Efficient query processing in distributed RDF databases

Verheijen, W.J.A.

Award date:
2008

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

MASTER'S THESIS

Efficient Query Processing in Distributed RDF Databases

W.J.A. Verheijen

Supervisors:
Prof. dr. ir. G.J.P.M. Houben
Ir. K.A.M. van der Sluijs

Eindhoven, February 2008

Abstract

The Semantic Web takes the concept of data integration in a mediator architecture to a new level. Since the distribution of data is not managed centrally, query processing over a set of autonomous data stores poses new challenges for query optimization. Because URIs provide a global resource identification mechanism and RDF data stores have no schema, all data stores can potentially have information about some resource.

Improvement of the query processing performance is possible, however, when information about the overlap between data stores is available. This enables the mediator to avoid sending queries to data stores that return no relevant information, and to optimize the queries that are necessary.

A query processing approach is introduced that takes advantage of the knowledge about structural properties of the data stores' RDF graphs and the relationships between these graphs. The approach allows to balance completeness of the answer, total response time and first answer response time. A cost model and a prototype implementation are developed to evaluate the approach on various distributions and queries.

Preface

This thesis concludes my Master of Science studies at the department of mathematics and computer science at the Technische Universiteit Eindhoven.

I would like to thank Geert-Jan Houben for giving me the possibility to conduct this work with a strong research component in it. It was perceived as a welcome change after my more practical internship. He gave me the freedom to work out my own ideas, but also reminded me to make these ideas concrete for the purpose of communicating them.

I also would like to thank Kees van der Sluijs for reviewing my thesis, pointing out unclear and imprecise phrases and providing helpful comments. A word of thanks goes to Jeen Broekstra for interesting discussions at the beginning of the project and his explanation of the Sesame 2 server architecture.

Another word of gratitude goes to Ad Aerts for joining my assessment committee and reviewing my work.

Finally, I would like to thank my parents for supporting me during the entire period of my studies.

— Wouter Verheijen
February 2008

Contents

Chapter 1

Introduction

Today's world wide web (referred to as "the Web") consists of a huge amount of information that can be retrieved in an instant, with a single mouse click, from anywhere around the world. The size of the Web covered by search engines is estimated to be at least 24 billion pages[1].

To obtain knowledge from the Web, users must locate relevant web pages and information systems, and extract the information that they require from each web page and information system. Locating relevant information sources, and combining data from different, heterogeneous sources is a tedious process. The automation of this process is known as *data integration*, and focuses on the problem of querying across, and combining data from, multiple autonomous and heterogeneous data sources. While developing data integration systems is common nowadays, this, in itself, is a manual and tedious process.

The Semantic Web takes the concept of data integration to a new level. According to the Semantic Web FAQ[2], "the Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries."

The resource description framework (RDF) is a cornerstone in the Semantic Web framework, which enables automated processing of Web resources, and standardizes the way resources and the semantic relationships between resources are described. A motivation for RDF is to allow data to be processed outside the particular environment in which it was created, in a fashion that can work at Internet scale [KC04, Sec. 2.1]. One of design goals of RDF is that "anyone can make statements about any resource" [KC04]. The current Web's hyperlink has a similar goal: if person A places a some information on a web page, person B can refer to it simply by placing a hyperlink on a web page of his own, to person A's web page.

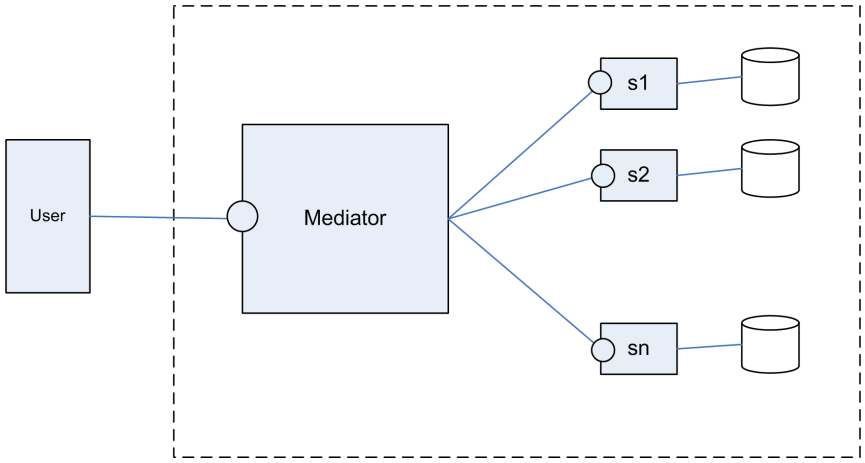


Figure 1.1: Distribution architecture. The boxes represent sites (machines), and the circles indicate interfaces for query processing. The cilinders on the right indicate databases, and the lines are communication channels.

1.1 Motivation

The focus of the research in the semantic web community has so far been mostly on query processing in a single RDF database. However, a key motivation for RDF and the semantic web is to enable mixing of information from different sites to obtain new information. One way to do so is to create a copy of the database from each site, and merge these copies into a single database. This architecture will be called an *integrated database system*. Integrating large amounts of data in a single database can however degrade the performance of query answering, and it may be difficult to have the most recent data from each site.

To improve the performance of query answering, a distributed system can be created by storing fragments of the original database onto one or more separate machines. This will be called a *parallel database system.* The goal of such a system is to speed up query processing by executing some parts of the query in parallel, on multiple machines, and combine the results.

Another type of distributed system is a *federated database system*. This is a database system type that logically integrates multiple autonomous database systems into a single virtual database. Hence, this architecture provides a single access point, but utilizes the autonomous external database systems to answer queries. The parallel and the federated database systems can be described with the same distribution architecture. This architecture is shown in Figure 1.1.

