means that joining data across two data sources is possible, therefore federated queries over multiple repositories can provide results that are not obtainable from any one individual repository.

Existing systems for integrating RDF data from distributed SPARQL endpoints generally rely on the availability of statistics about the data which are then used by an optimiser to compute a join order. This works well when the optimiser has accurate statistics about the data present in each of the repositories, allowing it to minimise the size of the data retrieved from each endpoint and

effectively optimise joins and other operations. In contrast this paper focuses on an adaptive approach that can respond to the characteristics of the data and SPARQL endpoints from which the data is retrieved (for example join predicate selectivity, rate at which the data is produced by the service etc.) while the data is being integrated.

The framework described compiles a federated SPARQL query into a number of source queries which are sent to individual SPARQL endpoints to retrieve the data required to answer the query. The results from source queries are used to construct a set of vertically partitioned RDF tables, which act as a temporary buffer to hold data while it is integrated. The vertical partitioning of RDF triples into relational predicate tables (one table for each predicate; subject and object values as the two columns of each table) has been shown to be effective in [12]. The vertically partitioned tables are processed by an extension of the adaptive query processing techniques presented in [13], which allows join order to change during query processing. This approach is adopted for optimising queries that perform time-consuming joins between multiple RDF data sources by dynamically building a query plan that can adapt to the characteristics of the data as the query is being processed. Although the technique presented uses relational database processing techniques to join the predicate tables, the approach in general aims to address challenges specific to integrating data from RDF repositories, where SPARQL endpoints are autonomous and managed by individuals resulting in unpredictability and a lack of accurate statistics about the data, meaning that adaptive query processing techniques can provide a significant benefit. In addition to this, adaptive approaches also have the potential to perform better in unpredictable environments, which is the case with integrating data on a large scale. For example, data sources may be busy or temporarily unavailable and therefore process queries more slowly than anticipated, and some endpoints may vary with respect to their support for specific features of the SPARQL query language and their efficiency in supporting those features.

As the basic data model for knowledge representation on the Semantic Web, RDF is currently becoming widely adopted. RDF is being used by various different individuals and organisations to publish information and a number of publicly available RDF databases contain millions of triples [1]. RDF forms the basis of other Semantic Web components such as RDF Schema (RDFS) and The Web Ontology Language (OWL). Together these components have the potential to provide an interoperable description of information that can be interpreted unambiguously and processed by automated reasoning and inference systems. Community efforts, such as the the Linked Data project aim to promote the sharing of data using RDF, and furthermore, RDF is being used by some publishers to offer dynamic content, for example BBC Backstage [2], which publishes various media in RDF, for example information about TV and radio programmes. All this means that publicly available RDF data is constantly being updated and

[1] The Uniprot protein database [http://www.uniprot.org], DBPedia and the Linked Open Data project are examples.

[2] BBC Backstage [http://ideas.welcomebackstage.com/data]

growing rapidly, presenting a significant challenge to the developers of applications that need to access this data. Furthermore, one of the key challenges related to the proliferation of RDF data is that it is widely distributed. A distributed RDF query processor addresses this issue by providing a technique for querying multiple RDF repositories as if querying a single repository using SPARQL. In particular, the use of adaptive query processing techniques is useful in scenarios where the statistics about the contents of the repositories are incomplete or unavailable, the data is constantly updated, and joins need to be performed between data in different repositories.

The technique described in this paper has been used to develop a federated SPARQL query processing interface, the Adaptive Distributed Endpoint RDF Integration System (ADERIS) [14], which allows users to compose SPARQL queries and execute them over multiple SPARQL services. This application is suitable for users with a knowledge of SPARQL allowing them to compose queries, however the approach is also useful for developing other applications where queries are composed automatically and hidden from the user behind the scenes.

2 Related Work

As SPARQL is relatively new, having become a W3C recommendation in 2008, work focusing on distributed data integration over SPARQL endpoints is at an early stage. Work in the wider area of distributed RDF query processing can be roughly divided into the following categories:

1. Search engine based approaches which aim to index a large number of individual documents containing Semantic Web data, for example Swoogle [15] and YARS2 [16].
2. Top-down approaches where an RDF data set is partitioned into smaller subsets which are processed in parallel. Examples of such approaches include decomposition using RDF molecules [17], data partitioning and placement strategies for parallel RDF query processing [18] and the use of peer-to-peer/distributed hash table based approaches for distributing query processing over a set of peers [19].
3. Mediator-based approaches where data sets from multiple autonomous endpoints are combined together by the mediator, which optimises a federated SPARQL query, providing transparent access to the individual endpoints as if they were a single RDF graph.

Work related to (3) is discussed here, where SPARQL is used as the query language and protocol via which data sources are accessed.

Firstly, although not a comprehensive distributed query processing solution, it is worth mentioning that ARQ [20] (the SPARQL query processor for the Jena [21] framework for developing semantic web applications) provides query language extensions for executing remote queries, extending SPARQL with a "SERVICE" construct which forwards triple patterns to a remote endpoint.

ARQ is extended by DARQ [22] (Distributed ARQ), a comprehensive RDF distributed query processor capable of parsing, planning, optimising and executing queries over multiple SPARQL endpoints. When using DARQ, it is not necessary to refer to named graphs or specify anything other than a standard declarative SPARQL query - DARQ parses the query, determines which data sources should be queried and optimises the process of retrieving and integrating data from individual data sources. The system optimises queries based on information about individual data sources provided by *service descriptions*, a metadata format introduced in order to describe an RDF data source. Service descriptions provide the DARQ optimiser with information such as predicate selectivity values and other statistics, and are utilised during the generation of source queries and join ordering, which is combined with physical optimisation implemented using iterative dynamic programming.

