PLANNING AND EVALUATION OF FEDERATED QUERIES ON THE WEB

By

Gregory Todd Williams

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

James A. Hendler, Thesis Adviser

Tim Berners-Lee, Member

Peter Fox, Member

Sibel Adalı, Member

Rensselaer Polytechnic Institute
Troy, New York

February 2013
(For Graduation May 2013)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Web of Data continues to increase in size and diversity, providing access to large amounts of structured, linked data. However, existing approaches to querying this data often fail to make use of existing database access points and must resort to web crawling to collect data of interest. Furthermore, in order to provide efficient query answering over this data, existing systems are forced to construct centralized database indexes, making it difficult to maintain up-to-date data. For approaches that do utilize existing databases, disregard for fundamental design principles of the Web results in query systems that lack some basic features of their web crawling counterparts. If an efficient query answering system can be provided that does not require centralized indexing, and leverages both existing databases and static web content, users may benefit from up-to-date access to structured, disparate data.

In this dissertation, we develop a federated query planning framework based on the RDF data model and the SPARQL query language. This framework is able to leverage the high performance of existing SPARQL databases while also providing access to linked data available as RDF documents on the web. These two access methods are used to provide a single interface to querying semantic data.

The primary challenge of evaluating queries over both SPARQL databases and linked data is in finding an efficient execution plan. Such a plan must perform better than the naïve approach of completely decomposing the query and executing each subquery against each data source or traversing linked data by web crawling. Moreover, it must allow metadata discovered during query execution to be incorporated into the existing plan.

Given this, in this dissertation, we develop three techniques to increase performance and flexibility of federated query evaluation: we develop a federated query planning algorithm that prioritizes the execution of subqueries that have high expected value (that is, expected relevant results with low latency); we develop a *re-planning* algorithm, able to augment an existing query plan with newly discovered data sources and a mechanism for discovering such sources; and we develop

a server-side technique to greatly enhance the web cacheability of SPARQL query results.

Finally, the developed framework is designed using a traditional query planner, allowing it to integrate with and benefit from existing work on query planning and optimization.

To demonstrate the practicality of this federated query planning framework, we present results of empirical evaluation of the framework components over a real-world dataset of bibliographic data. These results show that the federated query planning, evaluation, and caching techniques are able to produce query results quickly and efficiently. The effects of several optimizations on the execution of federated queries is discussed, and their impact on performance is evaluated.

CHAPTER 1
INTRODUCTION

Evaluating queries over structured data on the Web has attracted a great amount of attention in recent years as the amount of such data has increased dramatically. Structured data encoded using the Resource Description Framework[1] (RDF) format is primarily made accessible by two means on the web: served as static content and made accessible via the SPARQL Protocol[2]. Static RDF files made available via the Hypertext Transfer Protocol (HTTP) allow simple access and easy discovery by dereferencing the Uniform Resource Locator (URL) of a resource of interest. Data made available using the SPARQL query language[3] and protocol (which itself uses HTTP) allows for expressive queries, but requires *a priori* knowledge of the location of SPARQL database "endpoints".

Each of these access mechanisms has benefits and limitations, and the choice of which is used to make data available on the web may be based on several factors including the ease of publishing or consuming the data and resource requirements for hosting the data. As a result, the access mechanisms are often used in a complementary fashion, providing access to overlapping subsets of data. The data of interest to a user might be distributed between a collection of SPARQL endpoints and static RDF files. To answer a specific query, the relevant data from these sources must be retrieved and joined in such a way as to form a new dataset containing enough information to compute the answer.

However, little work has been done to address the ability to efficiently evaluate queries over data that are divided across the Web combining both of these static and dynamic forms. If a query system can provide efficient evaluation of queries over data widely dispersed on the Web, latent value of the data can be exposed that otherwise would require overly-burdensome centralization and/or high-latency crawling of the data. Moreover, if the query system can utilize the linked nature of the data, both static and dynamic data sources unknown at the time of query construction can be used during query evaluation to increase the quality and completeness of query

results.

In this thesis, we develop a query evaluation framework capable of federating existing sources of RDF data on the Web and permitting access to that data using the standard SPARQL query language. This framework makes use of existing SPARQL databases as well as static, linked RDF data available on the Web, and it also augments known data sources with those discovered during query evaluation. A primary advantage of this framework is that it is based on traditional cost-based query optimization, allowing integration with existing database optimization techniques.

The focus of this federation framework is on providing *early results* to basic graph pattern queries. That is, the query execution plan generated by the framework attempts to optimize the time to produce initial results, even at the cost of overall execution time. This focus is based on the fundamental assumption that in the context of query execution on the Web, very often the cost of generating *complete results* to a query may be prohibitive. This is especially true for this system in which new data sources may be continuously discovered during query execution, yielding an unpredictable and potentially unbound execution time. Instead, by prioritizing early results that can be delivered to the user quickly and concurrently with continued query evaluation, this framework is able to produce useful results to the user in a fraction of the time that would be required if trying to minimize overall execution time.

As a simple example of this federation approach, consider the following SPARQL query:

This query retrieves authors—and their email addresses—who published papers on the topic of SPARQL in the proceedings of the ESWC 2009 conference. The query has five triple patterns, which we will refer to by their predicates: `hasPart`, `author`, `hasTopic`, `label`, and `mbox`. Assume we would like to execute that query against the following two data sources:

- Source 1 is a SPARQL endpoint containing data matching all five predicates.
- Source 2 is a SPARQL endpoint containing data matching the first four predicates (all but `mbox`).

```
SELECT ?author ?mbox WHERE {
  <http://data.semanticweb.org/conference/eswc/2009/proceedings>
    swc:hasPart ?pub .
  ?pub swrc:author ?author ;
    swc:hasTopic ?topic .
  ?topic rdfs:label "SPARQL" .
  ?author foaf:mbox ?mbox
}
```

Query 1: Authors—and their email addresses—of research papers about SPARQL.

In constructing a query execution plan for this query and the available data sources, several things become apparent:

1. Source 1 may be able to immediately provide results to the query, as it contains data matching all five triple patterns;
2. Source 2 may be able to provide intermediate results matching four of the five triple patterns, which would need to be joined with data from Source 1;
3. Some results may require more granular intermediate results from each source (e.g. data for `author` and `mbox` from Source 1, and `hasPart`, `hasTopic` and `label` from Source 2.

Given this, intuitively a query plan seeking early results ought to first send the entire query to Source 1, allowing any results from this single remote call to be immediately returned; second, attempt to retrieve intermediate results matching `hasPart`, `author`, `hasTopic`, and `label` from Source 2 and join them with `mbox` triples from Source 1; third, attempt other possible permutations of the five triple patterns and two sources (preferring to send partial queries with more than one joining triple pattern at a time to a source).

We show results demonstrating that this framework is able to produce results faster than existing solutions to querying linked data and is competitive with existing SPARQL federation systems. However, without further augmentation there are query results this approach is unable to produce, despite potentially having access

to enough information to produce more results. For example, consider a situation in which during query evaluation, the response to a subquery from Source 1 includes the following HTTP header:

```
Link: <http://example.org/sparql>; rel="sparql"
```

If the `sparql` link relation is understood by the query system to indicate the location of a SPARQL endpoint containing more information about the resources in the query results, this header would provide enough information to augment the set of data sources with the SPARQL endpoint at `http://example.org/sparql`. Then, results could be produced from, e.g., sources 1 and 2, and this newly discovered endpoint. Therefore, we extend the query planning algorithms with data source augmentation techniques to expand the class of answerable queries.

One major challenge in evaluating federated queries is in limiting the amount of *unnecessary* work that is performed. Data relevant to a query is distributed across data sources in non-uniform fashion, and knowing specifics about the distribution can reduce the number of operations necessary to evaluate a query. The use of existing metadata standards enable federation systems to gain knowledge of data sources including their capabilities and rudimentary statistics about their data. However, these standards often stop short of providing metadata tailored specifically for query federation.

Another challenge to efficient federated query evaluation is the cost of repeatedly evaluating frequently used sub-queries. In real-world workloads, a few popular queries represent a disproportionately large proportion of all queries. This skewed distribution is even more apparent when dealing with individual query patterns. A small number of query patterns appear in many of the overall queries handled by the system. During query evaluation, repeatedly generating results for these sub-queries is costly and redundant.

To reduce the cost of these redundant operations, we have developed data structures and algorithms that allow SPARQL endpoints to support fine-grained caching of (sub-)query results. This improves performance by allowing SPARQL clients to use the features of HTTP to cache results for frequently used sub-queries.

By utilizing cached query results, latency and overall performance can be improved by reducing workload on the SPARQL endpoints.

In this dissertation, we formalize this federated query planner and the data source augmentation mechanism by which SPARQL endpoints and linked data sources may aid in improving query results. We present results of an evaluation of the federated query framework over a dataset based on real-world bibliographic data. This evaluation shows the framework's ability to effectively answer federated queries with varying initial knowledge of data sources while improving result set completeness. Moreover, we demonstrate that simple metadata beyond those currently in use cannot only improve query results but also query performance.

1.1 Contributions

The specific contributions of this dissertation are as follows:

1. A framework for federated query evaluation over SPARQL and linked data sources including a query planning algorithm designed for early results;
2. A method for augmenting knowledge of data source topology during query evaluation, resulting in the ability to produce more results with better overall performance;
3. A method for caching SPARQL (sub-)query results;
4. Proof of concept for metadata beyond current standards to improve federated query performance and result set completeness.

1.2 Organization

This dissertation is organized as follows. Chapter 2 reviews related work in the areas of query federation, query planning and optimization, methods of adaptive query plan rewriting, and query result caching. Chapter 3 introduces the federated query planning framework, providing specific detail for both the initial query planning and query re-planning phases. Chapter 4 describes the augmentation process which enables query re-planning through the discovery of potentially relevant sources

of new data. Chapter 5 defines the method for enabling SPARQL endpoints to participate in HTTP-level result caching. Chapter 6 presents results of an evaluation performed on the components discussed in chapters 3–5. Finally, chapter 7 presents concluding remarks and discusses potential areas of future work.

CHAPTER 2
RELATED WORK

2.1 Query Federation

2.1.1 Federation in Relational and Information Integration Systems

Query federation over distributed databases has been a topic of research for several decades. Kossmann[4] provides a good survey of the history of this research including work on traditional federation architectures, cost estimation, and query optimization. Sheth and Larson[5] describe a specific architecture for relational federated database systems.

Beyond relational databases, *information integration systems* design and use of wrapper functions to translate requests between a common query language used by the federation system and data-source specific operations allow heterogeneous data sources to operate in a federated system. Roth, Özcan and Haas[6] propose a cost model in the Garlic system that allows wrappers to provide source-specific function implementations for describing data statistics and operator costs. Allowing wrappers to provide customized cost model information allows the traditional cost-based query optimizer to be implemented without specific knowledge of any particular source, while enabling efficient query plan selection.

While the ability to provide source-specific statistics is crucial in a distributed environment, the shared data model of RDF and interface of SPARQL obviate the need for custom implemented wrappers. In chapter 4 we discuss how SPARQL endpoints and linked data sources may provide similar information for enabling efficient query evaluation without requiring any changes to the query planner or evaluation framework.

2.1.2 Semantic Web Federation

The first major work on federated SPARQL, Quilitz and Leser's DARQ[7] project relies on pre-defined service descriptions for participating SPARQL endpoints which declared endpoint capabilities in terms of supported predicate values.

During query evaluation each triple pattern of a query is sent to every endpoint which can provide answers for the pattern based on the pattern's predicate. This approach is simple and effective, but cannot support queries with unbound predicates and (except in trivial cases) requires all join operations to be evaluated at the client.

Langegger et al. [8] present SemWIQ, a mediator-based system for evaluating queries across heterogeneous data sources that are wrapped by SPARQL endpoints. While it uses a similar approach to DARQ, SemWIQ is better able to deal with changing datasets by maintaining a dynamic catalog of statistics about participating data sources.

Berners-Lee et al.[9] developed the Tabulator system as an in-browser tool to browse, view, and interact with RDF data. The Tabulator aggregates semantic data during user-driven navigation, resultint in a local RDF store containing data on the resources navigated to, as well as the closure of ontology terms used. This data may then be subjected to structured query and viewing in a variety of interfaces.

Hartig, Bizer, and Freytag's SQUIN[10] is a system for executing SPARQL queries directly over the web of linked data by iteratively retrieving RDF data identified by intermediate query results. This approach has the advantage that it can discover previously unknown sources of data during query evaluation and immediately incorporate data from those sources into query results. However, this approach has several limitations. The data over which queries are executed must be published according to Linked Data principles[11] (primarily the use of dereferenceable HTTP URLs) and a query must contain at least one such URL from which to start the traversal of RDF data. Moreover, despite aggressive pre-fetching and the use of non-blocking iterators, the system is still limited by the latency and throughput of the queried web servers.

Görlitz and Staab's SPLENDID system [12] leverages descriptions of remote SPARQL endpoints using the Vocabulary of Interlinked Datasets[13] (VoID). Statistical data in the form of VoID descriptions describe sources and data available for query evaluation, and is used for source selection and join ordering during query planning. This use of authoritative metadata in query planning is shown to be very

effective at generating efficient query plans. Our approach is designed to be able to benefit from similar statistical information but differs from SPLENDID in several ways. We neither require that all data sources be known to the federating system, nor that they provide accurate (or even *any*) statistical information about their data. More importantly, we utilize a much more relaxed system of data sources, allowing sources to be discovered mid-execution, and allowing static linked data to be used as sources.

The FedX system developed by Schwarte et al.[14] takes a proactive approach to query plan pruning by probing each SPARQL data source to determine if it will produce results to any of the query's triple patterns. This filtering process, combined with join order optimizations and the use of "bound joins" allow FedX to efficiently execute federated queries across many SPARQL sources.

2.2 Query Optimization

The issue of query optimization has been studied in databases extensively. Most relevant to this work are information integration systems which consider source capabilities [6], rewriting of queries to find all possible ways to answer a query [15, 16] and estimating the cost of remote queries [17] and distributed databases [4] that considers distribution of data among the servers. Most of this work does not explicitly deal with the problem of too many sources that can answer any part of the query, which is unique to Semantic Web[18] due to its emphasis on common identifiers and vocabularies. Very often sources have limited capabilities, which must be captured and used in query optimization. In our cases, all sources are assumed to have full query capability. Mocha [19] considers the shipping of operations that reduce the volume of shipped data closer to sources together with the data processing code. This is similar to the approach we take here, but we extend it to a federated scheme. In database literature, it is common to use a dynamic programming approach that uses rewrite rules that describe how the query tree can be modified. The optimization is then cast as a search and prune algorithm based on cost estimation. However, as the search space is very large, often heuristics are used to guide the search and reduce the complexity [20] which is what we propose here for our domain.

The issue of early returns has been studied in databases and information integration as well. However, these methods are mostly based on the availability of the query engines to send partial and/or ranked information [21, 22]. In contrast, we consider which queries to send to endpoints as a way of implementing early returns. The topic of personalization of optimization and query execution has received quite a bit of attention lately [23, 24, 25], especially due to the many different devices and contexts that might be used to execute a user query. While we do not explicitly study this problem, we show how our methods facilitate personalization. We leave the details of personalization in our framework to future work.

Magliacane et al.[26] discuss an extension to the SPARQL algebra to optimize the execution of top-k SPARQL queries where only the top k results are desired after ordering of results is performed based on a user-defined scoring function. While we do not address user-defined preferences, this work would be a natural fit in extending our approach to efficiently producing early results based on a scoring function.

2.3 Adaptive Query Planning

There has been much work in the use of adaptive planning to cope with unexpected or changing query execution environments. Much of this work has focused on changing query plans in order to compensate for unexpected latency and "bursty" throughput of data sources.

Amsaleg et al.[27] present "query scrambling", a mechanism to re-order the execution of query plan operators, and to rewrite the query plan mid-execution in response to the latency of remote data sources. With a more radical departure from traditional query planning, Avnur and Hellerstein developed a processing mechanism called "eddies" which "[merges] the optimization and execution phases of query processing"[28]. As source data and intermediate results become available during query processing, the eddy mechanism continuously applies query operators to applicable data. Once an intermediate result has had all query operators applied to it, the eddy emits it for further processing (or to the client if no further processing is required).

The SPARQL federation system ANAPSID by Acosta et al. focuses on adap-

tive query execution by developing query operators and a scheduler that is able to adapt to "data availability and run-time conditions"[29].

While these and similar systems demonstrate the potential effectiveness of making query planning decisions mid-execution, they do not address the challenges of changing a query plan to incorporate new data sources. The SQUIN[10] system does attempt to address this problem. However, SQUIN does not do query planning *per se*; it is designed to iteratively crawl linked data and continuously match that data with parts of the query pattern. This iterative crawling approach is inherently able to utilize newly discovered linked data sources and is shown to be effective, but it is not able to leverage data in existing SPARQL endpoints, and is unable to take advantage of the large, existing literature on query plan-based optimization.

2.4 Query Caching

Using caching to increase scalability touches upon several areas of research, including statistical distributions of queries affecting their cacheability, indexing structures used in Semantic Web query answering systems, and caching as it relates to both the Semantic Web and to databases. In this section we discuss work related to these areas.

Regarding the repetition of queries, work on analyzing web access logs by Breslau et al.[30] found that the statistical distribution of requests followed a "Zipf-like distribution" with the distribution exponent varying between different user communities. This finding suggests that caching can have a significant impact on real-world access patterns because a large portion of requests are made for a small set of resources. More recently, and related specifically to SPARQL requests, Gallego et al.[31] analyzed a set of SPARQL endpoint query logs, and found a high degree of queries duplicated from the same hosts. However, they only mention the repeated queries in passing, without specific details on the distribution of repeated queries.

There has been a trend in SPARQL systems to use search trees to efficiently index RDF data (following similar use in relational databases) and to use many indexes to support a range of access patterns. The YARS system[32] (and subsequently Hexastore[33] and RDF-3X[34]) demonstrated the effectiveness of maintaining many

search tree indexes to provide direct access to RDF data matching a certain triple- or quad-pattern. YARS made use of six B+ tree indexes ($\langle SPOG\rangle$, $\langle POG\rangle$, $\langle OGS\rangle$, $\langle GSP\rangle$, $\langle GP\rangle$, $\langle OS\rangle$) to cover all sixteen possible quad-access patterns. Hexastore and RDF-3X, while only considering triples, both made use of six indexes ($\langle SPO\rangle$, $\langle SOP\rangle$, $\langle PSO\rangle$, $\langle POS\rangle$, $\langle OSP\rangle$, $\langle OPS\rangle$) to provide complete indexing of triples, covering all eight triple access patterns and all possible orderings. The design of 4store[35] makes use of only three indexes ($\langle PS\rangle$, $\langle PO\rangle$, $\langle G\rangle$), using RADIX tries for the $\langle PS\rangle$ and $\langle PO\rangle$ indexes[1]. While all of these systems utilize search trees for performance, their use is restricted only to indexing the RDF data. In our work, we make use of the search trees not only to maintain many indexes over the RDF data but also to store additional metadata about when that data was modified. As described in the following sections, keeping such metadata allows a query processor to validate existing cached query results.