The single access point is called the *mediator*[3]. The other database systems are called *data stores*. A mediator in such a system performs the following actions [GP98]:

[1] http://www.worldwidewebsize.com/, retrieved 30 november 2007.

[2] W3C Semantic Web Frequently Asked Questions, http://www.w3.org/2001/sw/SW-FAQ

[3] The tasks that a mediator performs can encompass more than is considered here. Specifically, they often include the logical integration of heterogeneous data sources [Wie94]. In this thesis, the term is used for any system that provides a unified logical view over the data that is physically stored at multiple sites.

1. data store selection: choose the data stores that have data relevant for the user query.
2. query translation: find the query fragments to be executed on each data store, and send corresponding queries to each relevant data store.
3. result merging: combine the query results into the final query answer.

In both classes of distributed database systems, *data transparency* is important. This means that a user should not be required to know where the data is physically located or how it can be accessed. It can take several forms [SKS02, Cha. 19]: fragmentation transparency (users are not required to know which fragments of data exist); location transparency (users are not required to know the physical location of the data), and interface transparancy (users are not required to know the query interface that each data store uses).

In the federated database architecture, query processing poses additional difficulties. Since the data stores are assumed to be autonomous, the mediator has no control over how information is distributed over the data stores. Because of the assumption that "anyone can make statements about any resource," all data stores can potentially have information about some resource.

In both distribution architectures, the mediator is expected to benefit from information about the distribution of the database. Through this information, for instance, the mediator may avoid sending queries to data stores that return no relevant information.

For relational databases, both types of distributed systems have been extensively researched for over three decades. For the RDF data model, methods to describe a distributed database, and to optimize queries in such a system, are still in their infancies.

1.2 Scope

It is assumed that the data stores are accessed over a communication channel via some standardized mechanism, such as a TCP/IP network using the HTTP protocol. Issues such as data integrity, authorization and availability are inherently related to network-based information access, and therefore considered out of scope here.

All data stores are assumed to have the same query capabilities. The environment is assumed to remain constant: no change in the communication channel's availability, bandwidth or latency will occur.

In practice, when utilizing multiple data stores, there can be differences in how each data store describes the same concepts. This is common because the data stores may be unaware of each other at the time data was inserted. This issue is known as heterogeneity of the databases. An additional "alignment layer" that resolves these discrepancies can be added if necessary. A common way this is implemented is by using a "wrapper" in front of each data store (e.g. [TRV96]). Wrappers map from the general query language used by the mediators, to the particular query language, data model and interface of the data store. In this research, the assumption is made that the data in each data store is homogeneous; that

is, each identifier in the database refers to precisely one concept, and each (identifiable) concept is identified by a single identifier.

There exist several extensions to RDF that extend the semantics that are incorporated in the data model. Examples include RDFS and OWL. For this research, RDF is chosen as the data model. Since the semantics of RDF are contained in the other data models, most of the findings in this report should also be applicable to the data models that extend RDF. Nevertheless, improvements are to be expected when the additional semantics are taken into account.

1.3 Research questions

The main question for this thesis is: "How can queries in a distributed RDF database be processed effectively and efficiently?" To structure the answer to this question, the following research questions will be addressed.

1. How can the distribution of an RDF database be described, and how can the data in the data stores be related?
2. How can queries be answered in a distributed RDF database? How can the quality of a query processing approach be assessed?
3. How can the performance of query processing be improved when information about the distribution of the database is available?

The rest of this thesis is organized as follows. Chapter 2 introduces the RDF data model and ways to query RDF databases. Chapter 3 addresses the first research question, and Chapter 4 the second research question. Chapter 5 introduces a new index structure and associated query processing method. The effectiveness of this approach is investigated in Chapter 6. In Chapter 7 conclusions are presented.

Chapter 2

RDF data model and query processing

2.1 RDF data model

The Resource Description Framework (RDF) is a framework for representing information in the Web [KC04], in such a way that it is machine-processable. The underlying structure of any expression in RDF is a collection of statements. To identify the things referred to in RDF statements, Uniform Resource Identifiers *(URIs)* are used[1]. Each statement has three parts: a subject, an object, and a predicate (also called a property) that denotes a relationship.

Definition 2.1 (RDF statement [PAG06])**.** Assume there are pairwise disjoint infinite sets U (URIs), B (blank node identifiers) and L (literals). A tuple $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an *RDF statement*. In this tuple, s is the *subject*, p the *predicate*, and o the *object*. We denote the union $U \cup B \cup L$ by T (*RDF terms*). The term *triple* will be used interchangeably with the term "statement".

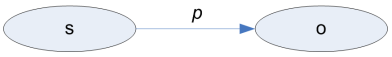


Figure 2.1: An RDF statement

A set of such statements is called an *RDF database*. An RDF database can be represented as a directed, labelled graph. In this representation, the subjects and objects of the statements are represented as nodes. The edges point from subject to object and are labeled with the predicate of a statement.

Definition 2.2 (RDF graph)**.** Let $G = \langle V, E, L, s, t, l \rangle$ be a labelled graph of an RDF database D where V is a set of nodes, E a set of edges, L a set of labels, $s, t : E \to V$ and $l : E \to L$. For every $e \in E$, we have $s(e) = r_1$, $t(e) = r_2$ and $l(e) = l_e$ if and only if $(r_1, l_e, r_2) \in D$.

[1] Technically, URI references (URIrefs) are used, which are URIs with an optional fragment identifier.