FeDeRate [23] is a system for executing distributed queries over multiple data sources supporting a variety of different interfaces, focusing on applications in the domain of bioinformatics. Queries are submitted to FeDeRate in SPARQL, mapped to a set of source queries which are sent to the individual data sources, the results of which are combined and returned as the result of the federated query. FeDeRate provides support for distributed queries with named graph patterns using SPARQL's syntactic support for queries over multiple named graphs. Multiple named graphs may be referred to in a query, each of which is accessed in the order that they appear in the query (as is the case with ARQ). FeDeRate aims to minimise query result sizes in order to more efficiently execute queries by using results from previously executed source queries as constant values in subsequent queries. Optimisation such as query re-writing and join ordering are not performed.

SemWIQ [24] is another system implemented using ARQ that offers a similar approach to DARQ, supporting RDF distributed queries with an optimiser that uses statistics about endpoints obtained by a monitoring component that issues SPARQL queries in order to generate statistics. In contrast to DARQ, SemWIQ is able to support queries over endpoints for which DARQ's statistics and metadata can not be obtained due to restrictions of autonomy or privacy etc., which is also the aim of the work presented in this paper. The monitoring component, RDF-Stats [25], available as a separate component to the distributed query processor, uses an extended SPARQL syntax supported by many SPARQL endpoints which allows the aggregate construct COUNT. The authors state that the monitoring component pulls a large amount of data from data sources and should therefore be installed close (or ideally on the same cluster/node) as the data source it is monitoring. SemWIQ implements various query optimisation strategies such as push-down of filter expressions, push down of optional group patterns, push-down of joins and join and union reordering. However, as is the case with DARQ, optimisation is done statically and requires statistics about data sources.

The work presented in this paper differs from the approaches discussed in this section in the sense that adaptive, rather than static optimisation, is used.

Furthermore, the approach presented in this paper focuses on efficiently executing a specific kind of query, that of ordering multiple joins with large input sizes. In practice, the adaptive approach could be used in conjunction with the optimisation techniques implemented by systems such as DARQ and SemWIQ.

3 Federated SPARQL Query Processing Framework

The framework is based on a mediator that accepts a federated SPARQL query, decomposes the query into a set of source queries and adaptively processes the results. The system's behavior is divided into two phases:

- Setup phase: the mediator is initialised with a list of SPARQL endpoints over which queries are to be executed in the next phase.
- Query processing: federated queries are accepted and executed by the mediator.

3.1 Setup Phase

To efficiently generate source queries, some metadata is required about each RDF repository. As one of the key assumptions made in this work is that repositories are autonomous and it may be difficult to retrieve detailed metadata and statistics, the metadata used is restricted to information obtainable via a straightforward SPARQL query. As a minimum requirement, the mediator requires requires knowledge of whether a particular predicate is known to be present or absent in a given data source. During the initialisation of the system, a list of SPARQL endpoint URIs is passed to the mediator, which submits the following SPARQL query to each endpoint:

```
SELECT DISTINCT ?p WHERE { ?s ?p ?o }
```

Regarding RDF data and SPARQL queries, in general the following observations made in [18] hold true:

- The number of distinct predicate values is much less than the number of distinct subjects or objects.
- In SPARQL queries, predicates are usually specified as query constraints, whereas subjects and objects are more likely to be variables.

These properties are exploited later during querying, and also here during initialisation, as the number of distinct predicate terms tends to be much smaller than subjects and objects, the above query can usually be executed in a reasonable amount of time by most endpoints. In cases where the query cannot be executed due to limits on query processing time or result sizes, an alternative method such as the RDF-Stats tool for generating statistics about RDF data can be used.