Caching database query results has been studied widely. Goldstein and Larson[36] show the potential of materializing views within a relational system to dramatically improve performance of expensive queries. Both Amiri et al.[37] and Larson, Goldstein, and Zhou[38] address caching relational query results using materialized views. Amiri et al. perform caching in edge caches separate from the origin database which reduces load on the origin server, but maintaining consistency of cached results requires that the origin database propagate every update, delete, and insert operation to all caches, making it unsuitable for environments with high write throughput. Larson, Goldstein, and Zhou improve upon the approach by Amiri et al. by supporting parameterized queries and allowing more flexible materialization of views, allowing the query optimizer to choose whether to evaluate the query on the origin server even in the presence of cached data. In contrast to these approaches, the fixed structure of RDF data makes supporting caching much easier. No complex logic or knowledge of database schemas is needed to determine which tables or columns might benefit from caching as all data is structured as triples.

Caching as it relates to the Semantic Web is a much more recent field of study. The work on caching SPARQL query results by Martin, Unbehauen, and Auer[39]

[1] 4store actually maintains two RADIX tries *for each predicate*, but for our purposes these may be understood as being equivalent to tries with P prepended to the actual keys.

shares the same goal and many details with our work (we frequently cite this work in chapter 5 as a source of common groundwork and greater detail). However, they perform caching by coupling a caching layer with an existing SPARQL processor. This has the benefit of portability across SPARQL implementations, but incurs high cache maintenance costs and is suitable only for caches which are tightly integrated with the underlying system and can intercept all write operations (e.g. ISP caches or caches built into a user agent cannot be used). Finally, the work in [39] deals with only a subset of SPARQL; in this work we make specific note of several features of SPARQL that deserve special attention in the context of caching.

Work by Hartig[40] shows performance improvements of using a local cache in evaluating queries by linked data link traversal. This work is complementary to ours in that both use caching to improve performance of query answering, one over static linked data, the other over potentially dynamic data available via structured queries.

CHAPTER 3
FEDERATED QUERY PLANNING

We consider a query federation planning framework where users write queries using a common vocabulary shared amongst different data sources. The query processor has access to an initial set of data sources (i.e. SPARQL endpoints and linked data resources). The initial set of sources does not necessarily contain all relevant sources. Other sources relevant to query answering may be discovered during the execution of the query, and should equally participate in query result generation.

We do not necessarily know in advance which graph patterns will match data available at any source. The availability of this information may alter the planning phase by, e.g., reducing the number and size of possible query plans, but we support the use of sources with no such metadata available.

The query planning problem we address can be stated as: given a SPARQL query and a set of initial data sources how do we construct a query plan that provides answers to the query using all the available sources while being able to take advantage of newly discovered sources mid-execution. In particular, we focus on the problems of *mid-execution augmentation* and *early returns*. We seek to leverage newly discovered information about data sources and their topology *during* query execution, allowing for more efficient and complete query answering. Also, in the web context, it is often the case that the set of all answers to a query is potentially too large for practical purposes and the user is not interested in all of them. Therefore, it is important to return *some* answers to a query as quickly as possible.

Given these goals, there are a number of issues to consider:

- All sources may potentially provide answers to all triple patterns in the query. This leads to an increase in the optimization search space as we illustrate in the next section.

- Data sources may be unknown to the planning system. That is, the system may not have any relevant information about a specific source such as database size, cardinality of frequent patterns, or join selectivities.

- It is not known how many results should be returned to the user quickly. The user may explicitly request a specific number of results, but without such information a good balance must be found between partial result set size and resource consumption.

- It is expected that query plans will frequently need to be expanded to incorporate new data sources. Therefore, query plans should be designed such that the overhead of rewriting the query plan is kept small.

We present a query planning framework that is a first step towards answering these issues. This framework is based on the idea that distributed RDF query answering represents an environment in which any server can potentially provide answers to any triple pattern. This represents the worst-case scenario for traditional distributed relational database systems (akin to every server having a random partition of every relation). For this reason our framework makes use of a logical query rewriting rule to constrain the otherwise exponential query plan search space. Specifically, the rewriting rule greedily groups triple patterns together into subqueries for execution at each remote endpoint.

The problem of database federation and information integration is well studied. However, RDF presents some new challenges. First of all, the open ended nature of RDF makes it hard to describe source capabilities. All sources may answer queries about all graph patterns. The availability and use of service descriptions and dataset statistics can alleviate but not solve this problem, as these metadata can be unavailable, incomplete, or inaccurate. Another challenge is that although there is a shared data model and API in RDF and SPARQL, the decentralized nature of these systems means that in general no system can guarantee complete knowledge of relevant data sources.

We adopt the traditional dynamic programming approach for enumerating query plans and computing cost of these plans. However, we introduce a new heuristic for "pushing joins down", and a novel query re-planning phase to expand in-progress plans. Our heuristic introduces redundant subqueries, increasing the overall query cost while decreasing the time to retrieve early results. This saves time for early results in two ways: first, costly join operations are handled by more

efficient structures; and second, the amount of data sent to produce a specific tuple is reduced, resulting in early returns. This is a departure from most query optimization heuristics that attempt to reduce the total query time or resource consumption. We attempt to address the issue of redundant subqueries by leveraging query result caching as described in chapter 5.

3.1 Architecture Overview

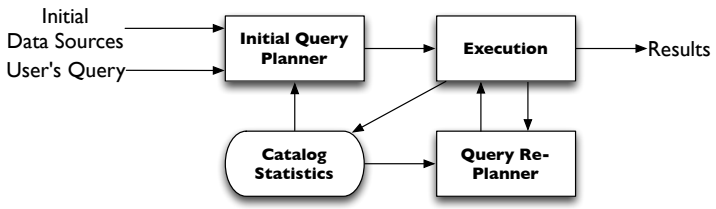


Figure 3.1: Federation Architecture

To address the challenges discussed above, the federated query planner is comprised of several components as shown in figure 3.1. An *initial query planner* is used to construct an initial query execution plan that is able to retrieve data from the initial set of data sources and produce initial query results. The initial planner constructs the query plan in such a way that the plan may be rewritten mid-execution by a *query re-planner*. During query execution, new data sources may be discovered by inspecting intermediate results and data accessed in constructing those intermediate results. Newly discovered data sources are periodically made available to the query re-planner, allowing the query plan to be expanded before resuming query execution. We refer to this process as *data source augmentation*, and provide details of how new sources are discovered in chapter 4.

3.2 SPARQL Query Language

The SPARQL Query Language[3] defines a syntax and semantics for queries over RDF data. Among its supported features are basic graph pattern (BGP) matching, pattern conjunctions and disjunctions, aggregation, negation, selection, and projection. BGPs are the primary mechanism of matching query patterns to

RDF data and mapping those values to variables (creating query solution mappings); most of the other SPARQL operators build upon the BGP matching mechanism, operating on a multiset of solution mappings produced by either BGP matching or by other SPARQL operators. Therefore, in this work we restrict ourselves to the problem of evaluating BGPs in a federated environment.

3.2.1 SPARQL Definitions

Below we provide the parts of the definition of the SPARQL Query Language[3] relevant to our discussion of federated query planning.

Definition The set of *RDF Terms*, T, is the union of all IRIs, I, the set of all RDF Literals, and the set of all blank nodes in RDF graphs.

Definition The set of *query variables*, V, is infinite and disjoint from T.

Definition A *triple pattern* is any member of the set: $(T \cup V) \times (I \cup V) \times (T \cup V)$

Definition A *basic graph pattern* is a set of *triple patterns*.

Definition A *solution mapping*, μ, is a partial function $\mu : V \mapsto T$. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined.

Basic graph patterns can be instantiated by replacing both variables and blank nodes by terms. Blank nodes are replaced using an RDF instance mapping, σ, from blank nodes to RDF terms; variables are replaced by a solution mapping from query variables to RDF terms.

Definition A *Pattern Instance Mapping*, P, is the combination of an RDF instance mapping, σ, and solution mapping, μ, such that $P(x) = \mu(\sigma(x))$. Let BGP be a basic graph pattern and let G be an RDF graph. μ is a *solution* for BGP from G when there is a mapping P such that $P(BGP)$ is a subgraph of G and μ is the restriction of P to the query variables in BGP.

3.2.2 Accessing RDF Data with SPARQL

We restrict our system to using only data sources that may be accessed and queried with SPARQL. This includes both native SPARQL endpoints accessible directly via the SPARQL Protocol[2] (an HTTP-based protocol for submitting queries to a remote database and a syntax for encoding query results) and "linked data" encoded natively as RDF and available via HTTP (with the use of a locally constructed wrapper that provides a SPARQL interface to the underlying RDF data). The use of technologies such as the RDB to RDF Mapping Language[41] also allows systems with data not natively stored as RDF to be served via SPARQL and so made accessible for query federation.

3.3 Initial Query Planning

The initial query planner is responsible for constructing a query execution plan over the initial set of data sources (either from a source list provided by the user, or from a catalog of sources known to the system). Given a query, the initial planner first determines all possible endpoints that can answer each triple pattern. Where available, service descriptions are used to filter any endpoints which are known not to be relevant to the query. In cases where such information is unavailable, an endpoint is conservatively assumed to be able to answer any triple pattern. Using the filtered endpoint list, each triple pattern in the query is replaced with an expression that sends each pattern to each relevant endpoint and then takes their union at the federation site. We will consider this the naïve query plan. As an example, consider Query 2 (we have given names to each graph pattern for simplicity).

Suppose we have two SPARQL endpoints available as data sources, labeled "1" and "2". The naïve query plan that accesses each source for each graph pattern is shown in Figure 3.2. In this query plan, $\texttt{topic}^1$ ($\texttt{topic}^2$, respectively) refers to the `topic` graph pattern answered by source "1" ("2"). In this query plan, all individual graph patterns are computed at the endpoints and are sent to the federation site and combined there. The naïve query plan can be optimized to determine the order of the joins using a cost-based dynamic programming approach common in query planning. Algorithm 3 in Appendix A shows the usual dynamic optimization

algorithm for finding the best execution plan for a query.

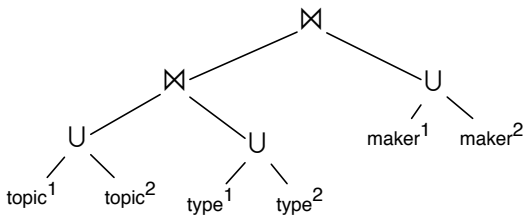


Figure 3.2: A logical query plan for three triple patterns at two sources.

However, before the best physical query plan is considered, we need to find the best logical query plan using a set of well-known algebraic equivalences (rewrite rule) and then for each plan find the best possible physical plan. As the search space of such an optimization algorithm is too large for practical purposes, the common method is to use a heuristic to give precedence to certain rewrite rules that are known to be beneficial and limit search space by eliminating query plans that cannot produce better cost estimates. Following existing work such as Stocker et al.[42], we disallow consideration of query plan join orders that would result in cartesian joins.

In our scenario, we focus primarily on a planning heuristic to group join operations that may be executed together at a remote endpoint. To accomplish this, we need to rewrite the query tree in Figure 3.2 so that unions are "pulled" toward the root and joins are pushed toward the leaves of the tree. Using this heuristic, the rewritten version of the same query can be seen in Figure 3.3. Here $\bowtie^1$ refers to a

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
SELECT * WHERE {
  ?paper rdf:type swrc:InProceedings .     # type
  ?paper foaf:topic dbpedia:Semantic_Web . # topic
  ?paper foaf:maker ?person .              # maker
}
```

Query 2: Authors of research papers about the Semantic Web.

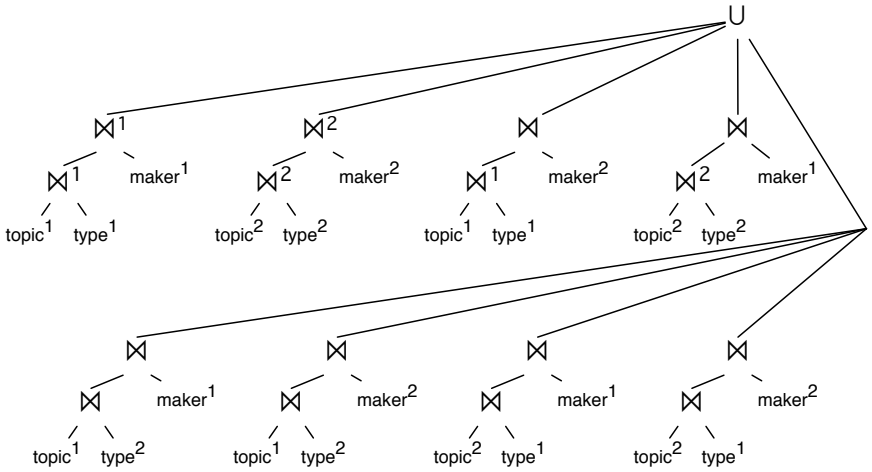


Figure 3.3: A rewritten logical query plan for three triple patterns at two sources.

join operation performed at source "1" (as opposed to $\bowtie$ which is a join performed by the federation client). Note that while source "1" may provide some answers to the entire query, to find all possible answers to this query we still need to retrieve each individual query pattern from source "1", and join with corresponding expressions from source "2". This is illustrated with the remaining six join subqueries in the query plan performed at the federation site.

This query plan provides some benefits over the previous query plan: it is possible to get quick answers to the query by using subqueries that are faster to compute at the endpoints and return fewer results hence reducing the latency. These results can be quickly shown to the user while other subqueries are processing. However, one does not want to send all the subqueries in this rewritten plan to the endpoints concurrently, as this would cause unnecessary load on the servers and might consume an unnecessary amount of resources locally at the federation site. A discussion of this tradeoff and potential approaches to addressing it are given in section 7.2.

Compared to a traditional query optimizer, this heuristic of grouping same-source joins reduces the search space by constraining the number of possible query rewritings to consider. Consider a query $Q = P_A \bowtie P_B$ and data sources 1 and 2, with the initial logical query plan $Q = (P_A^1 \cup P_A^2) \bowtie (P_B^1 \cup P_B^2)$. This heuristic

yields:

$$Q = (P_A \bowtie^1 P_B) \cup (P_A \bowtie^2 P_B) \cup (P_A^1 \bowtie P_B^2) \cup (P_A^2 \bowtie P_B^1)$$

with the join subquery $(P_A \bowtie P_B)$ being executed remotely twice (once at each of the two endpoints) and locally twice (once for each possible cross-source join). In comparison, a traditional query optimizer would also consider the variants:

$$Q = ((P_A^1 \cup P_A^2) \bowtie P_B^1) \cup ((P_A^1 \cup P_A^2) \bowtie P_B^2)$$

and

$$Q = (P_A^1 \bowtie (P_B^1 \cup P_B^2)) \cup (P_A^2 \bowtie (P_B^1 \cup P_B^2))$$

With more than two patterns or more than two endpoints, the number of possible variants increases exponentially.

The exponential size of the plan rewriting space makes traditional query optimization very expensive. This heuristic decreases the cost of query optimization by constraining the rewriting space and provides some of the benefits of query shipping[43]. Intuitively, this heuristic is appealing for "web scale" queries because the size of data and the number of potential sources involved makes it undesirable to transmit all intermediate results over the network and perform join operations locally at the federation client. Moreover, at "web scale" it is possible (even likely) that the client is only interested in some fraction of the total results. If the client is interested in only 100 results, executing the efficient subqueries of Q first (such as $P_A \bowtie^1 P_B$) may avoid the need to execute less efficient subqueries (e.g. $P_A^2 \bowtie P_B^1$) if the efficient subqueries can provide at least 100 results.

3.3.1 Cost Model

In evaluating potential query plans, we consider the cost of local operations and remote operations. The availability of service descriptions and dataset statistics generated by the data sources themselves can greatly improve the cost estimate for remote operations. Where available in service descriptions, we make use of predicate partition counts using the VoID[13] vocabulary. This data is used to both test for the potential presence or absence of a triple pattern in an endpoint (used in Algorithm

4, *accessPlans*) and to support in estimating the cardinality of triple patterns.

Other work, such as the FedX[14] system, have shown promising results in selecting relevant sources for each triple pattern through the use of locally cached data and SPARQL `ASK` queries performed before query federation begins. We do not evaluate such techniques, though note that they are orthogonal to our work and may prove themselves effective in combination with more complex cost models and the use of service description data.

3.3.1.1 Local Operations

The operations performed locally in our system are hash join and union. We implement a hash join algorithm, relying on pipelining of the union operator to enable early results. Due to the planning heuristic of pushing joins down (and thus unions up), results from subplans that appear as children of union operators may be returned immediately. The cost of join and union operations is computed using the expected execution time of the left- and right-hand side operands.

$$cost(p \bowtie q) = cost(p) + cost(q) + HashJoinCost(p, q) \tag{3.1}$$

To simplify cost estimation, and based on experimental evaluation, we assume that $HashJoinCost(p, q)$ is dominated by the IO cost incurred by evaluating p and q. We therefore simplify the cost of local joins to be:

$$cost(p \bowtie q) = cost(p) + cost(q) \tag{3.2}$$

A potentially more accurate (and expensive) cost estimate could be performed in situations where the local join cost is not dominated by IO and where accurate join selectivity data is available (e.g. obtained via more expressive service descriptions).

We do not use double-pipelined hash joins even though they would promote early returns because they require all intermediate results to be kept materialized. This would represent a burdensome memory requirement for the specific query plans we generate. Görlitz[12] shows promising federated query evaluation results with the use of both hash joins and *bind joins* which suggests that there is potential for improved performance through the use of more complex operators in future work.

The union operator is designed to execute its operands sequentially. When visualized, as in Figure 3.3, we understand the sequence of union operand execution to be in "left-to-right" order. To leverage this design and the heuristic of grouping plan operators so that entire union subplans may directly produce whole-query results, union operands are ordered with respect to the number of remote operations they invoke (e.g. the number of source-label superscripts in the syntactic form of the subplan). This ordering takes the form of a bucket sort of subplans based on the number of remote operations, and with subplans which invoke the same number of remote operations being sorted according to their estimated cost as usual.

The cost of a union operator is computed as the sum of the cost of its operands. This assumption represents the worst-case scenario for a union operator; subject to available resources, parallelization of the operands is possible and would in the best case allow the cost of a union operator to be reduced to the maximum cost of its operands. We propose further investigation into such parallelization in Chapter 7.

$$cost(\bigcup_{i=1..n} p_i) = \sum_{i=1..n} cost(p_i) \tag{3.3}$$

3.3.1.2 Cost of Remote Operations

The cost of making a call to a remote endpoint is estimated based on statistics either provided by the endpoint or computed from previous calls to these endpoints. Data is kept in a catalog for latency, the number of results returned per second (throughput), and the total number of results previously received for similar patterns.

We define $bf(P)$ to be the "bound/free" characterization string of a pattern, P (e.g. "fbb" for the triple pattern `?paper foaf:topic dbpedia:Semantic_Web`). Latency for an endpoint E, $Latency_E$, is defined as a set of mappings from a bound-/free string, b, to a multiset of observed latency values, L (expressed in seconds). Similarly, throughput for an endpoint E, $Throughput_E$, is defined as a set of mappings from a bound/free string, b, to a multiset of observed throughput values, T (expressed in results per second). Cardinality data is defined in two ways. $Cardinality_E$ is defined as a set of mappings from a bound/free string, b, to a multiset of observed

cardinality values, C. $SDCardinality_E$ is based on the (complete and accurate) cardinality data optionally provided by an endpoint via a service description, and is defined as a set of mappings from a predicate IRI, p, to a cardinality value, c. These statistics are continuously updated as new data and endpoints are observed, in order to improve estimates.

In case of an unknown source, an average situation is assumed (over all the known sources). It may be possible to obtain more accurate estimates based on more complex statistics (such as data broken down by time of the day, as it is typical for systems to exhibit cyclic behavior).

$$Throughput(E,b) = \begin{cases} \overline{T} & : (b \mapsto T) \in Throughput_E \\ \frac{\sum_{(b' \mapsto T) \in Throughput_E} \sum_{t \in T} t}{\sum_{(b' \mapsto T) \in Throughput_E} |T|} & : Throughput_E \; exists \\ \frac{\sum_{e \in knownE} \sum_{(b' \mapsto T) \in Throughput_e} \sum_{t \in T} t}{\sum_{e \in knownE} \sum_{(b' \mapsto T) \in Throughput_e} |T|} & : otherwise \end{cases} \tag{3.4}$$

$$Latency(E,b) = \begin{cases} \overline{L} & : (b \mapsto L) \in Latency_E \\ \frac{\sum_{(b' \mapsto L) \in Latency_E} \sum_{l \in L} l}{\sum_{(b' \mapsto L) \in Latency_E} |L|} & : Latency_E \; exists \\ \frac{\sum_{e \in knownE} \sum_{(b' \mapsto L) \in Latency_e} \sum_{l \in L} l}{\sum_{e \in knownE} \sum_{(b' \mapsto L) \in Latency_e} |L|} & : otherwise \end{cases} \tag{3.5}$$

With the goal of promoting early results, and relying on union plan operators to be pipelined and operate on sub-plans in ascending cost order (as described in the previous section), the cost of a triple pattern, t, is calculated as the estimated time of its execution. However, to avoid early execution of sub-plans which will result in no results, triple patterns with an estimated cardinality of zero are assigned an infinite cost.

$$cost(t^E) = \begin{cases} \infty & : Card(t^E) = 0 \\ \frac{Card(t^E)}{Throughput(E, bf(t))} + Latency(E, bf(t)) & : otherwise \end{cases} \tag{3.6}$$

The cost of joining two patterns is defined for remote execution:

$$cost(p \bowtie^E q) = \begin{cases} \infty & : Card(p \bowtie^E q) = 0 \\ \frac{Card(p \bowtie^E q)}{Throughput(E, bf(p \bowtie q))} + Latency(E, bf(p \bowtie q)) & : otherwise \end{cases} \tag{3.7}$$

3.3.1.3 Cardinality Estimation

Estimating cardinality accurately is a hard problem even for the more constrained situation involving a centralized database. When authoritative metadata is not provided by an endpoint, we choose to keep simple statistics that are aimed at providing approximate size estimates.

Our system keeps statistics about the number of results returned from an endpoint and uses this information for subsequent cardinality and cost estimation. If no statistical information about a pattern is known for the endpoint in question, the average of expected result cardinality over all known endpoints is used. If no known endpoint can supply an estimate, a simple heuristic is used in which the cardinality is estimated based on the number and position of variable terms in a pattern (e.g. `?paper foaf:topic dbpedia:Semantic_Web` is estimated to return fewer results than `?paper foaf:maker ?person`). For estimating join selectivity where no known data is available, we make the simplifying assumption that the size of the join result is bounded by the larger of the joined solution sets.

$$Card(t^E = \langle s, p, o \rangle) = \begin{cases} c & : bound(p) \wedge (p \mapsto c) \in SDCardinality_E \\ \overline{C} & : (bf(t) \mapsto C) \in Cardinality_E \\ \frac{\sum_{e \in knownE} Card(t^e)}{|knownE|} & : otherwise \end{cases} \tag{3.8}$$

$$Card(p \bowtie q) = \begin{cases} \overline{C} & : (bf(p \bowtie q) \mapsto C) \in Cardinality_E \\ \max_{r \in \{p,q\}} Card(r) & : otherwise \end{cases} \tag{3.9}$$

3.3.2 Issues

Two issues related to query execution are worth mentioning here. Even though evaluation of basic graph patterns on a set of triples should not produce duplicate query results, it is possible to receive duplicate results from different sources. The same data may be located in multiple locations. As a result, the query execution engine may need to remove duplicates at query time if the user wishes the results to appear as if matched against a single dataset.

Furthermore, the plan generator given in Appendix A may end up producing join orderings that are suboptimal as the join ordering is considered before the logical rewriting of the query to reduce the search space. To see this, consider the last 4 subqueries in Figure 3.3. It is possible to rewrite these subqueries so that the join ordering allows grouping triple patterns from the same source together so that they may be joined at that source. For example, provided there are appropriate join variables, the subplan $((topic^1 \bowtie type^2) \bowtie maker^1)$ in figure 3.3 can be reordered to allow a join at source 1: $((topic \bowtie^1 maker) \bowtie type^2)$. Note that in some cases this rewriting is fairly easy to perform. Simple rule-based rewriting might achieve this after the logical rewriting, or an approach to dynamically fix these suboptimal subplans during query execution might be used (similar to the approach described in [27]). We do not report on these optimizations in our evaluation.

3.4 Query Re-planning

Not all the data sources relevant to a query may be known at initial planning time. New data sources may be discovered during the execution of the query plan (we discuss three ways in which new sources may be discovered in chapter 4). In such situations, it is desirable to *re-plan* the query without causing query execution to start over and losing results which have already been computed. This re-planning process is responsible for rewriting the query plan in such a way that it yields the same results as would have been produced had the new data source been available at the time of initial planning. We call this process *query re-planning*.

3.4.1 Query Planning Hooks

To allow the query re-planner to easily re-write the query plan, we alter the initial planning approach described in the previous section. In addition to the initial set of data sources (provided by either the user or the system), we add an *empty* source labeled "$\varnothing$". During query execution, any plan operator labeled with the empty source is treated as a no-op: it returns no data. For example, a triple pattern P with data sources labelled "1" and "2" would (before rewriting) result in the plan $(P^1 \cup P^2 \cup P^{\varnothing})$. Plan operators labelled with the empty source are referred to as "hooks" and left in the query plan for use during re-planning.

3.4.2 Query Plan Rewriting

When new data sources become available by way of source augmentation (described in the next chapter), the re-planner is responsible for rewriting the query plan to incorporate the new sources. To ensure that generated results are complete, this rewriting must produce a query plan that is equivalent to one that would have been produced had all sources been available at the time of initial planning.

The query plan rewriting process is shown in Algorithm 8 in Appendix B. The function $expandQueryPlan(p, S)$ takes a query plan p and a set of data sources S as input, and returns a new query plan that extends the evaluation of the query to include intermediate results provided by the data sources in S. It may be noted that the implementation of *expandQueryPlan* described in Algorithm 8 removes subplans that do not reference the sources in S or $\varnothing$ (accessing the list of sources used in a subplan with the *ChildSources* function). This constraint is based on the decision to perform query rewriting *after* a query plan has been completely evaluated (as described in section 3.5). This is purely a pragmatic decision that simplifies the description and implementation of the rewriting process; it may be relaxed to provide a rewriting process applicable at any time during the evaluation of a query plan.

3.5 Query Execution

With the ability to produce an initial query plan, discover new data sources, and to rewrite the query plan to incorporate those data sources, query execution is simply a matter of synthesizing these pieces. Algorithm 1 provides pseudocode for this process. An initial query plan, p, is generated by the initial planner (leaving hooks for the re-planner in the form of plan operators labelled with source $\varnothing$). p is evaluated, returning a set of query results, R, which are output and a (possibly empty) set of newly available data sources, *pendingSources*. If *pendingSources* is empty, evaluation is finished and terminates. If *pendingSources* contains new data sources, the re-planner is invoked with the *expandQueryPlan* function to rewrite the query plan. Once the plan is rewritten to incorporate the new data sources, the process of plan evaluation is repeated.

Algorithm 1: Query Plan Execution Algorithm