Hence, for an edge $e \in E$, $s(e)$ denotes the node from which e originates, and $t(e)$ denotes the node that it points to. $l(e)$ denotes the label (predicate) of e. Since an RDF graph is just another represetation of an RDF database, the terms "(RDF) database" and "(RDF) graph" will be used interchangeably.

In this thesis, instead of using full URIs, simple words will often be used as abbreviations, where the precise URI is not relevant. Blank node identifiers are prefixed with "`_:`", i.e. blank node identifer `abc` is denoted as `_:abc`. Literals are surrounded with double quotes, i.e. the literal `xyz` is denoted as "`xyz`".

A *map* [GHM04] is a function $\mu : T \to T$ preserving URIs and literals, i.e., $\mu(u) = u$ and $\mu(l) = l$ for all $u \in U$ and $l \in L$. Given a graph G, we define $\mu(G)$ as the set of all $(\mu(s), \mu(p), \mu(o))$ such that $(s, p, o) \in G$.

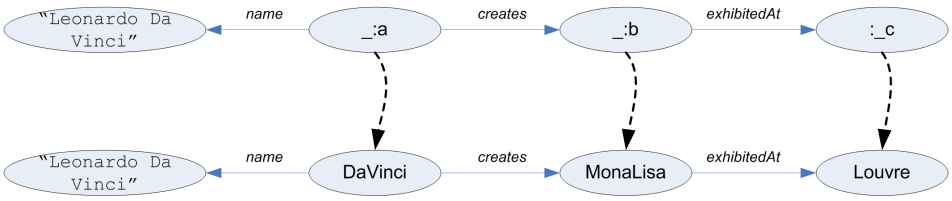


Figure 2.2: Two graphs. The second graph is an instance of the first one.

A map μ is *consistent* with G if $\mu(G)$ is an RDF graph, i.e. if s is the subject of a triple, then $\mu(s) \in U \cup B$, and if p is the predicate of a triple, then $\mu(p) \in U$. In this case, we say that the graph $\mu(G)$ is an *instance* of the graph G. An instance of G is *proper* if $\mu(G)$ has fewer blank nodes than G. This means that either μ sends a blank node to a URI or a literal, or identifies two blank nodes of G. An example of a graph and an instance of the graph is shown in Figure 2.2. Map μ maps blank node `_:a` to `DaVinci`, `_:b` to `MonaLisa`, and `_:c` to `Louvre`.

The meaning of map is overloaded and used also as follows: $\mu : G_1 \to G_2$ if there is a map μ such that $\mu(G_1)$ is a subgraph of G_2.

Two graphs G_1, G_2 are *isomorphic*, denoted $G_1 \cong G_2$, if there are maps μ_1, μ_2 such that $\mu_1(G_1) = G_2$ and $\mu_2(G_2) = G_1$.

2.2 RDF query processing

Several languages have been defined to query RDF graphs. The SPARQL query language [PS08] was first released in October 2004, and is a W3C Recommendation as of 15 January 2008.

RDF query languages typically allow users to specify *graph patterns.* A graph pattern is a set of *triple patterns.* A triple pattern is like an RDF statement except that each of the subject, predicate and object may be a variable. Formally, a tuple $(s, p, o) \in (U \cup V) \times (U \cup V) \times (U \cup V \cup L)$ is called a *triple pattern*, where V is a set of variable names.

Intuitively, a graph pattern can be seen as a template: literals and URIs in the graph pattern must match literally with an actual statement in the database, and variables take the

function of wildcards that match anything. This is the similar to the mapping of blank node identifiers that was described in the previous section. Indeed, blank nodes are equivalent with existentially quantified variables [Hay04]. This means that it is unnecessay to allow blank nodes in a triple pattern, as variables can be used instead.

In this thesis (and in SPARQL), variables will be denoted by a question mark followed by the variable name. When a graph pattern consists of multiple triple patterns, they are separated by a dot.

SPARQL supports, among others, the following operations:

1. Matching a single triple pattern
   (a) The property name is given. The subject is a variable, the object is a variable, both are variables, or no variables occur.
      i. `(?a, termP, term0)`
      ii. `(termS, termP, ?b)`
      iii. `(?a, termP, ?b)`
      iv. `(termS, termP, term0)`
   (b) The property is variable.
      i. `(termS, ?p, ?b)`
      ii. `(termS, ?p, term0)`
      iii. `(?a, ?p, term0)`
      iv. `(?a, ?p, ?b)`
2. Matching multiple triple patterns. Two or more triple patterns are combined by conjunction. Combining two triple patterns may result in either:
   (a) A natural join, which occurs when the same variable or term is used in multiple triple patterns. When the property name is given, these forms are possible:
      i. Subjects-join, e.g. `(?a, termP1, term01).(?a, termP2, term02).`
      ii. Objects-join, e.g. `(termS1, termP1, ?a).(termS2, termP2, ?a).`
      iii. Object-subject join, e.g. `(termS1, termP1, ?a).(?a, termP2, term02).`
      iv. Subject-object join, e.g. `(?a, termP1, term01).(termS2, termP2, ?a).`
   (b) Cartesian product, which occurs when the triple patterns do not share variables or terms. E.g. `(?a, termP1, term01).(?c, termP2, term02).`
3. Optional graph patterns (OPTIONAL). allow information to be added to the solution where the information is available. If the optional graph pattern does not match, no bindings are created, but the non-optional part of the solution is retained.
4. Alternatives (UNION) can be used to match at least one of the graph patterns mentioned.
5. Result reduction, such as:

(a) Projection: choose variables to be returned.