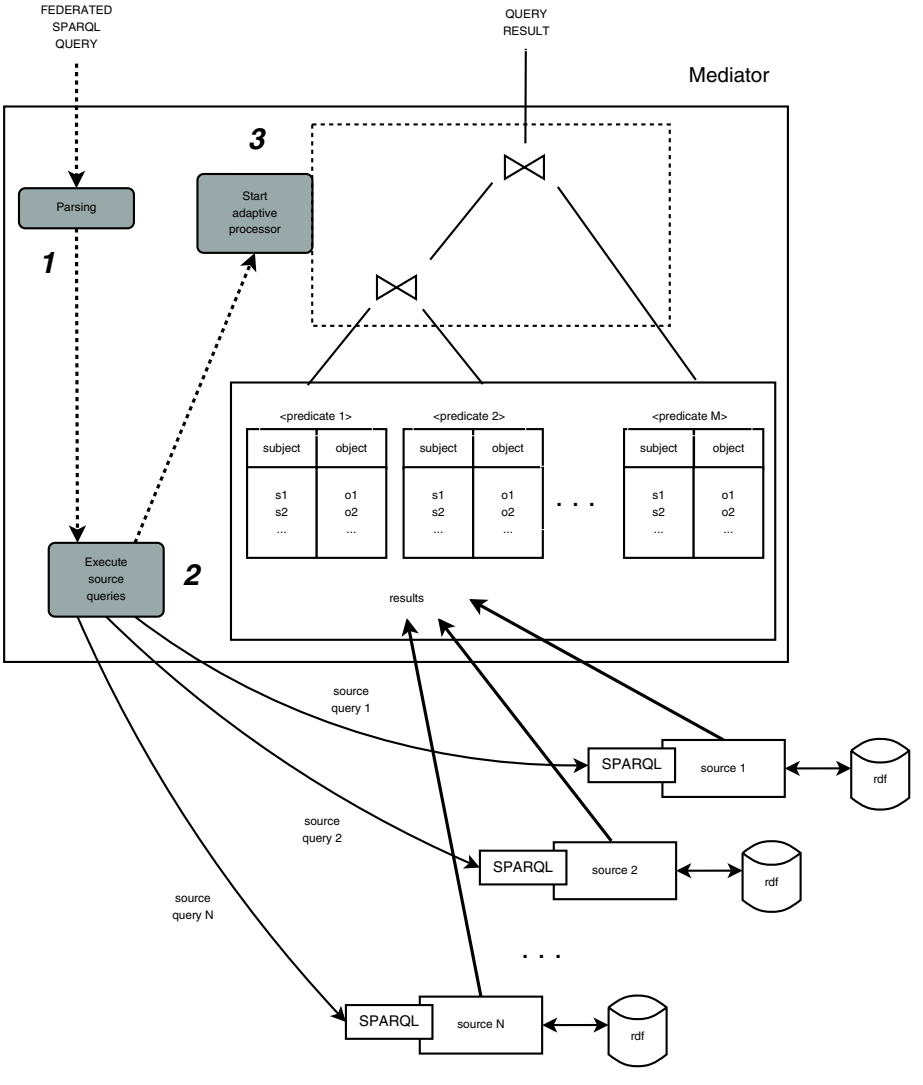

Fig. 1. Query processing framework

This figure illustrates the steps performed during query processing. In (1) the federated SPARQL query is parsed and a set of source queries over N data sources are compiled which are then executed in (2). An adaptive optimiser/evaluator is then started (3) which joins the M predicate tables, constructed using source query results, to answer the query.

3.2 Query Processing

Query processing, illustrated in Figure 1, consists of the following three steps:

1. Source query generation: the query is parsed and the data sources that need to be utilised in order to answer the query are determined. Source queries are constructed for each of these data sources, with the aim of retrieving all the data needed to answer the query while minimising the execution time of each query.
2. Execution of source queries and construction of predicate tables: source queries are sent to the SPARQL endpoints and the results are used to construct the predicate tables. Each triple returned from a query is placed, according to its predicate value, in the appropriate table.
3. Adaptive join processing: the predicate tables are joined dynamically, additionally any other operations in the federated SPARQL query, such as FILTER predicates that could not be pushed down to source queries, are applied here.

Steps 2 and 3 may overlap and take place concurrently, for instance, source queries may still be executing while some predicate tables have been constructed and are being joined. Each of the above steps is described in more detail below. The following example federated query will be referred in order to illustrate the process.

3.3 Example Federated SPARQL Query

To illustrate the approach, an example federated query over four data sources is used. The data sources contain triples from the Friend of a Friend (FOAF) [26] and DBPedia ontologies, distributed among data sources as follows.
Data source S1: Contains triples predicates `foaf:name` or `foaf:homepage`, i.e. triples of the following form exist in the data source:

```
subject <http://xmlns.com/foaf/0.1/name> object
subject <http://xmlns.com/foaf/0.1/homepage> object
```

Data source S2: like S1, also contains only triples with predicate values `foaf:name` and `foaf:homepage` and no triples where the predicate is `foaf:depiction` or `dbpedia:occupation`. Data source S3: contains triples where the predicate is `foaf:depiction` only. Data source S4: contains triples where the predicate `dbpedia:occupation` only.
Each of the above data sources exposes its data via a separate SPARQL endpoint. The following query, which fetches the name, URL and image properties associated with all entities classified as musicians, is executed over the data sources:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1>
PREFIX dbpedia: <http://dbpedia.org/property>
SELECT ?name ?url ?img WHERE {
   ?p foaf:name ?name .
```

```
   ?p foaf:homepage ?url .
   ?p foaf:depiction ?img .
   ?p dbpedia:occupation ?occupation
   filter (?occupation=<dbpedia:Musician>)
}
```

3.4 Source Query Generation

The generation of source queries, sent to remote endpoints to retrieve data needed to execute the federated query, is the only point at which the system requires metadata, collected in the setup phase, about the SPARQL endpoints being queried. For each data source, a source query is generated which contains all of the triple patterns from the federated query that could possibly match triples in the given data source, pushing down any filter expression predicates and joins where possible. For the example query, the source queries generated for data sources S1 & S2 are:

```
SELECT ?p ?o1 ?o2 WHERE {
  ?p foaf:homepage ?o1
  ?p foaf:name ?o2
}
```

The above query pushes down a part of the join between the triple patterns involving `foaf:name` and `foaf:homepage` to data sources S1 and S2 because these data sources contain triples with both of these predicate values. The source query for S3 retrieves all triples with the predicate value `foaf:depiction` as no filter predicate or join can be pushed down to this data source:

```
SELECT ?p ?o WHERE {
   ?p foaf:depiction ?o
}
```

Finally, the source query for S4 pushes down one filter predicate as follows:

```
SELECT ?p ?o WHERE {
   ?p dbpedia:occupation ?o
   FILTER (?o=<dbpedia:Musician>)
}
```

The metadata provides a mapping from predicates to data sources allowing source queries such as those exemplified above to be constructed, retrieving from each data source only triples with predicate values that are needed to evaluate the query.

3.5 Construction of Predicate Tables

Each source query produces a stream of result triples used to construct a set of predicate tables, where a table exists for each unique predicate value contained in the results produced by the source queries. Each predicate table possess two columns (subject and object) and maintains an index on any column that can be potentially used as a join predicate during the subsequent phase when the

tables are joined to answer the query. A predicate table is complete when all source queries that can possibly produce triples to be inserted into the given table have finished. In the example introduced in the previous section, the four predicate tables in Figure 2 are created.

In the work presented in this paper, predicate tables are stored in main memory, but they could potentially be written to a file system if necessary. To provide true scalability, a distributed file system could be used to store the tables over multiple nodes and parallelise the process of sending source queries and generating predicate tables from the results. Further performance enhancements are also possible, for example, caching predicate tables used in previous queries in order to speed up subsequent queries may be possible in some scenarios.

dbpedia:Occupation		Source query:
Subject	Object	S4
dbpedia:Beck	dbpedia:Musician	(sample row)
...	...	

foaf:Depiction		Source query:
Subject	Object	S3
dbpedia:Batman	http://../batman.png	
...	...	

foaf:Homepage		Source query:
Subject	Object	S1 and S2
dbpedia:Beck	www.beck.com	
...	...	

foaf:Name		Source query:
Subject	Object	S1 and S2
dbpedia:Ian_Rankin	foaf:Ian_Rankin	
...	...	

Fig. 2. Example predicate tables created by the source queries in Section 3.4

3.6 Adaptive Join Processing

Producing query results involves computing a set of joins between the constructed predicate tables. Here, all joins are index nested loop joins that consume each tuple from their left input, use an index on join attributes to lookup matching tuples from the right input and output joined tuples to the next operator in the plan. [13] presents a technique for reordering pipelined index nested loop join-based query plans where the notion of *depleted state* is introduced to encapsulate the moment in which a sequence of joins is in a state whereby the join order can be changed without throwing away results that have already been produced. Here, this technique is applied when joining the predicate tables as it allows the joins to be reordered based on run-time selectivity statistics (which

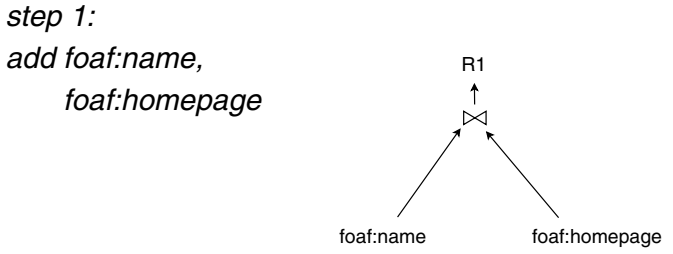

Fig. 3. Adaptive join processing is initialised when two tables become complete. This plan can be executed while the other predicate tables are being generated.

initially have to be roughly estimated). As some predicate tables become complete before others, it is necessary to extend the reordering approach in a way that allows the processing of joins between tables as soon as they become complete rather than being idle until the entire set of query queries has finished. When joins are executed, the selectivity of each join predicate is monitored so that subsequent optimisation can use accurate selectivity values to find optimal join orders. Monitoring is implemented within each join operator by recording the number of input and output tuples processed by the operator. This whole process takes place dynamically as the predicate tables constructed from source query results become available.