```
   Input: SPARQL query Q, initial data sources S
1  S' ← S∪{∅} ;
2  p ← InitialPlanner(Q, S') ;
3  pendingSources ← ∅ ;
4  repeat
5  |  (R, pendingSources) ← Evaluate(p) ;
6  |  output R ;
7  |  p ← expandQueryPlan(p, pendingSources) ;
8  until pendingSources = ∅;
```

The algorithm as shown outputs results only after the evaluation of each query plan. We make this simplification for the sake of readability. In reality, the output of query results may be pipelined by the *Evaluate* function, allowing results to be output as soon as they are computed. Such pipelining is possible if the query execution algorithm is implemented as an iterator or co-routine, or by outputting results using IO that relies on system-level parallelism to allow non-buffered access to the query results.

Finally, it is worth noting that the plan rewriting that occurs during query evaluation is greatly affected by the size of *pendingSources*. If a large number of sources are discovered during one round of query evaluation, *expandQueryPlan* will

produce a rewritten query plan that is much larger than the original. In section 7.2.3 we discuss the potential to limit the effects of this explosion in plan size on overall performance.

3.6 Summary

In this chapter, we presented the architecture for a federated SPARQL query planner optimized for producing early results. This optimization uses a "push-down" heuristic that benefits from data matching different query patterns being co-located at individual data sources. We also showed how the introduction of an *empty* data source during the query planning process enables query plan expansion during the re-planning phase in order to incorporate newly discovered data sources and allowing for an expanded query result set. Finally, we presented an algorithm that is able to use these components to iteratively execute and expand a query plan in order to produce query results.

Now that we have presented the architecture of the federated query planner, in the next chapter we describe how new data sources may be discovered during the execution of query plans, and thus provide the query re-planning system with input for plan expansion.

CHAPTER 4
DATA SOURCE AUGMENTATION

As query plans are evaluated to produce results, the *source augmentation* framework is responsible for generating a list of new data sources that will be provided to the query re-planner. This framework is designed as an extensible system of functions that operate over the query evaluation process and produce new data sources in various ways.

In the following sections we discuss three augmentation strategies, *SPARQL link relation augmentation*, *linked data augmentation*, and *RDF content augmentation*, which are able to discover sources of linked data and new SPARQL endpoints. We define three concrete functions that are able to identify new data sources using these strategies.

Definition The set of *SPARQL data sources*, S, is the union of SPARQL endpoints, $E \subset I$, implementing the SPARQL Protocol and the set of RDF graphs, G, available locally to the federation client through a query API.

Let Q be a query plan used in one round of federated query evaluation (produced by either the initial query planner or the query re-planner) and $\mathcal{E}(Q)$ be the abstract evaluation of Q by the federation client. That is, $\mathcal{E}(Q)$ represents the entire process of evaluating Q. This may be thought of as being comprised of the set of instantaneous snapshots of the execution process (including instruction set, program counter, stack, and heap).

Definition A *data source discovery function* is a function $d : \mathcal{E}(Q) \mapsto S$. Given the domain $\mathcal{E}(Q)$, such a function may compute the set of data sources S in any way based on the evaluation of the query. This includes returning constant values or extracting data sources from the data returned by other data sources. We define three such functions below.

Definition *Data source augmentation* is the process of computing newly discovered SPARQL data sources using all available *data source discovery functions* (D),

$pendingSources = \bigcup_{d \in D} d(\mathcal{E}(Q))$, and using them as input to the query re-planner using the *expandQueryPlan* function as shown in Algorithm 1.

In this chapter we define three *data source discovery functions* used by the federation framework to expand the set of data sources that may contribute to query results. Given the evaluation of a query plan, two of these functions are able to discover SPARQL endpoints for inclusion in the next round of re-planning while the third aggregates RDF data encountered during the query evaluation into a local RDF graph and provides SPARQL API access to it.

4.1 SPARQL Link Relation Augmentation

The *SPARQL link relation augmentation* strategy attempts to discover existing SPARQL endpoints referenced by the endpoints already known to the system. Existing SPARQL endpoints may indicate the existence of related SPARQL endpoints in many ways including through SPARQL Service Description[44] metadata, in HTTP headers, or encoded in the SPARQL Results content. These endpoints may be discovered through observation of the communication between the federation client and remote endpoints.

We propose the use of HTTP entity-header link relations[45] as the method used for indicating a set of related SPARQL endpoints. By introducing link relations between SPARQL endpoints, we bring the same hyperlinking benefits of linked data to SPARQL endpoints and their results. This has a similar effect on the discoverability of SPARQL endpoints as existing work on self-description has brought to RDF resources (e.g. the "hypermedia RDF" work of Kjernsmo[46]). For example:

```
Link: <http://dbpedia.org/sparql>; rel="sparql"
```

The use of the "sparql" link relation in this case is used to indicate to the federation client that `http://dbpedia.org/sparql` is the URL of a SPARQL endpoint that may provide relevant related data to those returned in the query result. As opposed to including such links in a service description document, endpoints making use of such link relations in the response to a query are free to arbitrarily determine the set

of endpoint links to include. If appropriate metadata indexes were available during the computation of query results, only endpoints known to be relevant to the resources in the result set might be linked. Alternatively, a fixed set of endpoints might be included in every response (removing the cost of computing relevant endpoints).

When a new SPARQL endpoint is discovered by SPARQL augmentation, it is made available to the query re-planner for use during plan rewriting. Optionally, upon discovery, metadata about the endpoint may be collected (e.g. by accessing the endpoint's service description) to improve the effectiveness of the cost model during plan rewriting.

4.1.1 Example of SPARQL Link Relation Augmentation

To demonstrate the operation of SPARQL link relation augmentation using HTTP headers, consider a sub-plan being evaluated for the pattern

```
?paper foaf:topic dbpedia:Semantic_Web
```

against a SPARQL endpoint source at `http://example.org/sparql`. The HTTP client-server interaction for this query using the SPARQL Protocol is:

```
> GET /sparql?query=PREFIX%20foaf%3A%20%3Chttp%3A%2F%2Fxmlns.com%2F
  foaf%2F0.1%2F%3E%0APREFIX%20dbpedia%3A%20%3Chttp%3A%2F%2F
  dbpedia.org%2Fresource%2F%3E%0ASELECT%20%2A%20WHERE%20%7B%20
  %3Fpaper%20foaf%3Atopic%20dbpedia%3ASemantic\_Web%20%7D HTTP/1.1
> Host: example.org

< HTTP/1.1 200 OK
< Link: <http://dbpedia.org/sparql>; rel="sparql"
< Content-Type: application/sparql-results+xml
<
< <?xml version="1.0"?>
< <sparql xmlns="http://www.w3.org/2005/sparql-results#">
<  <head><variable name="paper"/></head>
<  <results>
```

```
<    <result><binding name="paper">...</binding></result>
<  </results>
< </sparql>
```

Upon parsing this SPARQL results document, `http://dbpedia.org/sparql` is passed to the query re-planner as a new SPARQL data source.

4.2 Linked Data Augmentation

The *linked data augmentation* strategy attempts to dereference URIs found in intermediate results in an attempt to discover RDF content related to those URIs. When RDF content is discovered in this way, it is loaded into a local triplestore and made available as a data source. In an attempt to reduce the number of data sources constructed in this way, each of which will lead to growth in the rewritten query plan, all RDF discovered between re-planning rounds is loaded into a single local triplestore.

Algorithm 9 in Appendix B gives a simple overview of how intermediate results are handled by the linked data augmentation function *inspectResult*. This function is called whenever a query result structure is created, either during parsing of a SPARQL result document, or during query evaluation using a local triplestore. Additionally, URIs present in the query string are dereferenced and used for augmentation before query execution begins.

The linked data augmentation function, in the absence of any available SPARQL endpoints, results in a query answering system similar to linked data query systems such as SQUIN[10]. As Hartig, Bizer, and Freytag show, this can be an effective approach to generating query results when only linked data sources are available. However, as we discuss in chapter 6, this approach has inherent performance challenges when executing over a range of query patterns and data sources. These challenges make linked data augmentation unsuitable as a general purpose solution. Further, as it depends on RDF collected through linked data augmentation, RDF content augmentation (described below) shares these performance characteristics.

The ability to improve performance and data coverage by leveraging the presence of SPARQL sources in an otherwise linked-data-only scenario makes this ap-

proach potentially more performant and much more flexible than existing linked-data-only solutions. We present an initial evaluation of linked data augmentation in section 6.3.3, but leave a detailed analysis of the conditions of its general applicability to future work.

4.2.1 Example of Linked Data Augmentation

Here we provide a complete example of linked data augmentation providing relevant data to the federation system using the query `?paper foaf:topic dbpedia:Semantic_Web ; foaf:maker ?person`, and two data sources, "1" and "2". Consider a case in which during the execution of the sub-plan $topic^1$, retrieving data matching the `foaf:topic` triple at source 1, the data source returns the intermediate result { `?paper` ↦ `<http://example.org/papers#paper1>` }. Upon parsing this result from the SPARQL Protocol response, the linked data augmentation function will cause the following steps to be performed:

1. Dereference `<http://example.org/papers#paper1>` using the HTTP GET operation, obtaining data pertaining to the publication identified by this IRI:

```
> GET /papers HTTP/1.1
> Host: example.org

< HTTP/1.1 200 OK
< Content-Type: text/turtle
<
< @prefix foaf: <http://xmlns.com/foaf/0.1/> .
< <http://example.org/papers#paper1> foaf:name "..." ;
<     foaf:maker <http://example.org/people/alice> .
```

2. Parse the returned data into a local RDF graph (in this case from the Turtle serialization of RDF into two triples identifying a name and maker of the paper);

3. Construct a data source representation of the graph which provides SPARQL API access to the underlying RDF data;

4. Return this newly constructed data source to the augmentation system, which will use it in calling the *expandQueryPlan* function, incorporating it into the query plan and labeling it as, e.g., source "3";

5. This new data source will, after re-planning, provide one answer to the *maker* triple pattern, causing one query result to be generated from the sub-plan $topic^1 \bowtie maker^3$.

4.3 RDF Content Augmentation

The *RDF content augmentation* strategy attempts to discover SPARQL endpoints referenced in RDF content available to the federation system. Existing RDF vocabularies contain terms relevant to this process. Kjernsmo[46] proposes the use of the VoID[13] vocabulary terms `inDataset` and `sparqlEndpoint` to express the relationship between a resource and a SPARQL endpoint which contains information about it:

```
<resource> void:inDataset [ void:sparqlEndpoint </sparql> ] .
```

We take this approach, allowing any resource to be described as having descriptive data loaded in an identified endpoint.

As with SPARQL augmentation, when a new endpoint is discovered by RDF content augmentation, it is made available to the query re-planner for use during plan rewriting.

4.3.1 Example of SPARQL Augmentation using RDF Content

To demonstrate the operation of RDF content augmentation, consider a sub-plan that results in the intermediate result:

{ `?paper` $\mapsto$ `<http://example.org/papers#paper2>` }

Using the linked data augmentation function on this intermediate result (as described in section 4.2.1) results in the retrieval of RDF content:

```
<http://example.org/papers#paper2> foaf:name "..." ;
  foaf:maker <http://example.org/people/alice> ;
  void:inDataset [ void:sparqlEndpoint <http://dbpedia.org/sparql> ]
```

Upon parsing this RDF data, the `void:sparqlEndpoint` triple is recognized as providing source augmentation data, and `http://dbpedia.org/sparql` is passed to the query re-planner as a new data SPARQL source.

4.4 Summary

In this chapter, we defined *data source discovery functions* and their use in *data source augmentation*. We describe how data source augmentation is used by the query plan execution algorithm to iteratively expand the query plan in an attempt to expand the result set.

Although the set of resources described by data sources often overlap, it is often difficult to determine the topology of this sharing network. Even in cases where there is explicit knowledge of how datasets connect (as in the Linking Open Data project[47]), this knowledge is often not represented in a structured format that is discoverable through interaction with the data sources. In order to enable practical use of the network topology of data sources that share resources, we defined three discovery functions and discuss the metadata necessary to their use.

In the next chapter we address the issue of performance related to repeatedly evaluating sub-plans, both within and between discreet query executions.

CHAPTER 5
CACHING OF SPARQL QUERY RESULTS

In the process of evaluating federated queries, as described in the previous chapters, many small subqueries are executed against remote endpoints using the HTTP-based SPARQL protocol. Caching query results using the features of HTTP can dramatically improve the performance of query evaluation, both intra-query (where a subquery appears repeatedly in a query plan and cannot be materialized) and inter-query (in federated workloads where many queries share a common graph pattern).

When a client uses a conditional HTTP request to which the server responds with a "Not Modified" message, only the IO for the response header is required. On the server, validating a conditional request is likely to be faster and require fewer resources (both CPU time and working memory) than evaluating the whole query. This allows the server to respond to more and/or more complex queries given fixed resources (or, conversely, response to the same queries with fewer resources). Moreover, if successfully validating a conditional request is faster than evaluating the query, the client benefits not only from reduced IO but also reduced latency and potentially by avoiding the need to parse the response (if a client's local cache is able to store a parsed representation).

The benefits of caching are only realized if both the client and server support the caching protocol and if requests are repeatedly made for already-cached results. Client-side support for caching is already available due to the widespread support in HTTP libraries that are used to implement the SPARQL protocol. Below, we show how to enable support for caching in the data structures used on the server. Because of the widespread support for HTTP caching, and the high frequency of repeated queries in real-world workloads (as discussed in [31]), caching of SPARQL

Portions of this chapter previously appeared as: Gregory Todd Williams and Jesse Weaver. Enabling fine-grained HTTP caching of SPARQL query results. In *ISWC'11: Proceedings of the 10th International Semantic Web Conference*. Springer-Verlag, October 2011.

query results has the potential to significantly improve efficiency. We restrict our work to only consider caching at the HTTP level as the standard SPARQL Protocol is defined in terms of HTTP.

5.1 HTTP Caching

In this section we introduce the caching features available in HTTP[48] upon which our system relies.

HTTP supports two primary caching mechanisms, allowing servers to explicitly indicate a caching expiration (with an `Expires` date or a `max-age` duration) or indicating a cache validator (with a `Last-Modified` date or `ETag` value). Here we concern ourselves only with cache validators – specifically, `Last-Modified` dates – as they are a more natural fit for caching data that may be updated in the database at any time. However, as they relate to our work, both the `Last-Modified` and `ETag` headers may be understood as being effectively equivalent as we do not use the more expressive "weak validation" that ETags allow.

The `Last-Modified` validator works as follows. A client user-agent requests a resource (in this case the results to a SPARQL query) from the server:

```
GET /sparql?query=SELECT... HTTP/1.1
Host: example.org
```

The server sends back a response whose header includes the `Last-Modified` validator with a date indicating when the resource was last modified:

```
HTTP/1.1 200 OK
Last-Modified: Wed, 1 Jun 2011 12:45:15 GMT
Content-Type: application/sparql-results+xml

<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
...
</sparql>
```

At some point in the future, the client requests the resource again and, noting that the response is cached from the last time it was requested, indicates the request as *conditional* by using the `If-Modified-Since` header with the previously returned validator date:

```
GET /sparql?query=SELECT... HTTP/1.1
Host: example.org
If-Modified-Since: Wed, 1 Jun 2011 12:45:15 GMT
```

If the resource has not changed since the validator date, the server sends a response indicating that the already-cached content is still valid:

```
HTTP/1.1 304 Not Modified
```

Otherwise, the server responds as usual with a full response, including the resource content and any applicable cache validators (the updated `Last-Modified` time).

In terms of SPARQL, HTTP caching ought to make query results appear as valid ("fresh") so long as the query results do not change. Since determining precisely if results to a query have changed may require re-evaluating the query (negating one of the benefits of caching), we settle for a less strict condition: caching ought to make the query results appear as valid so long as data "relevant" to the query have not changed. Once a query result has been cached, updates to "irrelevant" data in the SPARQL endpoint should have no affect on the caching – upon re-submitting the query, the server should indicate that the cached results are still valid. "Relevant" data being updated prior to the query being re-submitted should result in the server re-evaluating the query and returning fresh query results. In section 5.2.2 we define "relevant" and "irrelevant" data with respect to a query.

5.2 Enabling Query Result Caching

We propose modifying the search trees used to index RDF data in SPARQL endpoints in a simple way to enable determining the effective modification time of data relevant to a query. In the following sections we show how the modification

time data stored in the search trees can be maintained during database updates, and how the data can be retrieved and used at query time. In determining what data is relevant to a specific query, we extend the work done by Martin, Unbehauen, and Auer[39] (what they call "Graph Pattern Solution Invariance") to support the much more expressive queries and graph patterns of SPARQL 1.1[3]. This includes the use of named graphs, property paths, and `DESCRIBE` queries.

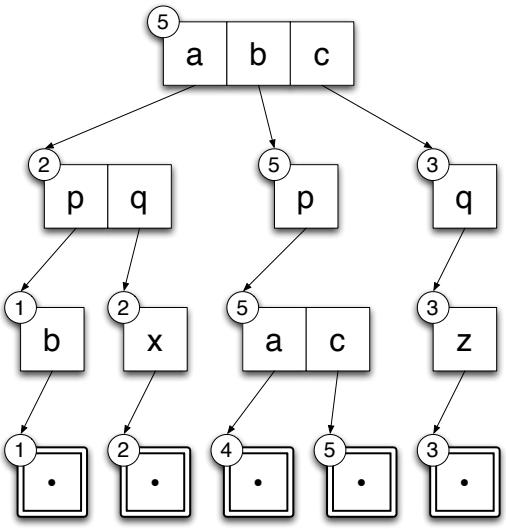
In this work we assume that the SPARQL processor is built using a quad-store, and that the SPARQL *RDF Dataset* is mapped directly to the statements in the quad-store (with a special graph name representing the default graph). This assumption simplifies the following discussion, but is not required by our approach. The algorithms we present can be extended to work with arbitrary mappings between RDF dataset and quad-store, or to work with graph-stores.

5.2.1 Search Tree Indexing

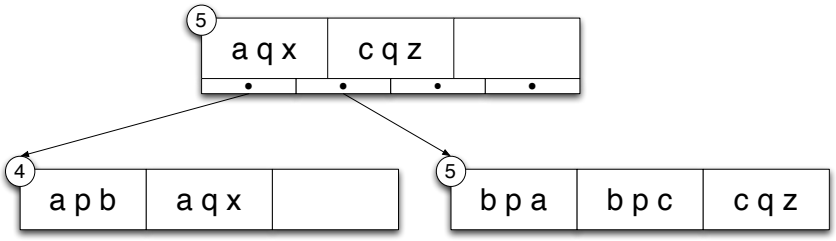
Search trees are a common data structure used to implement efficient access to RDF data for varying access patterns. To determine the modification-time (mtime) of "relevant" data in a search tree, we propose adding an mtime field to each node in the search tree. During an update operation (insertion or deletion of RDF data), we update the mtime field in each affected search tree node. Moreover, during an update we ensure the mtime of each node in the tree is greater than or equal to the mtime of all of its children. In this way, the mtime of a node in the tree can be used as a conservative proxy value for the mtime of any of its descendant nodes.

For each access pattern in a query, we can now determine an mtime after which we can be assured that no update operation has affected data matching the pattern. By calculating the maximum mtime over all the query access patterns, we arrive at a single timestamp which is at least as recent as the actual modification time of the query's "relevant" data.

While different SPARQL systems make use of different types of search trees, and use varying numbers of indexes, we propose a general solution that works with any number of indexes and across a variety of tree types (we discuss specifics of both B+ trees and tries). Although the caching results in our proposed system



(a) Trie Index



(b) B+ Tree Index

Figure 5.1: Example $\langle SPO \rangle$ search trees

are complete, soundness is affected by the choice of a specific search tree type and number of available indexes. For example, fewer indexes, or the use of B+ trees versus tries, may cause some query results to appear as if they have changed when in fact they haven't. However, query results that are asserted as being the same as cached results will always in fact be the same.

Figure 5.1 shows both a B+ tree of order 4 and a trie with example data[2]. The mtime of each tree node appears inside the circle attached to each node, and shows the mtimes that result from this example 5-update sequence (with mtimes starting at 1, and incrementing on subsequent operations):

1. Insert triple { `a p b` }

[2]The example data used here is comprised of triples for brevity; the handling of mtimes is identical for quad data.

2. Insert triple { `a q x` }
3. Insert triple { `c q z` }
4. Insert triple { `b p a` }
5. Insert triple { `b p c` }

As can be seen, the trie maintains the correct mtime for each leaf node while the mtimes in the B+ tree only indicate that the triples with subject `a` were not updated by operation 5. We discuss the reasons for this unsoundness in section 5.2.3.2.

5.2.2 Relevant data and graph patterns

We define data "relevant" to a SPARQL query as being data that *may* affect the results of the query. Martin, Unbehauen, and Auer claim: "the solution of a graph pattern stays the same at least until a triple, which matches any of the triple patterns being part of the graph pattern, is added to or deleted from the [graph]"[39]. This is true when considering only *triple*-patterns in the default graph, but to support the full expressivity of SPARQL, we must extend this claim: The solution of a query with a graph pattern stays the same at least until:

- a triple, which matches any of the triple patterns being part of the graph pattern, is added to or deleted from the default graph
- a triple, which matches any of the triple patterns being part of a `GRAPH <iri>` pattern, is added to or deleted from the `<iri>` named graph
- a triple, which matches any of the triple patterns being part of a `GRAPH ?var` pattern, is added to or deleted from any named graph
- a triple is added to a new named graph and the graph pattern includes an empty `GRAPH ?var` pattern
- a triple is removed from a named graph, leaving the graph empty, and the graph pattern includes an empty `GRAPH ?var` pattern

- a triple, with predicate matching any part of a property path being part of the graph pattern, is added to or deleted from the dataset
- a triple is added or removed from the dataset, and the graph pattern includes a zero-length or negated property path
- a triple is added to or deleted from the dataset, and the query uses the `DESCRIBE` form

We discuss each of these cases and how they relate to relevant data below.

5.2.2.1 Named Graph Patterns

As noted in [39], the addition or deletion of a triple (to the default graph) may change the solutions of a graph pattern. To fully support SPARQL datasets (which contain not just the default graph, but also any number of named graphs), we must also consider graph patterns scoped to a named graph. These patterns may either be scoped to a specific named graph (using the `GRAPH <iri> { ... }`) syntax) or be scoped to *any* named graph (using the `GRAPH ?var { ... }` syntax). For a graph pattern scoped to a specific named graph, `iri`, the solutions to the graph pattern may change with the addition or deletion of a triple matching the graph pattern to the named graph `iri`. For a pattern scoped to any named graph, the solutions to the pattern may change with the addition or deletion of a triple matching the graph pattern to any named graph.

5.2.2.2 Empty Named Graph Patterns

Beyond graph patterns scoped to named graphs, special handling is required for the *empty* named graph pattern:

```
SELECT ?g WHERE { GRAPH ?g {} }
```

This query returns the set of graph names in the dataset. The query has no triple-patterns which might match triples being added or removed, yet its results may change based on added or removed data. Specifically, adding a triple to a new

named graph, or removing the final triple from a named graph[3] may change the solutions to this pattern.

5.2.2.3 Paths

Property paths greatly increase the expressiveness of SPARQL, but as they relate to relevant data, may be reduced to the matching of triple patterns. We can partition property paths into two categories: fixed-length and variable-length. Fixed-length property paths are those that can be syntactically represented by basic graph patterns (BGPs). Due to their equivalence with BGPs (sets of triples), these paths can be handled in the same manner as triple patterns.

Variable-length paths are those that cannot be reduced to BGPs, and may rely on new algebraic operations to match data. These include zero-or-more paths (`?s <p>* ?o`), one-or-more paths (`?s <p>+ ?o`), and negated paths (`?s !<p> ?o`). Due to their complexity, we discuss only a subset of the expressivity of these path types.

With respect to "relevant" data, one-or-more paths with simple predicates (those in which the `+` path operator applies to an IRI) can be reduced to triple pattern matching with predicate-bound triple patterns. For example, the path `<s> <p>+ ?o` has the same relevant data as the triple pattern `?s <p> ?o`. Note that while the subject is bound to `<s>` in the path pattern, it must be unbound in the triple pattern equivalent as the path may be affected by triples where the subject is not `<s>`.

Zero-length paths and negated paths require special attention. The zero-length path connects a graph node (any subject or object in the graph) to itself. Therefore, any insertion (deletion) in a graph may affect the results to a zero-length path pattern by adding (removing) a node to the graph that didn't exist before (doesn't exist after) the update. Similarly, a negated path `?s !<p> ?o` implies that any insertion or deletion *not* using the `<p>` predicate may impact the results. While the relevant data for such a negated path is a subset of all the data, we assume that realistic datasets will contain a range of predicates and so the relevant data will be

[3]Some SPARQL systems allow empty named graphs to exist. Removing all triples from a named graph would not affect the set of graph names on such systems.

very large (in many cases approximating the size of the dataset itself). Therefore, we conservatively assume that the entire dataset is relevant to a negated path pattern.

5.2.2.4 DESCRIBE Queries

`DESCRIBE` queries present a challenge in determining relevant data. These queries involve matching a graph pattern just as `SELECT` queries do (a `DESCRIBE` query without a `WHERE` clause being semantically equivalent to one with an empty `WHERE` clause). However, the final results of a `DESCRIBE` query depend on the `WHERE` clause *and* the algorithm used for enumerating the RDF triples that comprise the description of a resource.

A naïve `DESCRIBE` algorithm would be to return all the triples in which the resource appeared as the subject. For our purposes, this algorithm would make this query:

```
DESCRIBE ?s
WHERE { ?s a <Class> }
```

roughly equivalent to a `SELECT` query with an additional triple pattern:

```
SELECT ?s ?p ?o
WHERE { ?s a <Class> .
        ?s ?p ?o }
```

Since most `DESCRIBE` algorithms will include *at least* these triples, and given the course-grained nature of the triple pattern `?s ?p ?o` (matching every triple in the database), we consider the "relevant" data for a `DESCRIBE` query to be all data in the database. We note that the work in [39] does not (and need not) address this issue as that work is concerned with caching of *graph pattern* results, not *query* results. Since the `DESCRIBE` query form takes *graph pattern* results (or ground IRIs) as input, and outputs an implementation-dependent set *query* results, the caching of *query* results must respect this process.

Given that the algorithm used for `DESCRIBE` queries is implementation dependent, our definition of "relevant" data for `DESCRIBE` queries is intentionally conser-

vative and we do not discuss specific handling of `DESCRIBE` queries in any further detail.

5.2.3 Maintaining and Probing Cache Status

In this section we describe the algorithms used during update operations to maintain the mtime field in the search tree. We then describe the probing algorithm used to determine the effective mtime of the relevant data for a specific query.

5.2.3.1 Cache Maintenance

Maintaining the mtime field in the search tree is a simple process:

1. Before each tree node is written to disk (due to an insertion of deletion), update the node's mtime to the current time.

2. For each node that is written to disk, write its parent to disk (thereby updating its parent's mtime).

This process will ensure our condition that every tree node's mtime is greater or equal to those of its descendants and can be used as the effective mtime of descendant, relevant data.

We distinguish between the effective mtime of data matching an access pattern, and that data's actual mtime. As discussed in section 5.2.1, the specific data structure used for the search tree affects the granularity (and therefore the expected accuracy) of the effective mtime. Due to their design, tries yield effective mtimes that are exactly the same as the most recent mtime of data matching an access pattern. B+ trees yield effective mtimes of matching data that may be affected by any non-matching data that is co-located on a leaf node with matching data.

During the update process, we note that the parent node(s) may already need to be written to disk (in the case of a node split), so step 2 may already be required on any given update. Moreover, an update at a leaf node in append-only and counted B+ trees cause a cascade of writes up to the (possibly new) root. In these cases, all IO incurred by the cache update algorithm is already required by the update operation, and so the cost of maintaining the cache data is effectively zero.

5.2.3.2 Cache Probing

The algorithm used for probing a database index to retrieve the effective mtime for a query is shown in algorithm 2. Given a query and a set of available search tree indexes, for each access pattern in the query, the algorithm probes the index that will yield the most accurate effective mtime, and returns the most recent of the mtimes. The index that will yield the most accurate effective mtime is the one with a key ordering that will allow descending as deep into the tree as there are bound terms in the access pattern. If no such index exists, a suitable replacement index is chosen that maximizes the possible depth into the tree that some subset of bound terms in the access pattern will allow. In the case of the completely unbound access pattern, the effective mtime is the same as the mtime of the entire dataset and so can be retrieved from the root node of *any* available index.

While this algorithm describes how the effective mtime of a query may be computed, it is worth noting that the specific steps described may be implemented in more or less efficient ways. For example, the algorithm calls for finding the lowest common ancestor (LCA) of data matching the access pattern. For a system using B+ trees, a naïve implementation might traverse the tree to find the leaves with matching data and then walk up the tree to find the LCA. A more efficient implementation avoids having to find all leaves with matching data by traversing tree edges until finding the LCA by using the bounds data contained in internal nodes.

As discussed above and in section 5.2.1, the soundness of results is affected by the choice of the search tree data structure used. B+ trees produce less sound results as a result of maintaining less accurate effective mtimes. Tries will result in more sound results as a result of being able to maintain accurate effective mtimes. Even though tries maintain accurate effective mtimes, their use does not guarantee perfectly sound cache validation as updates that affect data relevant to a query may not change the results to that query. This can occur when the relevant updated data does not appear in the query results due to join conditions, filter expressions, or projection. In these cases, cache validation will fail and the query must be evaluated again, despite accurate results already being cached.

Algorithm 2: Probe database for effective mtime of query results

Input: A SPARQL query graph pattern *query*, a set of available database indexes *indexes*

Output: $effectiveMtime$, the effective modification time of relevant data for the query