(b) Selection of answers in which a variable matches some boolean expression (FILTER)

(c) Restrict number of answers (LIMIT)

The focus for this research will be on triple patterns in which the property name is given (1a), and where the triple patterns are combined with natural joins (2a). This simplified query language means that a query can be fully represented as a connected RDF graph, in which the edges are URIs and the nodes can be URIs, literals or variables. Such a graph will be called a *query graph*.

A *valuation* [GHM04] v defines a way in which the variables in a query graph can be bound to RDF terms. Hence, a valuation for n variables is an n-ary tuple with bindings from variables to RDF terms. The notation $v(Q)$ for query graph Q is used to represent the graph obtained by replacing the variables in Q by RDF terms according to v.

A *matching* of the query graph Q in database D is a valuation v such that an instance of $v(Q)$ is a subgraph of D.

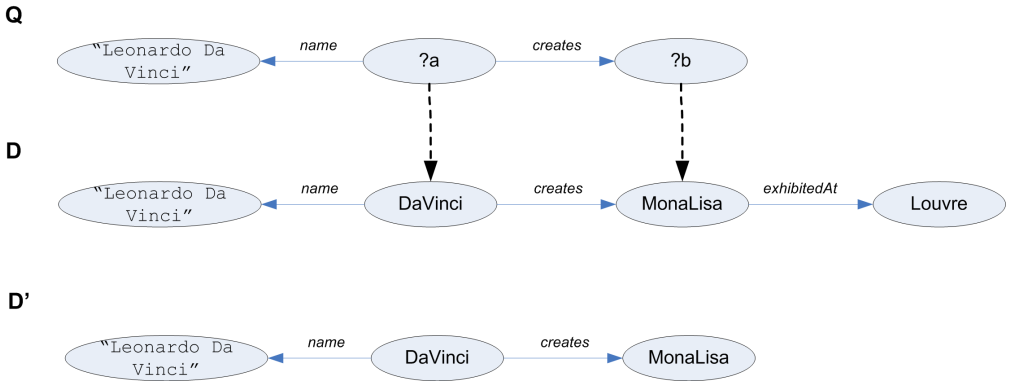


Figure 2.3: Matching of graph pattern Q to database D. D' is a matching of Q in D.

Figure 2.3 shows a matching of a graph pattern with two variables. The corresponding valuation binds `?a` to `DaVinci`, and `?b` to `MonaLisa`. D' is an valuation of Q, and a subgraph of D, and therefore a matching in D.

2.2.1 Answer sets

The representation and semantics of a query graph were defined above. The answer to such a query graph can also be represented in various ways. An *answer set* $A_D(Q)$ is a set of matchings in D. The matchings are also called 'tuples' in this case.

Consider the query graph in Figure 2.4. An answer set is shown in Figure 2.5a. By mapping the variables in the query graph to RDF terms according to the matching, an RDF graph is constructed for each matching. This is shown in Figure 2.5b. The set of RDF graphs that results from applying the matchings to the query graph Q is called the *extension* of Q. The

extension of Q in G is denoted as $Ext_G(Q)$. The term *instance* of a query graph is also used for "matching": each graph in the extension of Q is an instance of the query graph Q.

These matchings can again be combined to form a single RDF graph. This is done by taking the union of the matchings. The result is shown in Figure 2.5c. The structure of this final RDF graph obviously depends on the underlying RDF database. From this answer representation, it is less obvious how many answers are provided to the query, than with the other two forms. On the other hand, this representation can be smaller in size because each statement in it is unique.

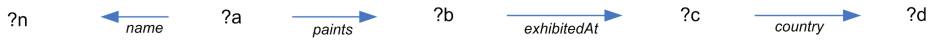
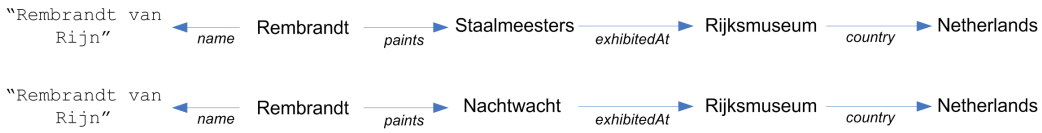


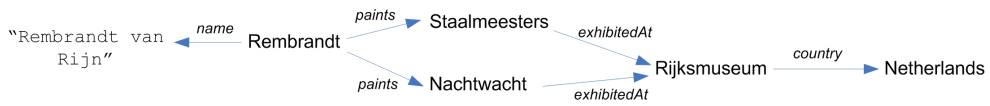
Figure 2.4: Query graph

n	a	b	c	d
"Rembrandt van Rijn"	Rembrandt	Staalmeesters	Rijksmuseum	Netherlands
"Rembrandt van Rijn"	Rembrandt	Nachtwacht	Rijksmuseum	Netherlands

(a) Tabular representation of the answer



(b) Matchings as set of instances of the query graph

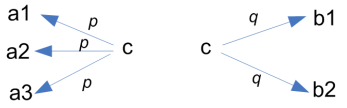


(c) Matchings as a single graph

Figure 2.5: Representations of answers to the query graph of Figure 2.4

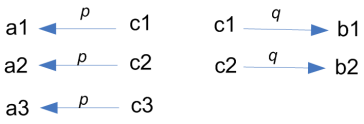
Answer set size

Given graph patterns P and Q that have one variable in common, what can be the cardinality of answer set $A(P \cup Q)$? Figure 2.6 shows three scenarios for the structure of the extensions of P and Q. The upper bound is achieved when every tuple in $A(P)$ can be combined with every tuple in $A(Q)$. This is similar to many-to-many cardinality between relations in relational databases. The result size is worst-case the same as the Cartesian product, and yields $|A(P)| \cdot |A(Q)|$ tuples. For one-to-one cardinality, the worst-case result size is $\min(|A(P)|, |A(Q)|)$. For many-to-one it is $|A(P)|$, and for one-to-many cardinality, it is $|A(Q)|$.