The approach is explained using the ongoing example, the query processing of which is illustrated in figures 3, 4 and 5. Initially, consider a state in which source queries S1 and S2 are the first ones to finish. This means that the predicate tables `foaf:name` and `foaf:homepage` are complete and may be joined, as illustrated by Step 1 in Figure 3. The optimiser chooses to join with `foaf:name` as the left input and use the index on `foaf:homepage` to execute the join. Following this, source query S4 finishes and the `dbpedia:occupation` table is complete. At this point `foaf:homepage` has only been used as the probed table (this is the the right input table to the join from which tuples are retrieved using the index); some rows from `foaf:name` have been consumed by the join and some joined tuples have been output (`R1`). In order to avoid losing the joined result, the optimiser creates two sub-plans as illustrated by Step 2 in Figure 4; the first sub-plan has the result produced in Step 1, `R1`, as the left input table to a join with `dbpedia:occupation`; the second sub-plan uses the unprocessed part of `foaf:name` (the part of this table that wasn't consumed by the join in Step 1), referred to as `foaf:name(*)`, and the optimiser is able to fully reorder this plan based on the selectivity statistics gathered as the joins are executed. Both sub-plans access the `dbpedia:occupation` table simultaneously; the first sub-plan using the index to probe the table, the second plan either using it as a probed input or possibly a left input to the first join, as is the case in Step 3 in Figure 5. This can be achieved by keeping a separate set of pointers for each sub-plan to determine which rows to read.

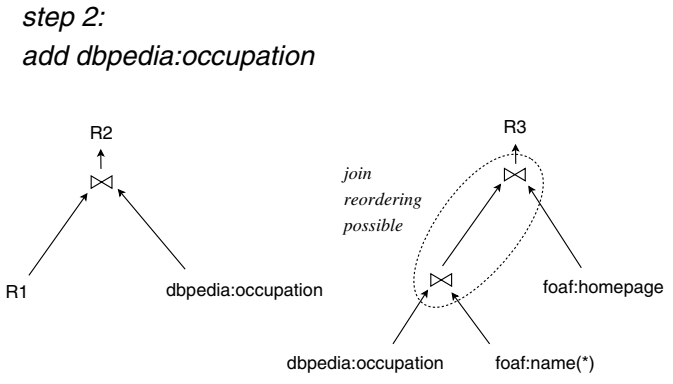

Fig. 4. A third table is added during adaptive join processing. Two independent plans are formed that will be integrated when all predicate tables are complete. The join order of the plan generating R3 may be changed (at any time, independently of the other plan) in response to monitored join selectivities. The other plan's join order cannot be changed because there is no index on the intermediate results R1 that was generated in the previous step.

When the remaining source query, S3, has finished, all predicate tables are now complete and the plan can be finalised. As two independent query sub-plans have been created in Step 2, this must take place when both plans are in states where reordering is possible (a 'depleted state' as defined in [13]), which can be achieved by pausing the first sub-plan to reach this state after the completion of S3 and waiting for the next sub-plan to enter this state. At this point the final plan may be constructed, which consists of the results from the two sub-plans created in Step 2 joined with the now-available `foaf:depication` table, and additionally a sub-plan processing the non-consumed parts of each of the four tables. It should be noted that the plan in Step 2 that produces `R2` needs to be executed until is has completely consumed its input from `R1`. The plans that produce `R1` (in Step 1) and `R3` (in Step 2) are discarded once the next step is reached as they do not read tuples from a cache operator, and in these cases only the cache needs to be preserved. Each of the sub-plans in Step 3 are combined by union operators to produce the query result as no new predicate tables need to be added. Processing joins as described here has the following advantages:

- Nothing is known about the selectivity of join predicates before the query is executed so it is difficult to produce a statically optimised query plan. Using selectivity monitoring and join reordering alleviates the need to produce a statically optimised plan.
- The predicate tables are processed as they become available so there is no need to wait until all source queries have finished before executing the query. This has the potential to reduce query response time in certain cases.

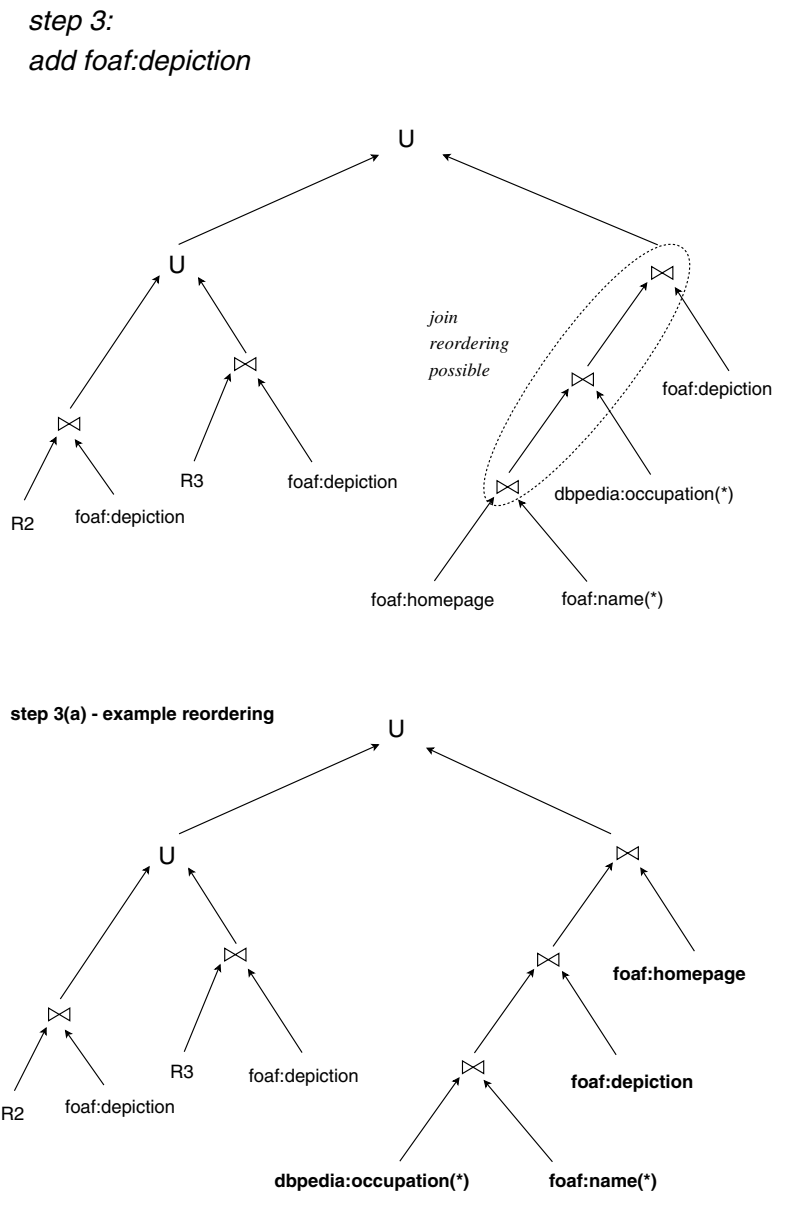

Fig. 5. The final table is added during adaptive join processing. Step 3 (a) shows an example reordering which can take place if join predicate selectivity monitoring shows that estimated selectivity is substantially different from the monitored selectivity. The reordered elements of the plan are highlighted in bold. Note that sub-plans can be reordered independently of other sub-plans.

– Predicate tables that have been joined can potentially be discarded. Compared to waiting until all predicate tables become complete, this can be beneficial in terms of memory usage when processing large amounts of data.

4 Performance Analysis

A prototype query processor has been implemented in Java, currently supporting only SPARQL SELECT queries with subject/object variables (predicates must be constrained) and without various language features such as support for the OPTIONAL construct. The prototype does however allow for an evaluation of the approach based on the execution of a simple federated query over multiple SPARQL endpoints.

4.1 Data Sources and Queries

An evaluation is made using QUERY-1:

```
PREFIX dbpedia: <http://dbpedia.org/property>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>
PREFIX foaf: <http://xmlns.com/foaf/0.1>
PREFIX skos: <http://www.w3.org/2004/02/skos/core>
SELECT ?ref ?comment ?page ?subj WHERE {
   ?obj dbpedia:reference ?ref .
   ?obj rdfs:comment ?comment .
   ?obj foaf:page ?page .
   ?obj skos:subject ?subj
}
```

This federated SPARQL query combines data from four separate RDF data sources, each of which contains data from one of the following four DBPedia data sets which are available for download individually from the DBPedia site:

1. The “External Links” data set containing triples with the predicate term `dbpedia:reference`.
2. The “Short Abstracts” data set containing triples with the predicate term `rdfs:comment`.
3. The “Links to Wikipedia Article” data set containing triples with the predicate term `foaf:page`.
4. The “Article Categories” data set containing triples with the predicate term `skos:subject`.

The SPARQL endpoints are constructed by downloading the files from DBPedia and using Joseki [27] to provide a SPARQL front end. Each endpoint is located on a separate (3GHz Intel Xeon) machine with 1MB available memory for the Java Virtual Machine providing the endpoint. Endpoint machines are connected to the machine on which the mediator is deployed (2GHz AMD Athlon X2, 2GB RAM) via a 100Mbs Ethernet LAN. The experiments compare the following four query processing strategies:

- *wait no-adapt*: The system does not allow join reordering (no-adapt) and the joins are not processed until all source queries have finished and the predicate tables are complete (wait).
- *no-wait no-adapt*: The system does not allow join reordering (no-adapt) but as soon as predicate tables are complete they may be joined if possible (no-wait).
- *wait adapt*: Join reordering is allowed (adapt), but the plan is not executed until all source queries have finished and the predicate tables are complete (wait).
- *no-wait adapt*: The full application of the technique presented in this paper; join reordering is allowed (adapt) and predicate tables may be joined as soon as they become available (no-wait).

Distributed query processing over autonomous data sources can be complicated by the unpredictability of the data sources and the communication channels between them. Data sources may sometimes be slow to respond if demand from multiple clients is high or maintenance/updates to the data are taking place. In some cases, queries may fail completely if data sources are temporarily unavailable or a network problem occurs when communicating with the data source. There may also be differences between the underlying implementation used by different data sources, for example, indexes used and support or lack of support for different aspects of the SPARQL query language. Adaptive query processing can help to mitigate the effects of such unpredictability by responding to different data source response times while a federated query is being processed. To model such an environment, source query failures are incorporated into the experiments. Source query failures are modeled as follows: for each federated query processed by the system, one of the source queries (randomly selected) encounters a fatal problem necessitating the source query to be executed again. This behaviour is introduced with the aim of simulating real-life situations where a SPARQL service is busy and cannot respond (temporarily unavailable) or a service is down and data must instead be retrieved from an alternative (replicated) service. Two experiments are performed, as described below.

Experiment 1. The first experiment investigates the performance with respect to scalability by varying the size of the data sets. Subsets of the complete data sets are used to execute the experiments with each data source containing the same number of triples, starting with each data source containing 100,000 triples. The experiment is repeated for larger data set sizes in increments of 100,000 triples until all data sources contain 600,000 triples.

Experiment 2. The second experiment introduces FILTER expressions into QUERY-1 to evaluate performance where predicates are evaluated by each of the data sources - this results in variance of the size of the data set returned by each data source and the amount of time taken by each data source to answer source queries. Randomly selected letters or common string patterns such as ‘the’, ‘as’, ‘in’ etc. are inserted into REGEX (regular expression) functions

associated with each triple pattern. In any randomly generated query, for each triple pattern there is a 0.25 probability of inserting a randomly selected word. For example, a randomly generated query could be:

```
PREFIX dbpedia: <http://dbpedia.org/property>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>
PREFIX foaf: <http://xmlns.com/foaf/0.1>
PREFIX skos: <http://www.w3.org/2004/02/skos/core>
SELECT ?obj ?ref ?comment ?page ?subj WHERE {
   ?obj dbpedia:reference ?ref .
   ?obj rdfs:comment ?comment .
   ?obj foaf:page ?page . FILTER regex(str(?page),'in')
   ?obj skos:subject ?subj
}
```

Inserting predicates into the query in this manner introduces increases the variance of the processing time per result triple for source queries (since REGEX functions may need to be implemented by data sources), the size of the result set returned from each data source, and hence, the size of the federated query result set and overall query processing time. 20 queries were generated randomly using the approach described, and for each generated query, the response times of the four different query processing strategies were compared.

4.2 Results

Response times for Experiment 1 are shown in Figure 6. It can be seen that the no-wait adapt strategy provides the fastest response times, in particular with large data sizes. Response times for Experiment 2 are shown in Figure 7, where each item on the x-axis corresponds to a randomly generated query (the order in which the queries appear is not significant and comparisons should be made between the four different response times of the different strategies within the same query only). Again, the no-wait adapt strategy generally performs better when compared to the other strategies, in particular when overall query processing times are relatively high. As with Experiment 1, results show some advantage of the two adaptive techniques (wait adapt and no-wait adapt) over the non-adaptive techniques, in particular when query response times are high. As the no-wait adapt strategy appears to perform best when the number of triples is large, e.g. when each data source contains 600k triples in in Figure 6 where it has a clear advantage over the non-adaptive strategies, future work will involve performing experiments with larger data sets in order to confirm that this observed trend continues as join input sizes grow. For low query response times, corresponding to queries that involve relatively small volumes of data being returned from source queries, there is little difference between the four strategies. Where the adaptive strategies perform worse than the non-adaptive strategies, this can be accounted for by the overhead in adapting, including monitoring for states at which reordering is possible and starting/stopping the executing plan.

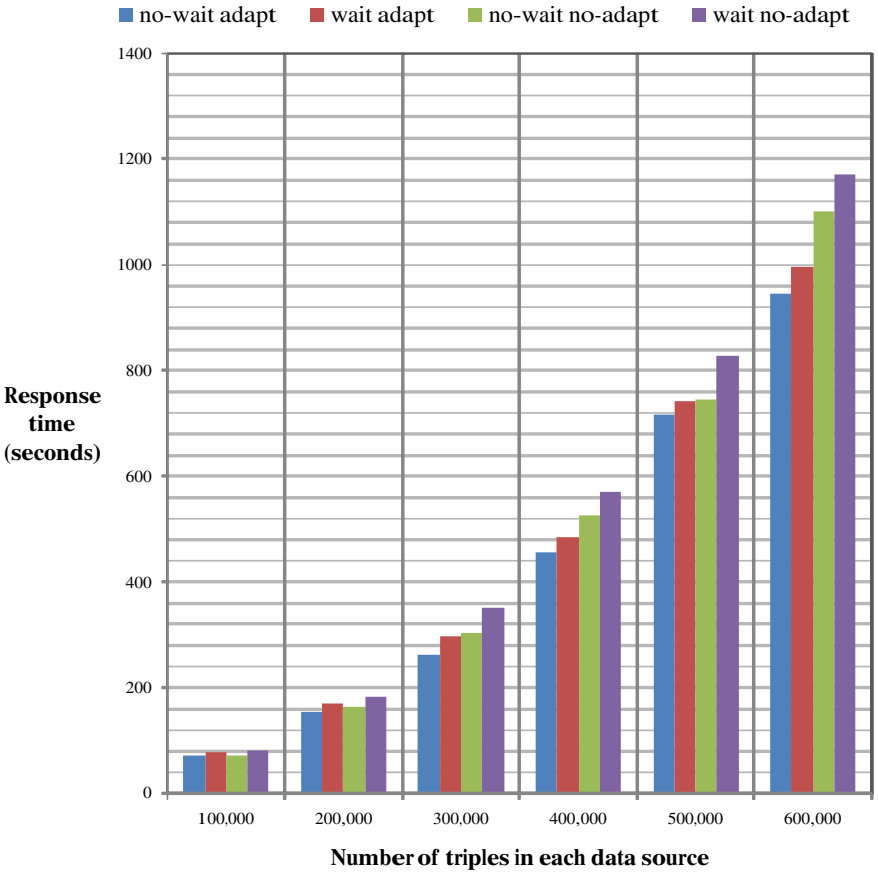

Fig. 6. Experiment 1 response times

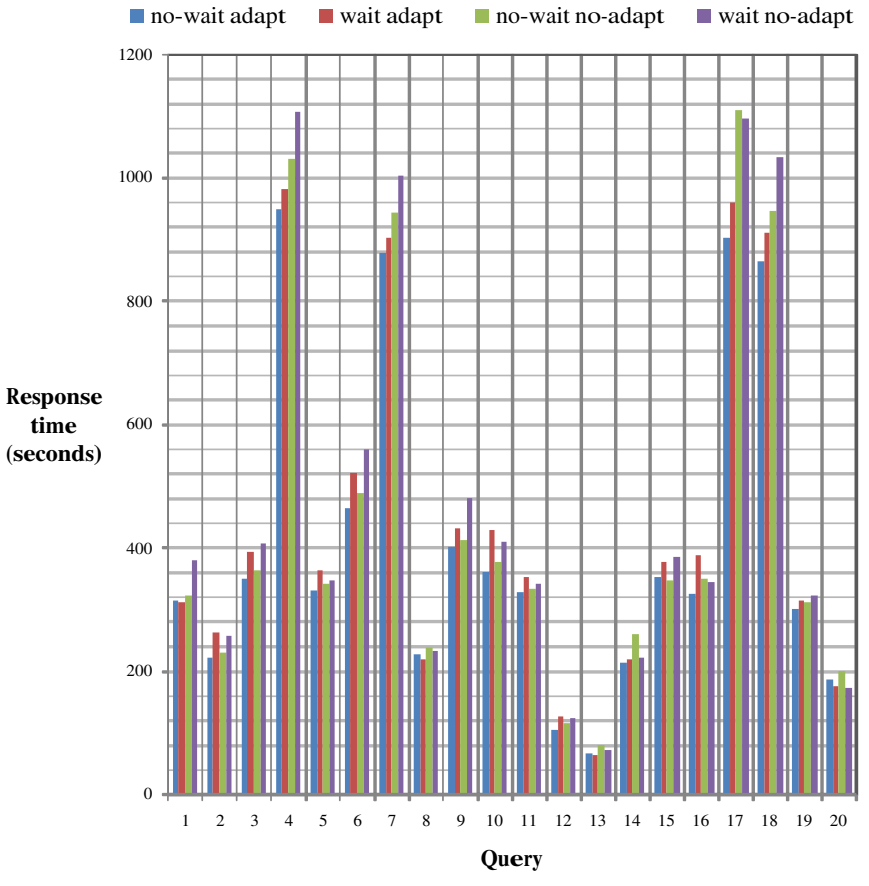

Fig. 7. Experiment 2 response times

5 Conclusions

An adaptive framework has been presented for executing queries over multiple SPARQL endpoints that differs from existing approaches which use static query optimisation techniques. Many SPARQL web services are currently available and the number of them is growing. The work presented in this paper is a framework for executing queries over federations of such services. The framework proposed in this paper, which allows adaptive query processing over dynamically constructed predicate tables to be performed in conjunction with the construction of the predicate tables, was shown to perform relatively well in unpredictable environments where source query failures may occur. The prototype implemented was evaluated using real data, showing some advantage in terms of response times of adaptive over non-adaptive methods using a subset of DBPedia. Future work will aim to investigate other data sets with different characteristics and

larger data sets. As the approach presented in this paper focuses on efficiently executing a specific kind of query, that of adaptively ordering multiple joins, further work will focus on optimising other kinds of queries and implementing support for more SPARQL query language features. Future work will also concentrate on investigating how the work can be applied in various domains.