```
1  mtimes = ∅ ;
2  foreach ap ∈ query do
3      orderedIndexes = {i | i∈indexes, ∃s⊆boundPositions(ap) s.t. the key
       order of i starts with s} ;
4      if |orderedIndexes| > 0 then
5          index = argmax_{i∈orderedIndexes} |s| ;
6          n = LCA of data matching ap in index ;
7          mtimes = mtimes ∪ {mtime(n)} ;
8      else
9          i = any index in indexes ;
10         mtimes = mtimes ∪ {mtime(root(i))} ;
11     end
12 end
13 effectiveMtime = Max(mtimes) ;
14 return effectiveMtime
```

One final case that is worth noting is the special case of determining the effective mtime for an *empty* named graph pattern (GRAPH ?g {}). As discussed in section 5.2.2.2, this pattern returns the set of available named graphs. If the set of available indexes are all covering indexes (using key orders that are just permutations of subject, predicate, object, and graph), then there is no way to determine an accurate effective mtime for this pattern. However, if there is an available index over just $\langle G \rangle$, an accurate effective mtime for the set of named graphs is stored in the $\langle G \rangle$ index root node.

5.3 Summary

The ability of HTTP caching to increase performance of SPARQL query evaluation is obvious. Yet caching does not enjoy widespread use in SPARQL implementations. We suggest that the biggest impediment to such use has been the potentially high cost and complexity of maintaining caches of SPARQL query results. To overcome this challenge, we showed in this chapter how modifying search tree

indexes can allow a SPARQL implementation to efficiently determine the effective modification time of data relevant to a query and presente the probing algorithm to accomplish this. We also described how the modification time data may be simply maintained during update operations.

The caching system we described maps naturally to the expressivity of validator-based caching, and so fits well with the HTTP-based SPARQL Protocol. In the next chapter we will present an evaluation of our federation system, and show how the use of HTTP caching in evaluating query sub-plans can dramatically improve performance, both within the span of a single query execution, and across query executions with the use of a persistent cache.

CHAPTER 6
IMPLEMENTATION AND EVALUATION

In the following sections, we evaluate the performance of federated query planning and evaluation under a variety of queries and data distributions. We also evaluate the performance impact of caching SPARQL query results on federated queries.

All evaluation was performed with SPARQL endpoints hosted on a Dell PowerEdge T610 machine with dual Intel Xeon E5504 Quad-core 2.0GHz CPUs, 24GB RAM, and a 160GB 7200RPM SATA hard disk. The federation client was hosted on an Apple 27-inch, Mid 2010 iMac with a Intel Core i7 Quad-core 2.93GHz CPU, 16GB RAM, and 250GB Apple Solid State storage. The machines were connected with gigabit ethernet across a local area network, with maximum sustainable bandwidth of 350 megabits per second. However, maximum bandwidth utilization seen between the federation client and SPARQL endpoints during evaluation (with the use of standard HTTP protocol compression) was 3.3 megabits per second.

Our evaluation was performed using an implementation of the described federation client written in a combination of C and Objective-C. We rely on many of the system libraries available on the Mac OS X operating system and Objective-C runtime environment. Of particular note, this includes the use of system libraries for HTTP message construction and network operations, including HTTP compression and caching support. The remote endpoints in our evaluation environment were constructed using a combination of Jena Fuseki 0.2.5 and custom C implementation supporting caching validators as described in [49].

6.1 Evaluation Queries

We performed evaluation using four queries over the bibliographic data. These queries are meant to span a range of query features and reveal strengths and weaknesses of the evaluated features and systems. The queries are summarized below, with the full queries included in Appendix C.

Two queries (**Q3** and **Q4**) are based on queries in the SP^{2}Bench SPARQL

benchmarking work of Schmidt et al.[50]. We chose these from the SP^{2}Bench benchmark queries as they focus on basic graph pattern queries without the need for more complex query operations such as union and left-join (as such operations are built upon the basic graph pattern operation).

Q1 *Select all publications and their titles for a specific author's name*
This is a path-shaped query exhibiting moderate selectivity;

Q2 *Select all publications and their titles for a specific author's IRI*
This is a star-shaped query exhibiting high selectivity;

Q3 *Finds all documents with property* `swrc:isbn` *(based on SP^{2}Bench* **Q3c***)*
This is a star-shaped query exhibiting overall moderate selectivity despite being comprised of very low selectivity triple patterns.

Q4 *Find all subjects that stand in any relation to Paul Erdös (SP^{2}Bench* **Q10***)*
This is a "lookup" query of a single triple pattern exhibiting high selectivity

These four queries capture the fundamental expressivity of basic graph patterns, ranging from single triple patterns, through patterns requiring joins, to more complex patterns requiring multiple joins and join variables. The range of selectivity and total number of query results generated from these queries provides the ability to evaluate the federation and augmentation systems ability to cope with differing execution characteristics such as low-latency queries (where query planning and augmentation has the potential to dominate total query time) and high latency queries in which many sub-plans do not produce any results (where good planning heuristics and caching are crucial in producing early results).

6.2 Evaluation Dataset

Our evaluation was performed on locally hosted copies of bibliographic data publicly available from the Linking Open Data project[47]. The datasets used were:

- Association for Computing Machinery (ACM) (RKBExplorer)[51]
- CiteSeer (Research Index) (RKBExplorer)[52]

- DBLP Computer Science Bibliography (RKBExplorer)[53]
- DBLP in RDF (L3S)[54]
- Semantic Web Dog Food Corpus[55]

These datasets collectively describe a variety of academic publications, their authors, citations, abstracts, and keywords. A summary of the size of each of these datasets and the number of people and publications in each is given in Table 6.1.

Dataset	Triples	People	Publications
ACM	8,260,550	933,921	643,067
CiteSeer	5,059,481	452,709	406,436
DBLP-RKB	24,183,312	2,975,148	1,919,509
DBLP-L3S	55,027,643	1,171,118	2,071,216
SemWeb-DogFood	260,055	8,883	3,752

Table 6.1: Evaluation Dataset Statistics

So that the focus of our evaluation is the federation architecture and augmentation system described in the previous chapters, two variants of these datasets were constructed using different partitioning strategies: *horizontal* and *vertical*. In the *horizontal partitioning* strategy, data from each dataset was published together as-is, with the exception that vocabulary alignment was performed. This allows the evaluation of a single federated query without any query-time schema mapping (a topic outside the scope of this work). IRIs of the resources described by each dataset are lexically scoped to the dataset (e.g. publications in the ACM dataset use ACM IRIs such as `http://acm.rkbexplorer.com/id/630623`). SPARQL endpoints were made available for each dataset, and configured to link query results to related endpoints using link relations[45] in the same topology in which these datasets are actually linked (as shown in figure 6.1). Resource IRIs were also configured to respond to linked data lookups (using HTTP GET operations) with all RDF triples for which the specified resource was the triple subject.

In the *vertical partitioning* strategy, IRIs of resources for authors and publications were rewritten so that they are shared between datasets. No vocabulary

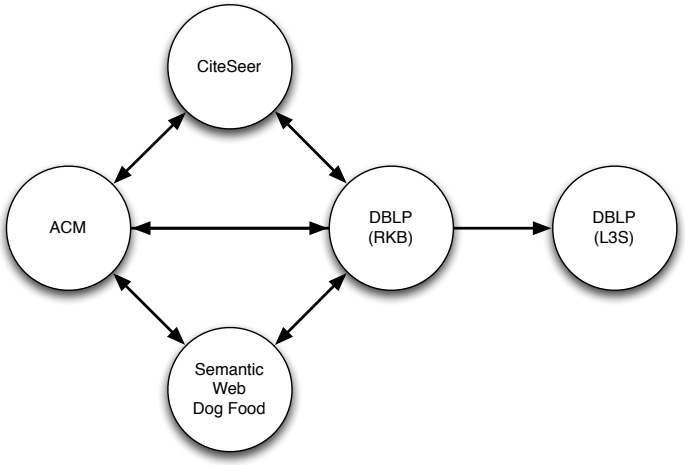


Figure 6.1: Bibliographic Dataset Endpoint Link Topology

alignment was performed, meaning a single resource is described by different vocabulary terms in different data sources. This allows evaluation of cases where endpoints using different vocabularies contain data about shared resources (as our evaluation datasets do). As in the horizontal partitioning case, SPARQL endpoints were configured to link to related endpoints using the topology in figure 6.1 and resource IRIs were configured to response to linked data lookups.

We refer to the evaluation of a query using a particular data partitioning using the query number followed by either **H** or **V**, indicating the **horizontal** or **vertical** partitioning, respectively. For example, **Q2H** represents the evaluation of query 2 using the horizontal data partitioning.

6.3 Evaluation Results

The federation system was designed for the common case of executing queries over at least partially unknown data sources. This is especially important in the context of source augmentation where data sources unknown to both the federation client and the user may be incorporated automatically into the query plan. In such an environment the ability to generate query plans that can successfully produce early results without the benefit of accurate catalog statistics is crucial. Therefore, except where noted, the results presented in this section are based on evaluating query federation without any pre-existing catalog statistics for the endpoints used.

6.3.1 Federation without Source Augmentation

Query federation without any source augmentation was evaluated to show the effectiveness of the heuristic of "pushing joins down". Results of this evaluation is summarized in Table 6.2.

Query	Completing Subsets	First result	25%	50%	70%	80%	90%	100%
Q1H	3	30ms	50ms	70ms	110ms	110ms	110ms	230ms
Q1V	4	30ms	40ms	50ms	50ms	79.4s	167.2s	167.8s
Q2H	16	30ms	30ms	30ms	30ms	30ms	30ms	30ms
Q2V	8	40ms	40ms	40ms	40ms	40ms	40ms	60ms
Q3H	16	11.2s	11.2s	11.3s	11.3s	11.3s	11.3s	11.3s
Q3V	16	15.3s	15.3s	15.3s	15.3s	15.4s	15.4s	15.4s
Q4H	16	30ms	30ms	30ms	30ms	30ms	30ms	30ms
Q4V	8	30ms	30ms	30ms	30ms	30ms	30ms	40ms

Table 6.2: Evaluation results without source augmentation. The number of input source subsets (out of 31 total) that produce all results is shown along with the mean time to produce early results over all source subsets

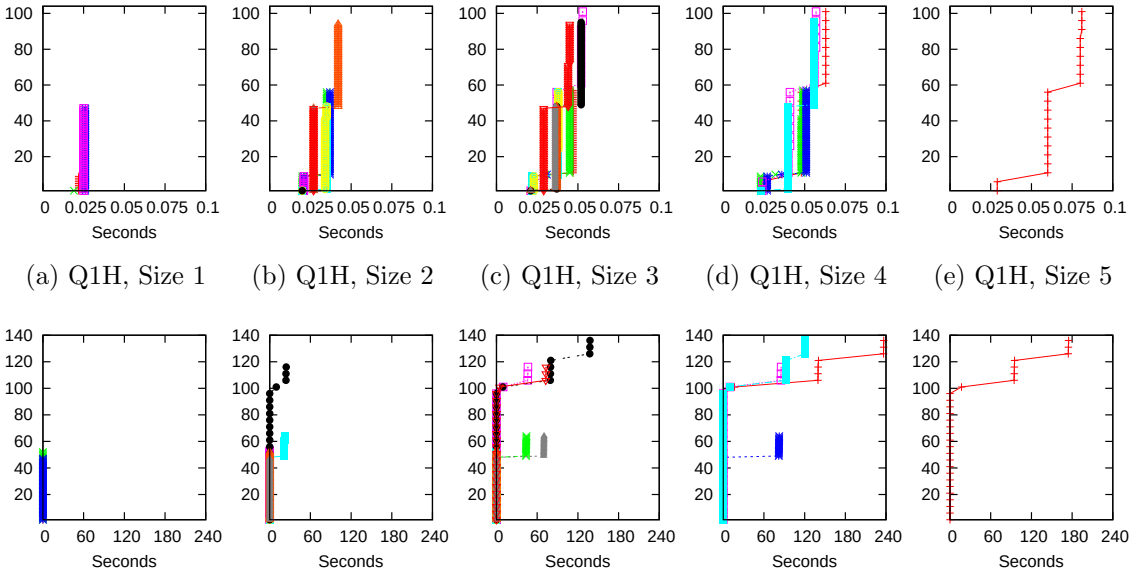
Figures 6.2, 6.3, 6.4, and 6.5 show the results of executing queries **Q1H**, **Q2H**, **Q3H**, and **Q4H**, respectively. The elapsed time from beginning the query execution until each result is produced is plotted with time in seconds on the horizontal axis and total results on the vertical axis.

The evaluation for each query was performed for all possible subsets of the five available endpoints as input. We plot the results of subsets of the same size in the same chart to give an indication of the performance variation caused by using federation with either full or partial knowledge of the available data sources. For example, chart 6.2(b) shows the results of using all 10 2-endpoint subsets of the five available endpoints, while chart 6.2(e) shows the results of using the single 5-element set containing all of the available endpoints.

Q1H has 104 total results over all endpoints. As shown in figure 6.2, evaluation of this query had a mean time to produce the initial result of 30 milliseconds and the final result (in the two subsets that produce all results) of 230 milliseconds. However, not all endpoints produce results to this query. As a result, chart 6.2(a) shows two single-endpoint subsets that produce 47 results each (ACM and CiteSeer),

one that produces 9 results (Semantic Web Dog Food), and one that produces just one result (DBLP RKB). One endpoint (DBLP L3S) is not shown at all as it does not produce any results. The remaining charts in figure 6.2 are similar, showing more results being obtained as larger subsets of available endpoints are used.

Q1V has 139 total results over all endpoints. As a result of using the vertically partitioned dataset, 100 of the results to this query are produced by individual endpoints, while the remaining 39 are produced as a result of cross-endpoint joins. This has an obvious impact on total query time as shown in figure 6.2. Over all subsets, the mean time to produce the initial result is 30 milliseconds while the final result (in the four subsets that produce all results) of 167.8 seconds. The mean time to produce 72% of results (representing the 100 results produced by individual endpoints) is 1.88 seconds. 80% of results were produced in 79.4 seconds, and 90% in 167.2 seconds. All results were produced in 167.8 seconds.



(a) Q1H, Size 1 (b) Q1H, Size 2 (c) Q1H, Size 3 (d) Q1H, Size 4 (e) Q1H, Size 5

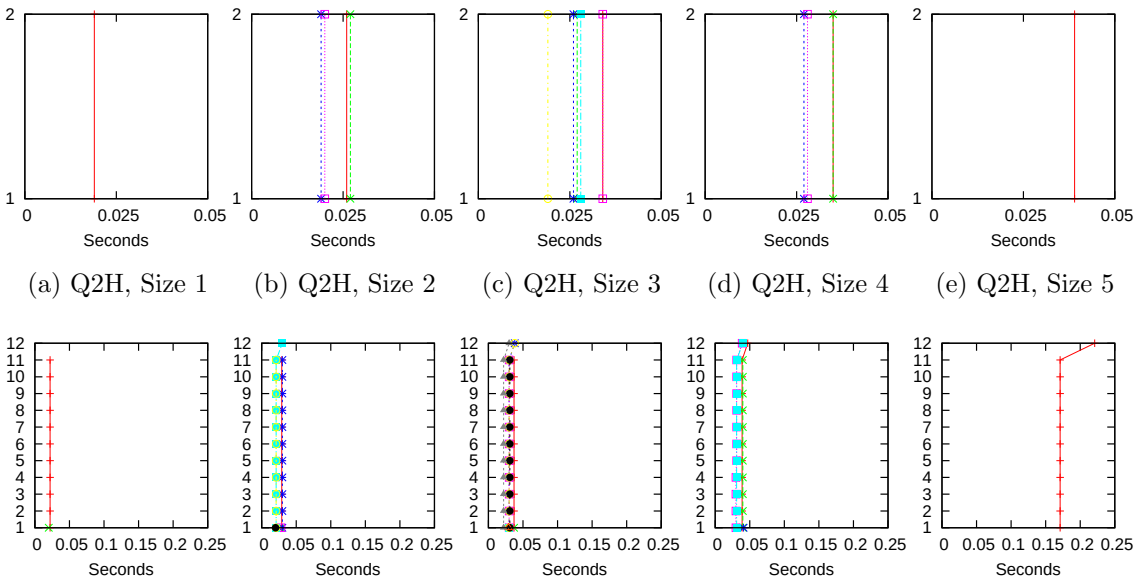
(f) Q1V, Size 1 (g) Q1V, Size 2 (h) Q1V, Size 3 (i) Q1V, Size 4 (j) Q1V, Size 5

Figure 6.2: Query Q1 results produced over time with no augmentation for endpoint subsets of sizes 1–5

As it is based on a dataset-specific IRI, **Q2H** has only 2 total results over all endpoints, both being produced by a single endpoint (CiteSeer). As a result, only subsets in each chart of figure 6.3 that contain the CiteSeer dataset are able to

produce the results. This results in 15 of the 31 non-empty subsets not being able to produce any results for **Q2H**. For those subsets that do produce the results, the mean time to both first and last result is 30 milliseconds.

Q2V has 12 total results over all endpoints. Unlike **Q1V**, the use of vertical partitioning does not introduce any results to **Q2V** produced through cross-endpoint joins, with all results being produced by just two endpoints. As a result, the time to produce all results for both horizontal and vertical partitioning remains much closer than for **Q1**. Over all subsets, the mean time to produce the initial result is 40 milliseconds while the final result (in the eight subsets that produce all results) of 60 milliseconds.



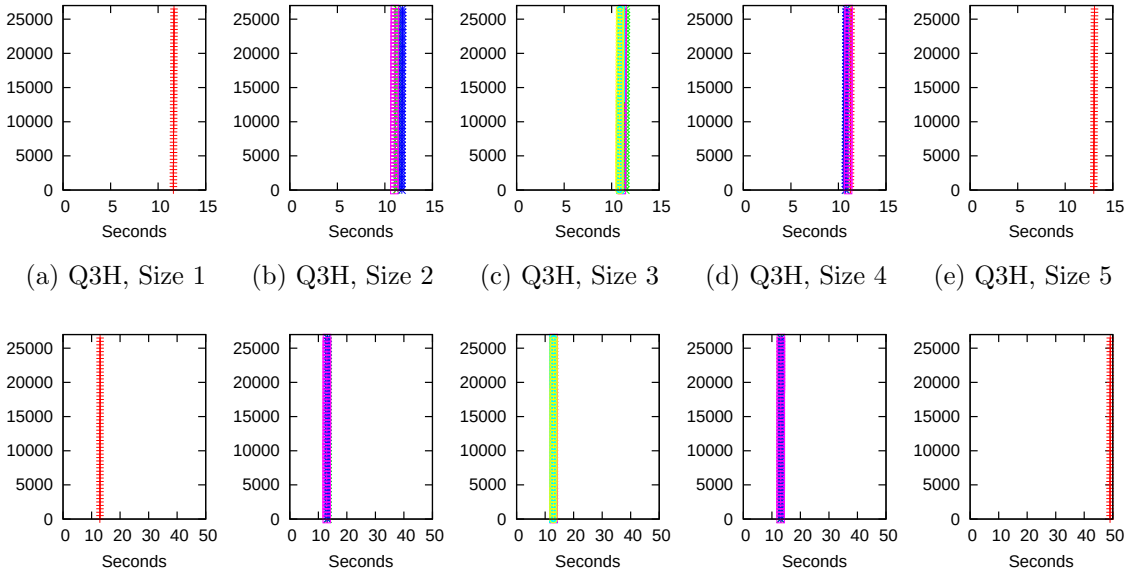
(a) Q2H, Size 1 (b) Q2H, Size 2 (c) Q2H, Size 3 (d) Q2H, Size 4 (e) Q2H, Size 5

(f) Q2V, Size 1 (g) Q2V, Size 2 (h) Q2V, Size 3 (i) Q2V, Size 4 (j) Q2V, Size 5

Figure 6.3: Query Q2 results produced over time with no augmentation for endpoint subsets of sizes 1–5

Queries **Q3H** and **Q3V** have a total of 26,980 results over all endpoints, all being produced by a single endpoint (DBLP L3S). As in the SP^{2}Bench query on which it is based, **Q3** exhibits high selectivity with the triple patterns matching many more records than the join of these patterns. Without accurate catalog statistics with which to order query sub-plans, the high selectivity results in high latency for source subsets including endpoints other than DBLP L3S. Figure 6.4 shows the

results of evaluating queries **Q3**. Across all source subsets, the time to produce all results (from first to last) for horizontal and vertical data partitions occurs within 100 milliseconds after 11.2 and 15.3 seconds of latency, respectively. Both accurate catalog statistics and the use of subquery result caching can dramatically improve the performance in cases such as this (as will be shown in section 6.3.4).



(a) Q3H, Size 1 (b) Q3H, Size 2 (c) Q3H, Size 3 (d) Q3H, Size 4 (e) Q3H, Size 5

(f) Q3V, Size 1 (g) Q3V, Size 2 (h) Q3V, Size 3 (i) Q3V, Size 4 (j) Q3V, Size 5

Figure 6.4: Query Q3 results produced over time with no augmentation for endpoint subsets of sizes 1–5

Similar to **Q2H**, **Q4H** is based on a CiteSeer-specific IRI and has only 2 total results produced by one endpoint. For those subsets that produce both, the mean time to both first and last result is 30 milliseconds. **Q4V** has 12 total results over all endpoints with no cross-endpoint joins, and results being produced by two endpoints. Over all subsets, the mean time to produce the initial result is 30 milliseconds. The mean time to produce the final result (in eight subsets) is 40 milliseconds.

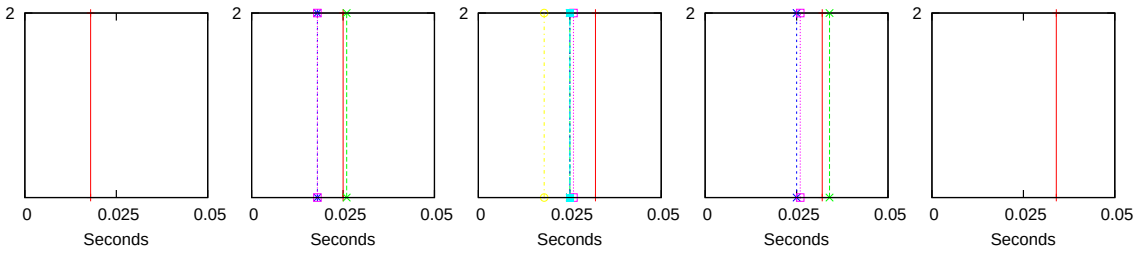
To compare the ability to produce early results with existing approaches to SPARQL query federation, we present results of running queries **Q1**–**Q4** without augmentation with results produced by the FedX[14] system in Figure 6.6. Each system was supplied with knowledge of all five SPARQL endpoints with no catalog statistics. In many cases it can be seen that the initial time used by FedX to probe

each data source to determine applicability of each triple pattern causes FedX to be slower to produce early results than our system. However, the tradeoff for these early results when no (or poor) catalog statistics are available is the execution of sub-optimal query plans during later stages of query execution. This may be seen in the relatively poor performance in producing the final quarter of results in Figure 6.6(e).

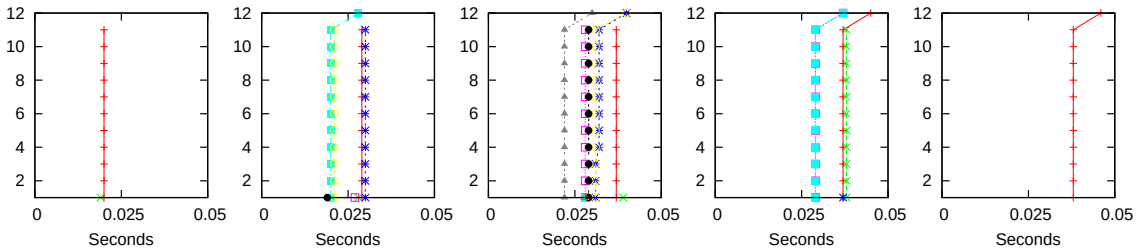
Interestingly, our federation system shows dramatically better performance on **Q3** compared with FedX. With no augmentation, and supplying all five endpoints as input sources, all results are produced for **Q3H** (**Q3V**) in 13.07 (49.1) seconds. However, for **Q3H** (**Q3V**) FedX for produces 3,133 (3,136) results in 276.3 (654.4) seconds before experiencing network timeout errors. This huge difference is explained by FedX's use of a "bind join" which emulates a semi-join by batching intermediate results and sending them in successive requests with query variables bound to the intermediate result values.

Given the huge number of intermediate results produced during evaluation of **Q3**, the use of a "bind join" caused approximately 28,000 batches of intermediate results to be sent to a single endpoint before query execution failed. While the "bind join" in FedX seems to benefit performance in evaluating some queries, in this case our use of more traditional join operators has clear advantages. We suggest that more work is needed to determine when the use the "bind join" is appropriate in federated query evaluation.

Finally, to show the effect of accurate catalog statistics on query planning, figure 6.7 shows results of evaluating queries **Q1V**–**Q4V**, comparing the time to produce results with and without accurate catalog statistics available to the query planner. For all queries the presence of accurate statistics allows more efficient production of query results. This occurs for two reasons. First, the query planner is able to execute sub-plans with higher expected value before plans with lower value. Second, sub-plans that the query planner knows will not contribute to producing query results can be pruned from the plan entirely.

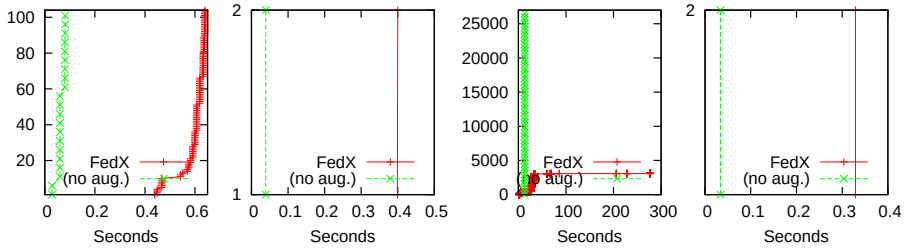


(a) Q4H, Size 1 (b) Q4H, Size 2 (c) Q4H, Size 3 (d) Q4H, Size 4 (e) Q4H, Size 5

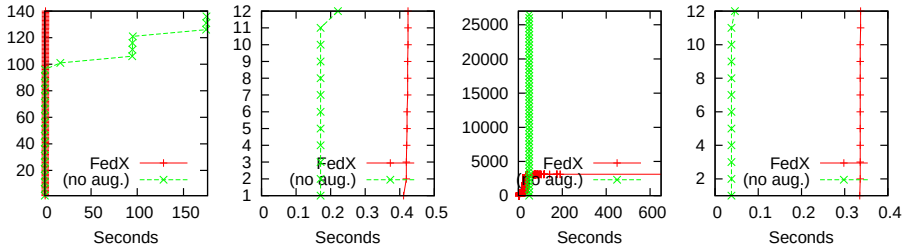


(f) Q4V, Size 1 (g) Q4V, Size 2 (h) Q4V, Size 3 (i) Q4V, Size 4 (j) Q4V, Size 5

Figure 6.5: Query Q4 results produced over time with no augmentation for endpoint subsets of sizes 1–5



(a) Q1H (b) Q2H (c) Q3H (d) Q4H



(e) Q1V (f) Q2V (g) Q3V (h) Q4V

Figure 6.6: Queries Q1–Q4 results produced over time with no augmentation compared with FedX

6.3.2 Federation with SPARQL Augmentation

Table 6.3 summarizes the results of performing the evaluation using SPARQL augmentation. Due to the use of link relations between SPARQL endpoints, every input source subset is able to produce all results available through the related-endpoint topology.

Query	Completing Subsets	First result	25%	50%	70%	80%	90%	100%
Q1H	30	20ms	2.7s	11.6s	42.0s	42.0s	42.0s	80.2s
Q1V	30	30ms	2.7s	45.5s	45.5s	157.3s	268.8s	324.2s
Q2H	30	3.4s	3.4s	3.4s	3.4s	3.4s	3.4s	3.4s
Q2V	30	500ms	4.0s	4.0s	4.0s	4.0s	4.0s	6.5s
Q3H	31	16.0s	16.0s	16.0s	16.0s	16.0s	16.0s	16.0s
Q3V	31	12.6s	12.6s	12.7s	12.7s	12.7s	12.7s	12.7s
Q4H	30	50ms	50ms	50ms	50ms	50ms	50ms	50ms
Q4V	30	30ms	40ms	40ms	40ms	40ms	40ms	50ms

Table 6.3: Evaluation results with SPARQL augmentation. The number of input source subsets (out of 31 total) that produce all results is shown along with the mean time to produce early results over all source subsets

Figure 6.8 shows evaluation results of query **Q1** using SPARQL augmentation. As in the case of **Q1H** without source augmentation, the mean time to produce the initial result is 20 milliseconds. Since four of the five data sources form a strongly connected component of the SPARQL endpoint link topology, all but a single case (the subset containing just the DBLP L3S source) produce all 104 query results. The mean time to produce 50% of results is 11.6 seconds. 55% of results are produced in 15.8 seconds, with all remaining results being produced in 80.2 seconds on average.

As with horizontal partitioning, all but a single subset are able to produce all results for query **Q1V**. The initial results are produced in mean time of 30 milliseconds. The mean time to produce 50% of results is 45.5 seconds. 75% of results are produced in 123.7 seconds, with 100% of results being produced in 324.2 seconds.

Figure 6.9 shows results of evaluating query **Q2** using SPARQL augmentation. The mean time to produce both initial and final results for query **Q2H** is 3.4 seconds. In the case of vertically partitioned data used by **Q2V**, the mean time to initial

result is 500 milliseconds, while the final result is produced in 6.5 seconds. The time to produce results in this case depends heavily on the specific data source subset used. For example, the 5-endpoint (complete) set produces all results in 170 milliseconds, while the mean time to produce all results across subsets of size 3 is 7.9 seconds. Accurate cost model statistics and improvements to the scheduling of query sub-plans may be able to improve the production of early results in cases such as this.

Figure 6.10 shows the results of evaluating query **Q3** using SPARQL augmentation. Due to source augmentation, all initial source subsets are able to produce the entire solution set. As in the horizontal partitioning case, the results here show high latency as a result of high selectivity. The mean time to produce the all results (from first to last) for **Q3H** is 16 seconds. **Q3V** exhibits similar performance characteristics, with a mean time of 12.6 seconds to produce the first result and 12.7 seconds to produce the final result.

Figure 6.11 shows results of query **Q4** using SPARQL augmentation. The mean time to produce all results for **Q4H** is 50 milliseconds. Source augmentation allows query **Q4V** to produce extra results based on discovered sources at the cost of higher latency for the full result set. The mean time to produce the first result is 30 milliseconds, while the final result is produced in 50 milliseconds.

6.3.3 Federation with Linked Data and RDF Content Augmentation

As mentioned previously, the use of linked data augmentation without any SPARQL endpoints available as input data sources results in a query answering system similar to linked data query systems such as SQUIN[10]. In evaluating linked data and RDF content augmentation we find that our approach is competitive with SQUIN and able to produce results for simple queries and bounded data sizes.

However, in many situations we find the use of linked data augmentation to be problematic with respect to the volume and frequency of HTTP operations it produces. For example, executing **Q3V** with linked data augmentation and access to the five available endpoints (either specified as input data sources or discovered partially through SPARQL augmentation) causes 26,931 URIs to be queued for

loading within the roughly 13 seconds that elapse before any results are produced. Given that the vast majority of these URIs share the same hostname (URIs minted by a single data producer), this causes tremendous load on the linked data server and shows that this approach does not scale (even without employing any sort of request throttling). For comparison, the execution of **Q4V** with linked data augmentation and no input data sources causes just three HTTP lookup operations to be performed for the three URIs present in the query string (`rdf:type`, `foaf:Document`, and `swrc:isbn`).

Table 6.4 shows the number of HTTP lookups queued in the first 15 seconds of query execution for queries **Q1V**–**Q4V**, revealing that two of the four queries (**Q1V** and **Q3V**) exhibit scalability problem on vertically partitioned data.

Query	LOD Lookups
Q1V	64,589
Q2V	14
Q3V	26,931
Q4V	14

Table 6.4: HTTP lookup operations caused by linked data augmentation

For queries **Q2V** and **Q4V** (the two queries that do not suffer from the linked data augmentation scaling problems), we show in figure 6.12 the results of evaluating queries with linked data augmentation and all five endpoints available as initial data sources compared to evaluating the same queries with SQUIN.

The deluge of URIs needing to be dereferenced based on linked data augmentation is easily caused by the ability to match sub-queries that are much less selective than the query as a whole (e.g. the triple pattern `?article swrc:isbn ?value`). This same deluge can occur in linked-data crawling systems such as SQUIN by dereferencing a very large single document (e.g. if the `swrc:isbn` URI was configured to return data on every ISBN known to the author).

6.3.4 Performance Impact of Caching on Query Federation

To show the effect of caching subquery results on performance, we evaluate queries **Q1V**–**Q4V** with three caching settings: without any caching enabled; us-

ing an in-memory cache which allows using cached content for repeated subqueries within a single query execution (the default behavior used in the rest of this chapter); and using a persistent, on-disk cache which allows re-use of subquery results across query executions. Before execution, the server and caches were warmed by executing the query three times. Figure 6.13 shows the results of this evaluation. The times to produce specific subsets of the total query results and the performance improvement due to caching is summarized in Table 6.5.

As the data in Table 6.5 shows, the use of sub-query result caching improves performance on all queries for both early results and total results. As mentioned previously, query **Q3V** shows particularly large performance increases due to caching based on the ability to avoid executing high-latency sub-queries on endpoints that would not result in data yielding query results. With a full cache, the execution time to produce all results to query **Q3V** drops from 12.6 seconds to 300 milliseconds.

6.4 Summary

These results show that our federation system is competative with existing approaches to federation over SPARQL and linked data sources (without source augmentation). In the case of federation of SPARQL endpoints, our system succeeds at producing early results compared to existing state of the art systems such as FedX. The time to perform query planning and produce the first result is measured in tens of milliseconds for queries **Q1**, **Q2**, and **Q4**, several times faster than FedX. In the case of **Q3**, FedX is able to produce early results faster than our system due to the reliance on "bind joins," but as discussed in section 6.3.1 this reliance results in a timeout after only a fraction of the result set is produced.

In the case of federation over linked data resources, our system is able to generalize the approach of systems such as SQUIN and produce the same answer set. Both systems produce the full result set for queries **Q2** and **Q4** in seconds with SQUIN producing results earlier than our system. Our federation system's generalized design—especially our use of asynchronous HTTP lookups in the linked data augmentation process—introduces overhead that SQUIN avoids by nature of being purpose-built for linked data crawling. However, our generalized approach is