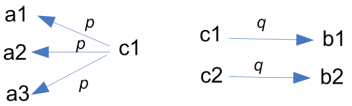
?x	?y	?z
a1	c	b1
a1	c	b2
a2	c	b1
a2	c	b2
a3	c	b1
a3	c	b2

(a) Many to many



?x	?y	?z
a1	c1	b1
a2	c2	b2

(b) One to one



?x	?y	?z
a1	c1	b1
a2	c1	b1
a3	c1	b1

(c) Many to one

Figure 2.6: Three scenarios for the structure of the graphs to be joined. Given pattern graph `(?y, p, ?x).(?y, q, ?z)`, the answer set for each scenario is shown on the right hand side.

Chapter 3

Distributed RDF graphs

This chapter considers the possibilities to distribute an RDF graph, and to describe such a distribution. Section 3.1 motivates the need for describing the distribution in a data integration system. Section 3.2 describes ways to define subgraphs of an RDF graph. Section 3.3 researches how graphs can be related to one another. Section 3.4 shows how subgraphs can be mapped onto data stores and investigates the consequences for the data store relationships.

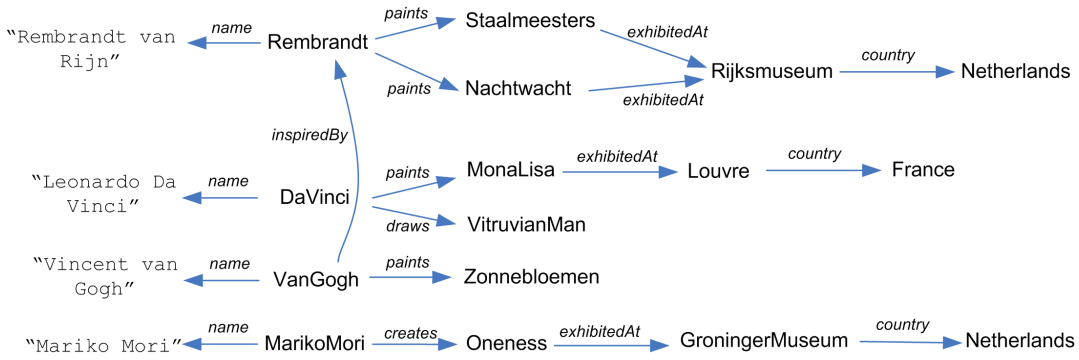


Figure 3.1: Example RDF graph

Throughout this thesis, an example RDF graph and distribution will be used to demonstrate the issues associated with distribution and query processing. The RDF graph is shown in Figure 3.1. A distribution over three data stores of this graph is shown in Figure 3.2.

3.1 Motivation for describing distributions

In relational databases, data is structured with the help of concepts like relations, relation schemas, superkeys and foreign keys [SKS02]. None of these is present in RDF. This means that a priori, any distribution of statements to data stores is possible. Consider an RDF graph that consists of n statements. These statements are to be distributed over k data stores.