Acknowledgments

This work was supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications (MIC), Japan.

References

1. Klyne, G., Carroll, J.J.: Resource description framework (rdf): Concepts and abstract syntax. Technical report, W3C (2004)
2. Berners-Lee, T.H., Lassila, O.J.: The Semantic Web. Scientific American 284(5), 28–37 (2001)
3. Eric Prudhommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C (2008)
4. Bergman, M.K.: The Deep Web: Surfacing Hidden Value. The Journal of Electronic Publishing 7 (2001)
5. Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF. Technical report, W3C (2008)
6. Beckett, D., Broekstra, J.: SPARQL Query Rresults XML Format. Technical report, W3C (2008)
7. DBPedia, `http://dbpedia.org/`
8. D2R Server publishing the DBLP Bibliography Database, `http://www4.wiwiss.fu-berlin.de/dblp/`
9. Gutiérrez, M.E., Kojima, I., Pahlevi, S.M., Corcho, Ó., Gómez-Pérez, A.: Accessing RDF(S) data resources in service-based grid infrastructures. Concurrency and Compututation: Practice and Experience 21(8), 1029–1051 (2009)
10. Kojima, I., Kimoto, M.: Implementation of a Service-based Grid Middleware for Accessing RDF Databases. In: Proceedings of Workshop on Semantic Extensions to Middleware: Enabling Large Scale Knowledge Applications (SEMELS 2009) (November 2009)
11. Linked Data - Connect Distributed Data across the Web, `http://linkeddata.org/`
12. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J 18(2), 385–406 (2009)
13. Li, Q., Sha, M., Markl, V., Beyer, K., Colby, L., Lohman, G.: Adaptively Reordering Joins during Query Execution. In: Proc. ICDE, pp. 26–35. IEEE Computer Society, Los Alamitos (2007)
14. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: Adaptively integrating RDF data from SPARQL endpoints (Demo Paper). In: Proceedings of the Database Systems for Advanced Applications (DASFAA) Conference 2010 (2010) (to appear)
15. Ding, L., Finin, T., Joshi, A., Peng, Y., Cost, R.S., Sachs, J., Pang, R., Reddivari, P., Doshi, V.: Swoogle: A Semantic Web Search And Metadata Engine. In: 13th ACM Conference on Information and Knowledge Management (2004)

16. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., et al. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)
17. Newman, A., Li, Y.-F., Hunter, J.: Scalable Semantics, The Silver Lining of Cloud Computing. In: 4th IEEE International Conference on e-Science (e-Science 2008) (2008)
18. Tanimura, Y., Matono, A., Kojima, I., Sekiguchi, S.: Storage Scheme for Parallel RDF Database Processing Using Distributed File System and MapReduce. In: International Conference on High Performance Computing in the Asia Pacific Region (2009)
19. Liarou, E., Idreos, S., Koubarakis, M.: Continuous RDF Query Processing over DHTs. In: International Conference Semantic Web Computing (2007), http://iswc2007.semanticweb.org/papers/323.pdf
20. ARQ SPARQL query processing framework, http://jena.sourceforge.net/ARQ/
21. Carroll, J.J., Dickinson, I., Dollin, C., Seaborne, A., Wilkinson, K., Reynolds, D., Reynolds, D.: Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories (2004)
22. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021. Springer, Heidelberg (2008)
23. Prudhommeaux, E.: Optimal RDF access to relational databases. Technical report, W3C (2005), http://www.w3.org/2004/04/30-RDF-RDB-access/
24. Langegger, A., Woss, A., Bloch, W.: A Semantic Web Middleware for Virtual Data Integration on the Web. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021. Springer, Heidelberg (2008)
25. RDFStats home (subproject of Semantic Web Integrator and Query Engine), http://semwiq.faw.uni-linz.ac.at/rdfstats/
26. The Friend of a Friend (FOAF) Project, http://www.foaf-project.org/
27. JOSEKI - A SPARQL Server for Jena, http://www.joseki.org/