Query	First result	25%	50%	75%	100%
Q1V (none)	50ms	60ms	80ms	94.9s	192.2s
Q1V (mem)	40ms (+25%)	60ms (+0%)	80ms (+0%)	47.6s (+99%)	77.4s (+148%)
Q1V (disk)	20ms (+150%)	20ms (+200%)	20ms (+300%)	31.8s (+198%)	61.6s (+211%)
Q2V (none)	40ms	40ms	40ms	40ms	50ms
Q2V (mem)	40ms (+0%)	40ms (+0%)	40ms (+0%)	40ms (+0%)	50ms (+0%)
Q2V (disk)	10ms (+300%)	10ms (+300%)	10ms (+300%)	10ms (+300%)	10ms (+400%)
Q3V (none)	12.6s	12.6s	12.6s	12.6s	12.6s
Q3V (mem)	12.5s (+0%)	12.5s (+0%)	12.5s (+0%)	12.6s (+0%)	12.6s (+0%)
Q3V (disk)	220ms (+5604%)	240ms (+5137%)	260ms (+4742%)	280ms (+4399%)	300ms (+4106%)
Q4V (none)	40ms	40ms	40ms	40ms	50ms
Q4V (mem)	40ms (+0%)	40ms (+0%)	40ms (+0%)	40ms (+0%)	50ms (+0%)
Q4V (disk)	10ms (+300%)	10ms (+300%)	10ms (+300%)	10ms (+300%)	10ms (+400%)

Table 6.5: Performance improvement as a result of caching. The the mean time to produce early results using no cache, memory-caching, and (persistent) disk-caching

able to produce the same result sets while being able to integrate other types of data sources beyond just linked data.

Where our system differentiates itself from existing approaches to federation is in its ability to use data source augmentation. Our evaluation shows that—provided data sources make explicit the links between themselves and other, related endpoints—query evaluation need not begin with knowledge of all, or even any of the relevant data sources. So long as links between data sources are available, data source augmentation and our query re-planning system are able to produce complete result sets while continuing to produce early results.

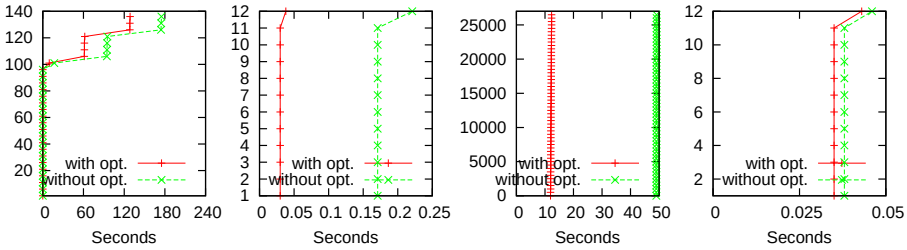
Table 6.3 summarizes the benefits of SPARQL augmentation, showing that in all cases, all or all but one initial source subsets will produce the complete result set. Moreover, the use of SPARQL augmentation continues to allow the production of early results as the time to first result in many cases remains measured in milliseconds, even as the time to produce complete result sets is measured in seconds or minutes. In those cases where one inital source subset is unable to produce all results, this limitation is inherent to the topology of the evaluation datasets and the use the source subset comprised of the DBLP (L3S) source which is prevented from aiding the augmentation process by its lack of knowledge of any related endpoints (as shown in figure 6.1).

Without further research, the source augmentation processes that rely on linked data lookup operations (linked data augmentation and RDF content augmentation) seem to have limited general purpose utility. The potential to generate a flood of HTTP requests for even simple queries poses real performance challenges. As we discuss in the next chapter, several possible approaches might mitigate these challenges.

The evaluation shows that the effects caching can benefit query federation tremendously. This benefit is realized for both early results as well as the complete result set. The effets of caching are especially pronounced for queries such as **Q3** for which sub-plan execution exhibits high-latency as a result of the low selectivity of the query's constiuent patterns. The ability to use cached results for otherwise high-latency, low selectivity sub-plans, is important in a federated system as it

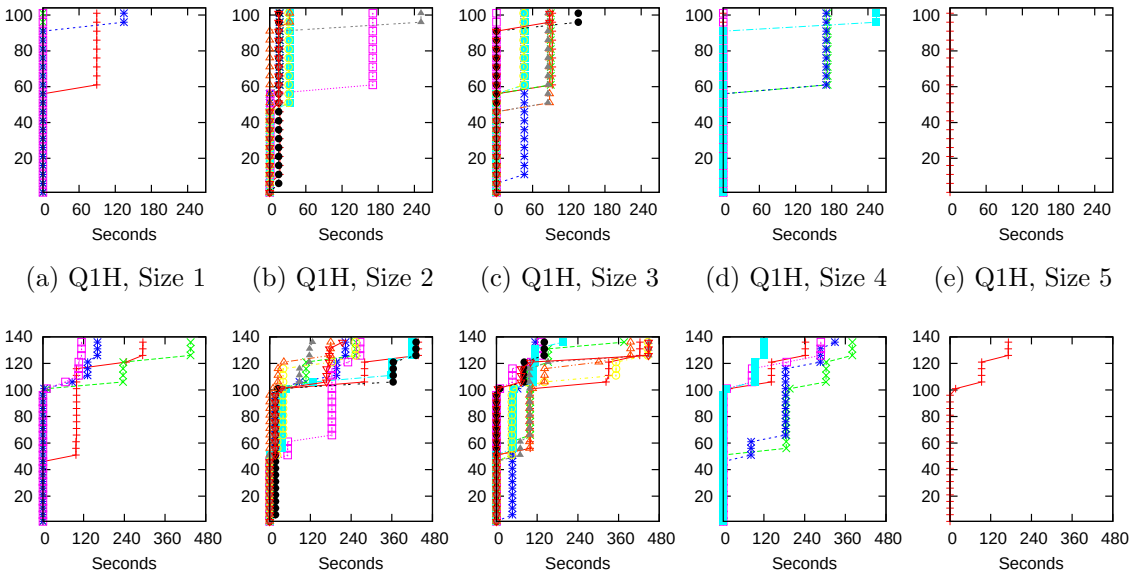
complements query plan pruning (based on accurate catalog statistics) as a method of reducing the unnecessary latency of evaluating cross-source joins that do not contribute to the final result set.

Finally, we note that our evaluation on the impact of sub-query result caching considers queries over a static dataset only. No updates (neither relevant nor irrelevant) are performed and so once cached, query results are always valid. For a more detailed look at the impact of caching on both read-only and read-write use cases, and on the cost of implementing caching support in a SPARQL implementation, we direct readers to our evaluation in [49].



(a) Q1V (b) Q2V (c) Q3V (d) Q4V

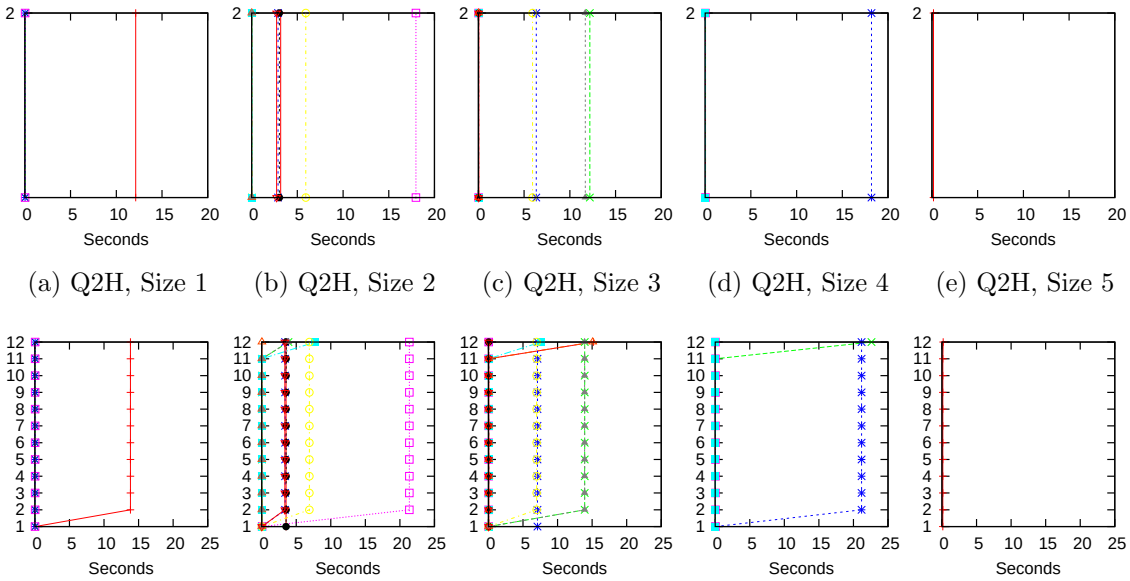
Figure 6.7: Queries Q1V–Q4V results produced over time with and without query plan optimization based on accurate catalog statistics



(a) Q1H, Size 1 (b) Q1H, Size 2 (c) Q1H, Size 3 (d) Q1H, Size 4 (e) Q1H, Size 5

(f) Q1V, Size 1 (g) Q1V, Size 2 (h) Q1V, Size 3 (i) Q1V, Size 4 (j) Q1V, Size 5

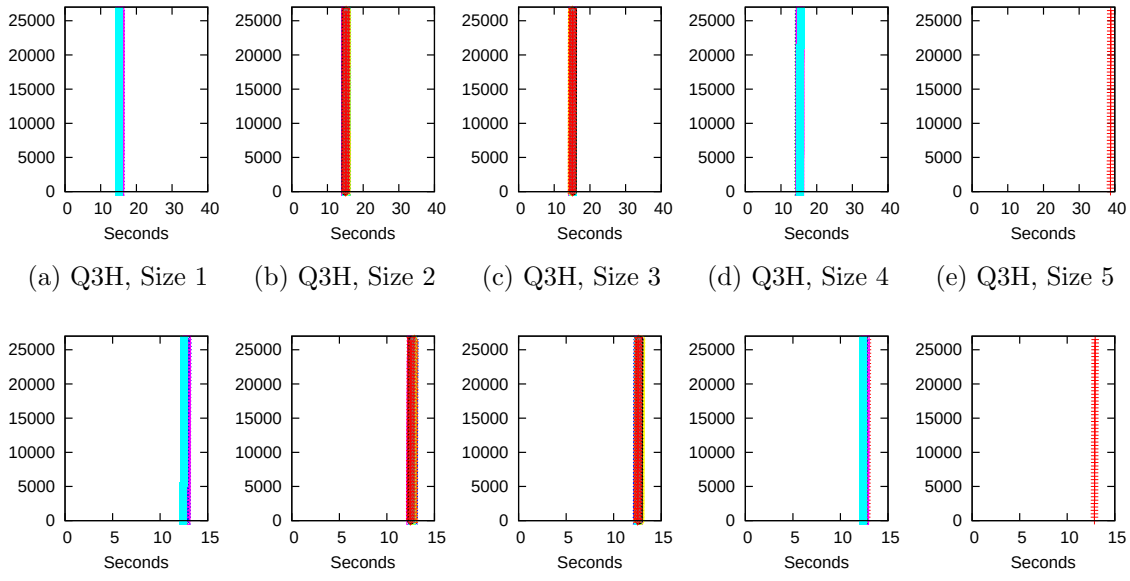
Figure 6.8: Query Q1 results produced over time with SPARQL augmentation for initial endpoint subsets of sizes 1–5



(a) Q2H, Size 1 (b) Q2H, Size 2 (c) Q2H, Size 3 (d) Q2H, Size 4 (e) Q2H, Size 5

(f) Q2V, Size 1 (g) Q2V, Size 2 (h) Q2V, Size 3 (i) Q2V, Size 4 (j) Q2V, Size 5

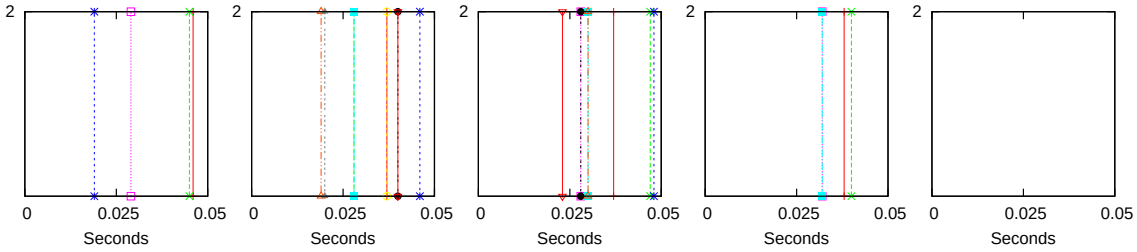
Figure 6.9: Query Q2 results produced over time with SPARQL augmentation for initial endpoint subsets of sizes 1–5



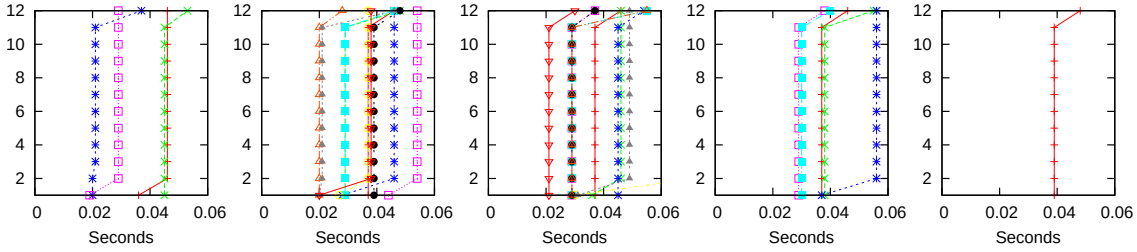
(a) Q3H, Size 1 (b) Q3H, Size 2 (c) Q3H, Size 3 (d) Q3H, Size 4 (e) Q3H, Size 5

(f) Q3V, Size 1 (g) Q3V, Size 2 (h) Q3V, Size 3 (i) Q3V, Size 4 (j) Q3V, Size 5

Figure 6.10: Query Q3 results produced over time with SPARQL augmentation for initial endpoint subsets of sizes 1–5

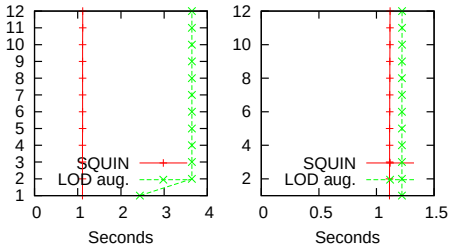


(a) Q4H, Size 1 (b) Q4H, Size 2 (c) Q4H, Size 3 (d) Q4H, Size 4 (e) Q4H, Size 5



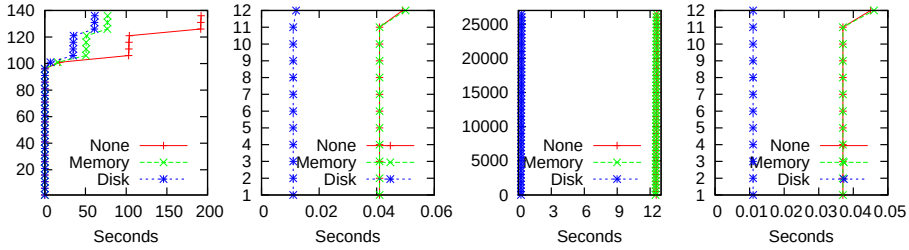
(f) Q4V, Size 1 (g) Q4V, Size 2 (h) Q4V, Size 3 (i) Q4V, Size 4 (j) Q4V, Size 5

Figure 6.11: Query Q4 results produced over time with SPARQL augmentation for initial endpoint subsets of sizes 1–5



(a) (b)

Figure 6.12: Queries Q2V (a) and Q4V (b) results produced over time with linked data augmentation compared with SQUIN



(a) Q1V (b) Q2V (c) Q4V (d) Q4V

Figure 6.13: Queries Q1V–Q4V results produced over time showing the effects of HTTP caching

CHAPTER 7
DISCUSSION AND CONCLUSIONS

We have introduced query planning techniques for SPARQL that allows queries to be evaluated over a set of remote SPARQL endpoints and optimized for early returns. Our evaluation shows promising results for situations in which early results are desired, for queries of variable selectivity and data distribution characteristics.

7.1 Discussion of Results

Our federation framework, and in particular the definition of data source augmentation, allows for the federated execution of queries over an expanding set of data sources which need not be known at the start of execution. The extensible nature of the source augmentation definition allows for a wide variety of approaches to discovering new, relevant data sources for query execution.

The use of SPARQL source augmentation allows query federation to quickly discover relevant data sources to query. Implementing SPARQL source augmentation using link relations included in the query response enables this process in a simple manner and makes explicit the web of connections between existing SPARQL endpoints that overlap in their coverage. Moreover, including link relations in query results allows implementors the flexibility to trade simplicity and performance (using a static list of connected endpoints) with accuracy (computing a list of endpoints relating specifically to the query results being returned).

As can be seen in the evaluation results, the use of SPARQL augmentation allows the federation client to discover relevant, related data sources provided by remote endpoints, and to incorporate those sources into the query plan. The federation framework succeeds in producing early results while being able to incorporate new data sources through query re-planning. In the cases where sufficient metadata is provided by the data sources to discover all relevant endpoints, the federation client is able to produce a complete result set with respect to all available endpoints, regardless of the initial set of sources provided. Moreover, in many cases a

significant fraction of the result set is able to be produced in a much shorter time than the complete result set.

When combined with support for (sub-)query result caching, these features greatly improve the prospects for efficient federated queries on the web. The use of query result caching improves performance dramatically for both early results as well as complete result sets.

7.2 Future Work

In this work we focused on rewriting techniques for basic graph patterns. We hope to investigate more expressive graph patterns and their potential rewritings in the face of distributed query evaluation in the future. We also hope to investigate the effect on performance of union parallelization, query-time join re-ordering or scrambling, and more accurate methods of estimating the cost of evaluating graph patterns on remote endpoints.

While we assume that remote endpoints support only pure SPARQL, extensions to SPARQL can augment the query optimizations we have discussed and provide for more efficient query evaluation strategies. For example, existing work on using semi-joins in SPARQL [56, 57] could reduce communication costs in subplans produced by our optimizer while still providing for early results.

7.2.1 Federation Awareness of Reasoning and Data Ownership

The partitioning of data across data sources remains a big challenge to practical and efficient federated query execution. Both resource identifiers and the vocabularies to describe those resources often vary from source to source, causing problems for users and systems trying to reconcile data from multiple sources. Recent developments in standardizing reasoning support in SPARQL[58] may prove effective at easing the process of SPARQL query federation in the future. However, many SPARQL and linked data sources will lack reasoning support for the foreseeable future. Thus, work is needed to enable reasoning in the tools consuming semantic data when server support is absent.

We believe support for identity reasoning (e.g. the use of `owl:sameAs` map-

pings) may be implemented within the federation framework, aided by the previously mentioned work on semi-joins, as well as the newly standardized `VALUES` feature in SPARQL. More research is needed to determine if such an approach could be made efficient, and if similar run-time approaches might be used to address schema mapping across data sources.

A better understanding of which URIs are likely to be served by which endpoints (e.g. through pattern matching), and how those URIs are related with `owl:sameAs` mappings, would also aid in the use of linked data augmentation. Such an understanding would allow the federation client to use SPARQL endpoints when appropriate and to resort to linked data lookup operations when necessary (helping to avoid the deluge of URL lookup operations as discussed in section 6.3.3).

A better understanding of query structures might also allow a system to determine before execution that linked data augmentation is appropriate. More and better metadata about data sources and their ownership of certain data might also allow a system to recognize certain linked data sources as mirroring the data available at an existing endpoint and to employ linked data augmentation only in situations where that endpoint was unavailable.

7.2.2 Parallelization of query operators

The current implementation of query operators in the framework is entirely serial (although the linked data augmentation that is initiated by query evaluation runs independently). Parallel execution of query operators is a well studied area with a large literature and long history. Without turning this thesis into one on parallel databases, there are some obvious opportunities for parallelizing query operators with very little cost in the way of system resources.

One such case that is the parallelization of the union operator in such a way as to immediately and concurrently evaluate all union branches that require only constant resource utilization (subplans executed entirely at a remote data source). Remaining union branches may be evaluated serially (as they are now), or with a limited amount of parallelism subject to system resource availability. The integration of techniques such as query scrambling [27] meant to minimize the effect of

poorly performing sequential operations is left to future work.

7.2.3 User Preferences in Cost Model

Beyond objective data that may be available to the cost model and query planner such as dataset statistics and observed latency, *subjective* data may play a role in producing optimal query plans. For example, a user may have preferences regarding the quality and trustworthiness of data sources. Such preferences may imply that the query planner ought to make different choices in producing a query plan than would be made utilizing only the objective cost model data.

User-expressed preferences on the overall value of data sources may be especially important in situations where many sources are involved in the same round of query rewriting. In such a situation, the resulting query plan will contain a large number of (very likely sequential) union branches. Prioritizing high-value sources (based in part on user preferences) during query rewriting is crucial to avoid unnecessary delays and low-value results.

7.2.4 Providing Global Context to Local Evaluation

There are many possibilities for improving the query performance and metadata quality when sending subqueries to a SPARQL endpoint. One such possibility is in providing context for the subquery, allowing the remote endpoint a more full understanding of *why* the subquery is being executed. For example, if a subquery requesting authors and topics for publications is sent to an endpoint, knowledge that the original query is trying to obtain authors' phone numbers might allow the endpoint to provide metadata about related endpoints that are known to contain such information instead of simply a list of *all* related endpoints. Future work may attempt to determining how such contextual information might be provided to endpoints and how an endpoint might leverage such information and indicate its use in the returned results.

7.3 Summary

In this work we have shown that it is possible to leverage both a priori knowledge of and newly discovered data sources in query answering on the web. We described a framework for federated query planning including an initial query planner (section 3.3) to produce a query plan targeting *early results*, a query re-planner (section 3.4) that is able to expand existing query plans to incorporate newly available data sources mid-execution, and a source augmentation mechanism with three augmentation function definitions (chapter 4) that are able to discover new data sources during query execution. In chapter 5 we described a method to enable HTTP-level caching of query results by maintaining data modification timestamps in the search indexes used by SPARQL endpoints.

The evaluation in chapter 6 demonstrates that the initial query planning can successfully produce early results from known sources and is competitive with existing SPARQL federation systems. The evaluation also shows that source augmentation and query re-planning can leverage newly discovered sources to iteratively enlarge the result set. Finally, it shows that SPARQL result caching can dramatically increase overal performance of query answering. The evaluation over real-world bibliographic data and data source toplogies demonstrates that these components can not only work individually, but can be combined into a powerful federated query answering system.

REFERENCES

[1] G. Klyne and J. J. Carroll, "Resource Description Framework (RDF): Concepts and Abstract Syntax," W3C, W3C Recommendation, Feb. 2004, retrieved November 8, 2012, from http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. 1

[2] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "SPARQL 1.1 Protocol," W3C, W3C Proposed Recommendation, Nov. 2012, retrieved January 29, 2013, from http://www.w3.org/TR/2013/PR-sparql11-protocol-20130129/. 1, 18

[3] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," W3C, W3C Proposed Recommendation, Nov. 2012, retrieved November 8, 2012, from http://www.w3.org/TR/2012/PR-sparql11-query-20121108/. 1, 16, 17, 40

[4] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surveys*, vol. 32, no. 4, pp. 422–469, Dec. 2000. [Online]. Available: http://doi.acm.org/10.1145/371578.371598 7, 9

[5] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, no. 3, pp. 183–236, Sep. 1990. [Online]. Available: http://doi.acm.org/10.1145/96602.96604 7

[6] L. Haas, M. T. Roth, and F. Ozcan, "Cost models do matter: Providing cost information for diverse data sources in a federated system," in *Proc. 25th Int. Conf. Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 599–610. 7, 9

[7] B. Quilitz and U. Leser, "Querying distributed rdf data sources with sparql," in *Proc. 5th European Semantic Web Conf.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 524–538. [Online]. Available: http://dl.acm.org/citation.cfm?id=1789394.1789443 7

[8] A. Langegger, W. Wöß, and M. Blöchl, "A semantic web middleware for virtual data integration on the web," in *Proc. 5th European Semantic Web Conf.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 493–507. [Online]. Available: http://dl.acm.org/citation.cfm?id=1789394.1789441 8

[9] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets, "Tabulator: Exploring and analyzing

linked data on the semantic web," in *Proc. 3rd Int. Semantic Web User Interaction Workshop*, vol. 2006, 2006. 8

[10] O. Hartig, C. Bizer, and J.-C. Freytag, "Executing sparql queries over the web of linked data," in *Proc. 8th Int. Semantic Web Conf.* Berlin, Heidelberg: Springer-Verlag, 2009, pp. 293–309. [Online]. Available: http://data.semanticweb.org/papers/iswc/2009/paper301.pdf 8, 11, 33, 61

[11] "Linked Data – Design Issues," retrieved February 1, 2013, from http://www.w3.org/DesignIssues/LinkedData.html. 8

[12] O. Görlitz and S. Staab, "SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions," *Proc. 2nd Int. Workshop on Consuming Linked Data*, Jan 2011. [Online]. Available: https://files.ifi.uzh.ch/ddis/iswc_archive/iswc/pps/web/iswc2011.semanticweb.org/fileadmin/iswc/Papers/Workshops/COLD/GoerlitzAndStaab_COLD2011.pdf 8, 22

[13] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, "Describing Linked Datasets with the VoID Vocabulary," W3C, W3C Interest Group Note, Mar. 2011, retrieved December 15, 2012, from http://www.w3.org/TR/2011/NOTE-void-20110303/. 8, 21, 35

[14] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "Fedx: Optimization techniques for federated query processing on linked data," in *Proc. 10th Int. Semantic Web Conf.*, 2011. [Online]. Available: http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/05/70310592.pdf 9, 22, 57

[15] A. Levy, "Answering queries using views: A survey," *The Int. Journal Very Large Data Bases*, vol. 10, no. 4, pp. 270–294, Dec. 2001. [Online]. Available: http://dx.doi.org/10.1007/s007780100054 9

[16] C.-N. Hsu and C. A. Knoblock, "Semantic query optimization for query plans of heterogeneous multidatabase systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 6, pp. 959–978, Nov. 2000. [Online]. Available: http://dx.doi.org/10.1109/69.895804 9

[17] S. Adali, K. Candan, Y. Papakonstantinou, and V. Subrahmanian, "Query caching and optimization in distributed mediator systems," in *Proc. ACM Sigmod Conf.*, 1996. 9

[18] "Semantic Web," retrieved February 1, 2013, from http://www.w3.org/standards/semanticweb/. 9

[19] M. Rodriguez-Martinez and N. Roussopoulos, "Mocha: A self-extensible database middleware system for distributed data sources," in *Proc. 2000 ACM SIGMOD international conference on Management of data*, 2000. 9

[20] D. Kossmann and K. Stocker, "Iterative dynamic programming: a new class of query optimization algorithms," *ACM Trans. Database Syst.*, vol. 25, no. 1, pp. 43–82, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/352958.352982 9, 81

[21] M. J. Carey and D. Kossman, "On saying "Enough already!" in SQL," in *Sigmod Record*, vol. 26, 1997. 10

[22] C. Mishra and N. Koudas, "Interactive query refinement," in *Proc. 12th Int. Conf. Extending Database Technology*. New York, NY, USA: ACM, 2009, pp. 862–873. [Online]. Available: http://doi.acm.org/10.1145/1516360.1516459 10

[23] ——, "Stretch 'n' shrink: resizing queries to user preferences," in *Proc. 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1227–1230. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376741 10

[24] C. Bobineau, C. Collet, and T.-T. VU, "A strategy to develop adaptive and interactive query brokers," in *Proc. Int. Conf. Manage. Data*, 2008, pp. 237–247. 10

[25] A. Miele, E. Quintarelli, and L. Tanca, "A methodology for preference-based personalization of contextual data," in *Proc. 12th Int. Conf. Extending Database Technology*. New York, NY, USA: ACM, 2009, pp. 287–298. [Online]. Available: http://doi.acm.org/10.1145/1516360.1516394 10

[26] S. Magliacane, A. Bozzon, and E. D. Valle, "Efficient execution of top-k sparql queries," in *Proc. 11th Int. Semantic Web Conf.*, 2012. [Online]. Available: http://iswc2012.semanticweb.org/sites/default/files/76490337.pdf 10

[27] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan, "Scrambling query plans to cope with unexpected delays," in *Proc. PDIS Conf.*, 1996. 10, 26, 72

[28] R. Avnur and J. Hellerstein, "Eddies: Continuously adaptive query processing," *ACM SIGMoD Record*, Jan 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=335420 10

[29] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "Anapsid: An adaptive query processing engine for sparql endpoints," in *Proc. 10th Int. Semantic Web Conf.*, 2011. [Online]. Available: http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/03/70310017.pdf 11

[30] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," *Proc. 8th Annu. Joint Conf. IEEE Comput. Commun. Societies*, vol. 1, pp. 126 –134 vol.1, mar 1999. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=749260 11

[31] M. Gallego, J. Fernández, M. Martínez-Prieto, and P. Fuente, "An empirical study of real-world SPARQL queries," *USEWOD2011 - 1st Int. Workshop on Usage Analysis and the Web of Data*, 2011. [Online]. Available: http://ir.ii.uam.es/usewod2011/usewod2011_arias.pdf 11, 37

[32] A. Harth and S. Decker, "Optimized index structures for querying RDF from the web," *Proc. 3rd Latin American Web Congress*, 2005. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/LAWEB.2005.25 11

[33] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, Aug. 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1453965 11

[34] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for rdf," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 647–659, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453927 11

[35] S. Harris, N. Lamb, and N. Shadbolt, "4store: The design and implementation of a clustered rdf store," in *Proc. 5th Int. Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009. [Online]. Available: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-517/ssws09-paper7.pdf 12

[36] J. Goldstein and P. Larson, "Optimizing queries using materialized views: a practical, scalable solution," *Proc. 2001 ACM SIGMOD Int. Conf. Management of Data*, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=375706 12

[37] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "Dbproxy: A dynamic data cache for web applications," *Proc. 19th Int. Conf. Data Engineering*, 2003. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/ICDE.2003.1260881 12

[38] P. Larson, J. Goldstein, and J. Zhou, "MTCache: Transparent Mid-tier Database Caching in SQL server," in *Proc. 20th Int. Conf. Data Engineering*, 2004, pp. 177 – 188. 12

[39] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with SPARQL query caching," *Proc. 7th Extended Semantic Web Conf.*, 2010. [Online]. Available: http://www.springerlink.com/index/764M684325739V67.pdf 12, 13, 40, 42, 43, 45

[40] O. Hartig, "How caching improves efficiency and result completeness for querying linked data," *Proc. 4th Linked Data on the Web (LDOW) Workshop*,

Mar 2011. [Online]. Available: http://wtlab.um.ac.ir/parameters/wtlab/filemanager/LD_resources/LDOW2011/ldow2011-paper05.pdf 13

[41] S. Das, S. Sundara, and R. Cyganiak, "R2RML: RDB to RDF Mapping Language," W3C, W3C Recommendation, Sep. 2012, retrieved December 15, 2012, from http://www.w3.org/TR/2012/REC-r2rml-20120927/. 18

[42] M. Stocker, A. Seaborne, A. Bernstein, and C. Kiefer, "SPARQL basic graph pattern optimization using selectivity estimation," in *Proc. 17th Int. World Wide Web Conf.*, 2008. 19

[43] M. J. Franklin, B. T. Jónsson, and D. Kossmann, "Performance tradeoffs for client-server query processing," in *Proc. Int. Conf. Management Data*. New York, NY, USA: ACM, 1996, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/233269.233328 21

[44] G. T. Williams, "SPARQL 1.1 Service Description," W3C, W3C Proposed Recommendation, Nov. 2012, retrieved November 8, 2012, from http://www.w3.org/TR/2012/PR-sparql11-service-description-20121108/. 31

[45] M. Nottingham, "Web Linking," *Internet RFC 5988, ISSN 2070-1721*, Oct. 2010, retrieved December 15, 2012, from http://tools.ietf.org/html/rfc5988. 31, 52

[46] K. Kjernsmo, "The necessity of hypermedia rdf and an approach to achieve it," *Proc. First Linked APIs Workshop*, 2012. 31, 35

[47] "Linked Data," retrieved December 15, 2012, from http://linkeddata.org/. 36, 51

[48] Roy T. Fielding and James Gettys and Jeffrey C. Mogul and Henrik Frystyk Nielsen and Larry Masinter and Paul J. Leach and Tim Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," *Internet RFC 2616, ISSN 2070-1721*, Jun. 1999, retrieved December 15, 2012, from http://tools.ietf.org/html/rfc2616. 38

[49] G. T. Williams and J. Weaver, "Enabling fine-grained HTTP caching of SPARQL query results," in *Proc. 10th Int. Semantic Web Conf.* Springer-Verlag, Oct. 2011. [Online]. Available: http://www.springerlink.com/index/MQ0X5238661U4526.pdf 50, 66

[50] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP^{2}Bench: A SPARQL Performance Benchmark," in *Proc. IEEE Int. Conf. Data Engineering*, 2009. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4812405 51

[51] H. Glaser, "Association for Computing Machinery (ACM) (RKBExplorer)," Jan. 2013, retrieved January 10, 2013, from http://datahub.io/dataset/rkb-explorer-acm. 51

[52] ——, "CiteSeer (Research Index) (RKBExplorer)," Jan. 2013, retrieved January 10, 2013, from http://datahub.io/dataset/rkb-explorer-citeseer. 51

[53] ——, "DBLP Computer Science Bibliography (RKBExplorer)," Jan. 2013, retrieved January 10, 2013, from http://datahub.io/dataset/rkb-explorer-dblp. 52

[54] J. Diederich, "DBLP in RDF (L3S)," Jan. 2013, retrieved January 10, 2013, from http://datahub.io/dataset/l3s-dblp. 52

[55] K. Möller, "Semantic Web Dog Food Corpus," Jan. 2013, retrieved January 10, 2013, from http://datahub.io/dataset/semantic-web-dog-food. 52

[56] J. Zemánek, S. Schenk, and V. Svátek, "Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-joins," *ISWC 2008 Poster and Demo Session Proc.*, Dec 2008. [Online]. Available: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-401/iswc2008pd_submission_69.pdf 71

[57] G. T. Williams, "Supporting Identity Reasoning in SPARQL Using Bloom Filters," in *Proc. of Workshop on Advancing Reasoning on the Web*, F. van Harmelen, A. Herzig, P. Hitzler, Z. Lin, R. Piskac, and G. Qi, Eds. CEUR-WS, ISSN 1613-0073, 2008. [Online]. Available: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-350/paper8.pdf 71

[58] B. Glimm and C. Ogbuji, "SPARQL 1.1 Entailment Regimes," W3C, W3C Proposed Recommendation, Nov. 2012, retrieved January 29, 2013, from http://www.w3.org/TR/2013/PR-sparql11-entailment-20130129/. 71

APPENDIX A
Static Query Planning Algorithms

Algorithm 3: Classic Dynamic Programming Algorithm *StaticPlanner*

Input: BGP query q with triple patterns P_1,...,P_n, a set of remote endpoints E
Output: A query plan for q