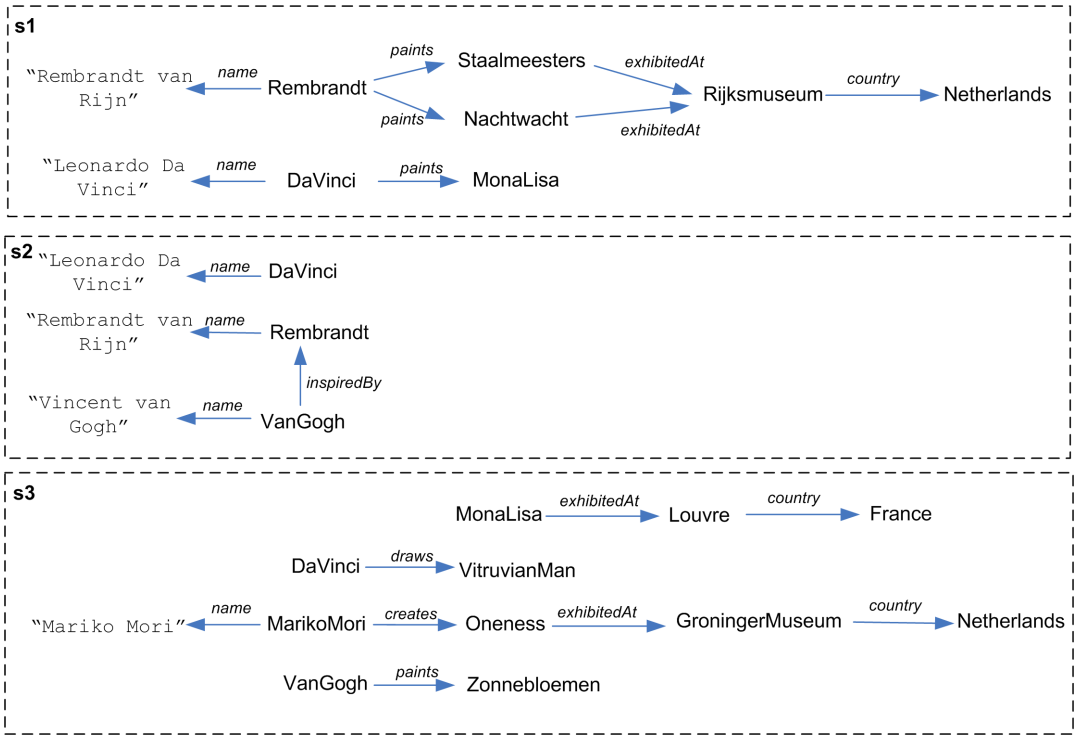


Figure 3.2: Example distribution of the RDF graph of Figure 3.1

Before addressing a more structured decomposition, first the question is addressed of how many distributions are theoretically possible. Assume that a statement can be distributed only to one data store. Hence, replication of a statement is not allowed. This is equivalent to the partitioning of a set of size n into k non-empty subsets. The Stirling numbers of the second kind[1] $S(n, k)$ count the number of ways to do so. As an example of the number of possibilities, consider a distribution with $k = 3$ data stores. The number of distributions for 3 statements up to 9 statements is: $S(3, 3) = 1$, $S(4, 3) = 6$, $S(5, 3) = 25$, $S(6, 3) = 90$, $S(7, 3) = 301$, $S(8, 3) = 906$, $S(9, 3) = 3025$.

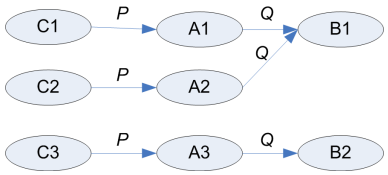


Figure 3.3: RDF graph consisting of 6 statements

Figure 3.3 represents an RDF graph of 6 statements. There are $S(6, 2) = 31$ possible ways to distribute these statements over two data stores. Four of them are shown in Figure 3.4.

[1] Weisstein, Eric W. "Stirling Number of the Second Kind." From MathWorld, http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html

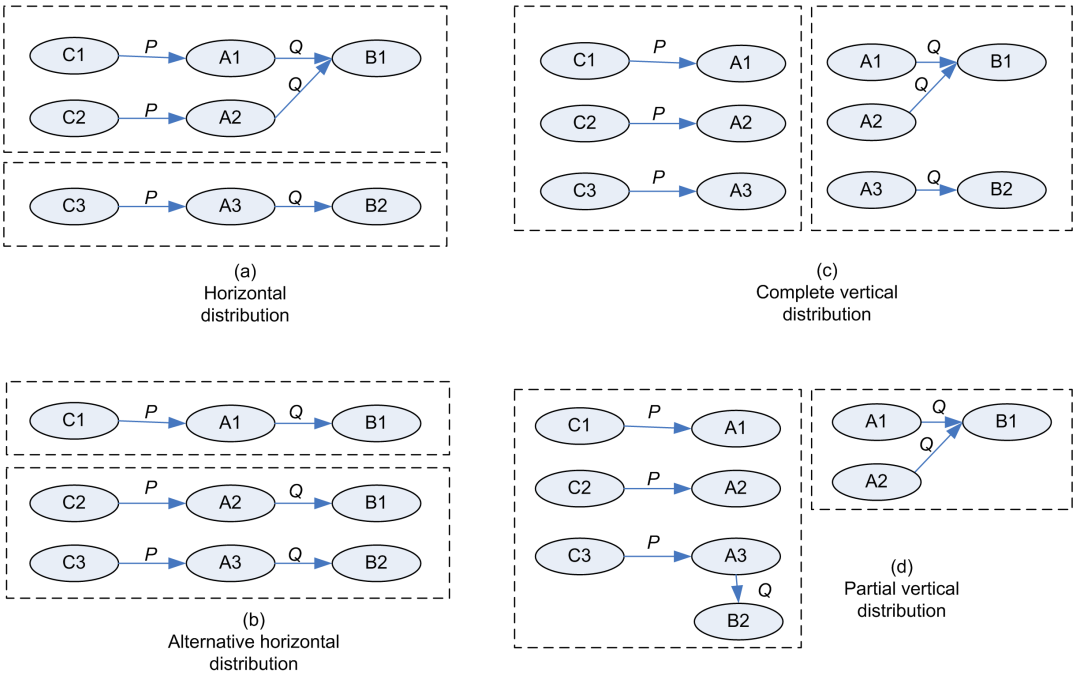


Figure 3.4: Four possible ways to distribute the graph of Figure 3.3 over two data stores.

The huge amount of possible distributions can also be motivated by the autonomous nature of data stores on the Web. This autonomy is supported by the RDF model ("anyone can make statements about any resource"). As a consequence, information about a resource is not confined in a single place.

Nevertheless, it can be assumed that the data contained in each data store is structured such that a partial characterization of the distribution can be made. In the running example, an alternative distribution could be to have one data store covering all Dutch musea and their collections, and another one that contains all the works done by Rembrandt van Rijn.

In this chapter, it is researched which meta-information from an RDF graph can be utilized to describe the distribution.

3.2 Fragment specification

This section starts with an overview of how fragments can be defined in relational databases. Hereafter, the applicability to RDF is researched.

3.2.1 Fragments in relational databases

A fragment of a relation is a subset of it. If a fragment is mapped onto multiple data stores, we say this fragment is replicated.

Horizontal fragmentation of a relation can be defined as follows [SKS02, Sec. 19.2.2]:

> In horizontal fragmentation, a relation r is partitioned into a number of subsets $r_1, r_2, \ldots, r_n$. Each tuple of relation r must belong to at least one of the fragments. [...] In general, a horizontal fragment can be defined as a *selection* on the global relation r. That is, we use a predicate P_i to construct fragment r_i:
>
> $$r_i = \sigma_{P_i}(r)$$
>
> We reconstruct the relation r by taking the union of all fragments; that is:
>
> $$r = r_1 \cup r_2 \cup \cdots \cup r_n$$

Vertical fragmentation is described as:

> In its simplest form, vertical fragmentation is the same as decomposition. It involves the definition of several subsets of attributes $R_1, R_2, \ldots, R_n$ of the schema R so that:
>
> $$R = R_1 \cup R_2 \cup \cdots \cup R_n$$
>
> Each fragment r_i of r is defined by
>
> $$r_i = \Pi_{R_i}(r)$$
>
> The fragmentation should be done in such a way that we can reconstruct relation r from the fragments by taking the natural join
>
> $$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

3.2.2 Graph patterns

Since an RDF graph has no schema, the above fragment definitions cannot be applied to RDF in a straightforward manner. Instead of defining fragments on relations, they are defined on subgraphs. Such a subgraph will be called a *view*. First, it will be analyzed which subgraphs of the database can be described by graph patterns.

Triple patterns

The smallest graph pattern consists of a single statement. Consider graph pattern $V_p =$`(?a, paints, ?b)`. The extension of this graph pattern is shown in Figure 3.5.

The predicate can also be a variable in a triple pattern: $V_{DV} =$`(DaVinci, ?p, ?b)`. The extension of this graph pattern is shown in Figure 3.6.

Combining triple patterns

Triple patterns can be combined in the same ways that regular triples can be combined.

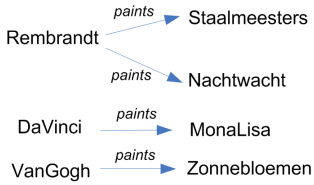


Figure 3.5: Extension of V_p in G

Figure 3.6: Extension of V_{DV} in G

Figure 3.7 shows the union of the extensions of graph patterns `(?a, paints, ?b)` and `(?c, exhibitedAt, ?d)`. From these graph patterns alone, one cannot predict if and how the instances of these graph patterns are connected. More precisely, whether there exists an instance of `(?a, paints, ?b)` and an instance of `(?c, exhibitedAt, ?d)` that can be combined to form a connected graph consisting of two triples. For example, the combination of statements `(Rembrandt, paints, Nachtwacht)` and `(Nachtwacht, exhibitedAt, Rijksmuseum)` is a connected graph, since the triples have one term in common, namely the URI `Nachtwacht`.

In the example above, suppose variable `?b` is identified with variable `?c`. This results in the view V_{pe} =`(?a, paints, ?bc).(?bc, exhibitedAt, ?d)`. The extension of this view is shown in Figure 3.8. The difference with Figure 3.7 is that everys statement in Figure 3.8 is part of the extension of V_{pe}.

3.2.3 Fragmentation

To limit the number of instances in the extension of a graph pattern, two operations on graph patterns are now defined. These operations, selection and projection, result in horizontal and vertical fragmentation of the extension respectively.

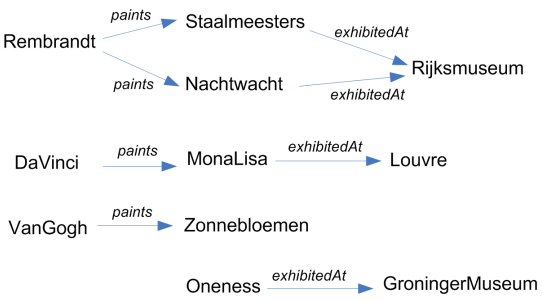


Figure 3.7: Union of the extensions of `(?a, paints, ?b)` and `(?c, exhibitedAt, ?d)`

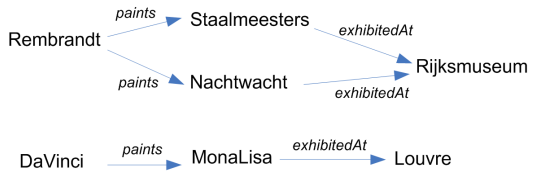


Figure 3.8: Extension of V_{pe} = `(?a, paints, ?bc).(?bc, exhibitedAt, ?d)`

Horizontal fragmentation

Definition 3.1. The extension of V, $Ext(V)$, is horizontally fragmented if some instance of $Ext(V)$ is contained in a different fragment than other instances.

Proposition 3.1. If V_1 is an instance of V_2 then $Ext_G(V_1) \subseteq Ext_G(V_2)$. In other words, all subgraphs of G that are instances of V_1, are also instances of V_2.

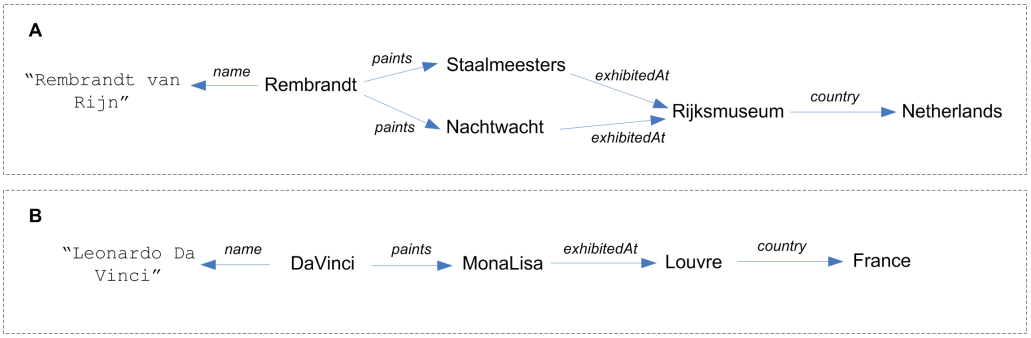


Figure 3.9: Partitioning in fragments A and B, of V_{npecNL} respectively V_{npecFR}. The union of these fragments is V_{npec}.