```
1  for i = 1 to n do
2  |  optPlan({P_i}) = accessPlans(P_i, E) ;
3  |  prunePlans(optPlan({P_i})) ;
4  end
5  for i = 2 to n do
6  |  forall the S ⊆ {P_1, ..., P_n} such that |S| = i do
7  |  |  optPlan(S)=∅ ;
8  |  |  forall the O ⊂ S do
9  |  |  |  optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O),
   |  |  |  optPlans(S-O)) ;
10 |  |  |  prunePlans(optPlan(S)) ;
11 |  |  end
12 |  end
13 end
14 finalizePlans(optPlan({P_1, ..., P_n})) ;
15 prunePlans(optPlan({P_1, ..., P_n})) ;
16 return optPlan({P_1, ..., P_n}) ;
```

Algorithm 4: Distributed accessPlans implementation

Input: Triple pattern P, a set of remote endpoints $E = \{E_1 \ldots, E_m\}$
Output: A query plan for P

```
1  remoteCall(P) = sortedUnionPlan({P^{E_i} | 1 ≤ i ≤ m}) ;
2  return remoteCall(P) ;
```

Algorithm 3 shows the classic dynamic programming algorithm. No changes have been made to this algorithm, and so it could easily be replaced with the classic greedy algorithm or the iterative dynamic programming algorithm [20].

Algorithm 4 shows the *accessPlans* function. The traditional implementation based on providing access plans via indexes and/or table scans has been replaced

Algorithm 5: Rewriting *joinPlans* implementation

Input: Query plans P, Q, a set of remote endpoints $E_1 \ldots, E_m$
Output: A query plan for $P \bowtie Q$

```
1  joins(P,Q) = ⋃_{i∈JoinAlgorithms} P⋈_i Q ;
2  foreach R ∈joins(P,Q) do
4      foreach plan node S∈R in postfix order do
5          if S = P^{E_i} then
6              Source(S) ← E_i ;
7              remoteCalls(S) ← 1 ;
8          else if S = U⋈_j V and Source(U)=Source(V)=E_i then
10             S = U⋈^{E_i} V ;
11             remoteCalls(S) ← 1 ;
12         else if S = U⋈_j V then
13             remoteCalls(S) ← remoteCalls(U) + remoteCalls(V) ;
14         else if S = Union(U_1, ..., U_m) then
15             remoteCalls(S) ← Σ_{i=1}^{m} remoteCalls(U_i) ;
16         end
17         R ← pushdownJoins(R) ;
18     end
19 end
20 return joins(P,Q) ;
```

with a version that matches a triple pattern by taking the union over executing the triple pattern at all known remote endpoints. A potential improvement (not shown, but discussed in section **??**) would be to ignore endpoints that will not return any answers for the triple pattern. This might be known at planning time based on dataset descriptions provided by the endpoint or by online statistics kept about the endpoint.

Algorithm 5 augments the classic *joinPlans* function with several rewriting rules. The first rule (algorithm 5 line 10) merges sibling QEP tree nodes into a single remote operation in cases where the siblings are marked for execution on the same remote endpoint (by lifting the Source($\cdot$) value from the children nodes to the parent node). The second rule (algorithm 6) swaps the ordering of union and join operators such that unions are pushed towards the root of the QEP tree and joins are pushed down towards the leaves. The third rule (algorithm 7) flattens nested union operators (union(union(A,B),C) $\equiv$ union(A,B,C)). Finally, the fourth rule

Algorithm 6: Query Plan Rewriting Algorithm *pushdownJoins*

Input: Query plan p
Output: A new query plan with joins pushed below unions

```
1  if Type(p) = Union then
2  |  return p ;
3  end
4  (lhs, rhs) ← Children(p) ;
5  if Type(lhs) = Union and Type(rhs) = Union then
6  |  plans ← {copy(lplan)⋈copy(rplan) |
   |  lplan∈Children(lhs), rplan∈Children(rhs)} ;
7  |  return sortedUnionPlan(plans) ;
8  else if Type(lhs) = Union then
9  |  plans ← ⋃_{uplan∈Children(lhs)} uplan⋈copy(rhs) ;
10 |  return sortedUnionPlan(plans) ;
11 else if Type(rhs) = Union then
12 |  plans ← ⋃_{uplan∈Children(rhs)} copy(lhs)⋈uplan ;
13 |  return sortedUnionPlan(plans)
14 else
15 |  return p ;
16 end
```

(algorithm 7 line 9) re-orders unions so that sub-plans with fewer remote operations are "preferred" (that is, appear to the left of sub-plans with more remote operations). Among the subplans with the same number of calls, we prefer the nodes with higher utility: preferred sources and higher number of expected results.

Algorithm 7: Union Query Plan Constructor *sortedUnionPlan*

Input: A set of union subplans *plans*
Output: A new union plan with subplans ordered by remote invocation count

```
1  uplans = ∅ foreach p∈plans do
2  |  if Type(p) = Union then
3  |  |  uplans ← uplans ∪ Children(p) ;
4  |  else
5  |  |  uplans ← uplans ∪ {p} ;
6  |  end
7  end
9  subplans ← sort plans q∈uplans s.t. remoteCalls(q) is in ascending
   remoteCalls(q) values (and descending expected utility values) ;
10 if |subplans| = 1 then
11 |  return q∈subplans ;
12 else
13 |  return Union(subplans) ;
14 end
```

APPENDIX B
DYNAMIC QUERY PLANNING AND AUGMENTATION ALGORITHMS

Algorithm 8: Query Plan Expansion Algorithm *expandQueryPlan*

Input: Query plan p and a set of new data sources S
Output: A new query plan adding S to p

```
1  planType ← Type(p) ;
2  if planType = Triple then
3      sourceLabel ← Source(p) ;
4      if sourceLabel = ∅ then
5          Q ← ∅ ;
6          foreach s ∈ S do
7              q ← copy(p) ;
8              Source(q) ← s ;
9              Q ← Q ∪ {q} ;
10         end
11         return sortedUnionPlan(Q ∪ {p}) ;
12     else
13         return copy(p) ;
14     end
15 else if planType = Union then
16     subplans ← ⋃_{q∈Children(p)} expandQueryPlan(q, S) ;
17     activeSubplans ← {q∈subplans|(S∪{∅})∩ChildSources(q) ≠ ∅} ;
18     return sortedUnionPlan(activeSubplans) ;
19 else
20     subplans ← ⋃_{q∈Children(p)} expandQueryPlan(q, S) ;
21     newPlan ← new planType(subplans) ;
22     return pushdownJoins(newPlan) ;
23 end
```

Algorithm 9: Linked Data Augmentation *inspectResult*

```
   Input: Result r
1  foreach (var → term) ∈ r do
2  |  if term is an IRI and term has not been dereferenced already then
3  |  |  response ← dereference(term) ;
4  |  |  if response.isSuccess() and response.isRDFContentType then
5  |  |  |  parseRDFIntoTriplestore(response.content) ;
6  |  |  end
7  |  end
8  end
```

APPENDIX C
EVALUATION QUERIES

Query Q1

This query selects all publications and their titles for a specific author's name.

```
PREFIX akt: <http://www.aktors.org/ontology/portal#>
SELECT ?pub ?title WHERE {
  ?person akt:full-name "James Hendler" .
  ?pub akt:has-author ?person .
  ?pub akt:has-title ?title .
}
```

Query Q2

This query selects all publications and their titles for a specific author's IRI. Newlines have been inserted for typesetting purposes.

The exact IRI used for the person changes depending on the data partitioning used during evaluation. In the horizontal partitioning scheme, the query is:

```
PREFIX akt: <http://www.aktors.org/ontology/portal#>
SELECT * WHERE {
  ?pub akt:has-author <http://citeseer.rkbexplorer.com/id/
     resource-CSP86227-643fd78cf979d2c2551e04d62872a399> .
  ?pub akt:has-title ?title .
}
```

In the vertical partitioning scheme, the query is:

```
PREFIX akt: <http://www.aktors.org/ontology/portal#>
SELECT * WHERE {
  ?pub akt:has-author <http://example.org/id/Paul_Erdos> .
  ?pub akt:has-title ?title .
}
```

Query Q3

This query finds all proceedings with property `swrc:isbn`. It is based on SP^{2}Bench **Q3c**, using documents instead of articles.

```
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  ?article a foaf:Document .
  ?article swrc:isbn ?value .
}
```

Query Q4

This query selects all subjects that stand in any relation to Paul Erdös. It is based on SP^{2}Bench **Q10**. Newlines have been inserted for typesetting purposes.

The exact IRI used for Paul Erdös changes depending on the data partitioning used during evaluation. In the horizontal partitioning scheme, the query is:

```
SELECT * WHERE {
  ?s ?p <http://citeseer.rkbexplorer.com/id/
    resource-CSP86227-643fd78cf979d2c2551e04d62872a399>
}
```

In the vertical partitioning scheme, the query is:

```
SELECT * WHERE {
  ?s ?p <http://example.org/id/Paul_Erdos>
}
```