Consider graph pattern V_{npec} = `(?a, name, ?n).(?a, paints, ?b).(?b, exhibitedAt, ?d).(?d, country, ?e).`

V_{npecNL} = `(?a, name, ?n).(?a, paints, ?b).(?b, exhibitedAt, ?d).(?d, country, France).`

V_{npecFR} = `(?a, name, ?n).(?a, paints, ?b).(?b, exhibitedAt, ?d).(?d, country, Netherlands).`

V_{npecNL} and V_{npecFR} are both instances of V_{npec}, that are obtained by mapping variable `?e` to an RDF term. Alternatively, these views can be defined by using the selection operator (σ) on V_{npec}: $V_{npecNL} = \sigma_{\texttt{?e}=\text{Netherlands}}(V_{npec})$, and $V_{npecFR} = \sigma_{\texttt{?e}=\text{France}}(V_{npec})$.

The extensions of V_{npecNL} and V_{npecFR} are both subsets of the extension of V_{npec}. This is illustrated in Figure 3.9. Hence, V_{npecNL} and V_{npecFR} are horizontal fragments of V_{npec}.

Vertical fragmentation

If V_1 is a subgraph of V_2, then $\Pi_{V_1}(Ext(V_2))$ denotes the matchings of V_2, restricted to instances of V_1. The result includes only those subgraphs of the matchings of V_2, that are instances of V_1. This results in vertical fragmentation of $Ext(V_2)$.

Definition 3.2. The extension of V, $Ext(V)$, is vertically fragmented if V is fragmented in several subgraphs $V_1, \ldots, V_n$, and each fragment $Ext(V_i)$ of $Ext(V)$ is defined by

$$Ext(V_i) = \Pi_{V_i}(Ext(V))$$

Figure 3.10: Extension of $\Pi_{V_{np}}(Ext(V_{npecFR}))$.

Consider view V_{npecFR}, that was defined above. Next consider graph pattern V_{np} = `(?a, name, ?n).(?a, paints, ?b)`. Observe that graph pattern V_{np} is a subgraph of V_{npecFR}. V_{np} is now projected on $Ext(V_{npecFR})$, i.e. $\Pi_{V_{np}}(Ext(V_{npecFR}))$. The result is shown in Figure 3.10. This graph is a vertical fragment of $Ext(V_{npecFR}$.

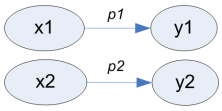
3.3 Instance overlap

In case the global RDF graph is not known, how can a statement from one data store be related to a statement from another data store?

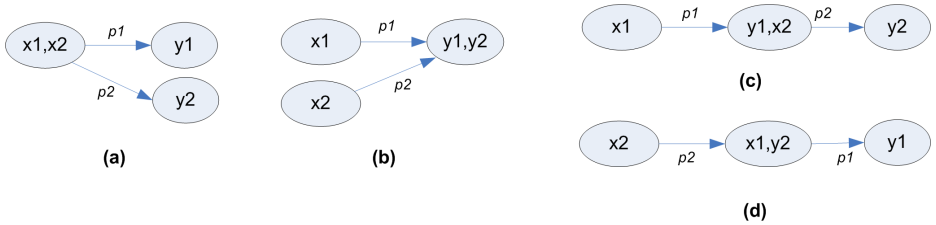
Figure 3.11: Two graphs, each consisting of a single statement.

Graph G_1 contains statement (x_1, p_1, y_1), and graph G_2 contains (x_2, p_2, y_2), as shown in Figure 3.11. It is researched in what ways these statements may be related when they are combined in a single RDF graph.

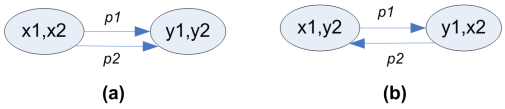
1. No overlap. $x_1 \neq x_2$, $y_1 \neq y_2$, $p_1 \neq p_2$.



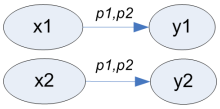
2. Single resource overlap. There are four cases: a) $x_1 = x_2$, b) $y_1 = y_2$, c) $y_1 = x_2$, or d) $y_2 = x_1$.



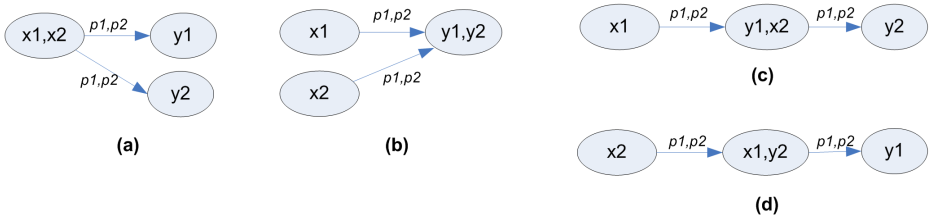
3. Two resources overlap. There are two cases: a) $x_1 = x_2$ and $y_1 = y_2$, b) $x_1 = y_2$ and $x_2 = y_1$.



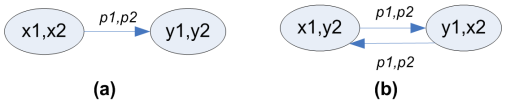
4. Same property, no resource overlap. The same four cases for resource overlap as under 1) are distinguished. But in this case $p_1 = p_2$.



5. Same property, single resource overlap.



6. Same property, two resources overlap. The most common case is the *replication* of the triple: $x_1 = x_2 \wedge p_1 = p_2 \wedge y_1 = y_2$, which corresponds to case (a) here.



These forms of overlap can occur within a data store, or between two data stores. The former will be called *intra-store overlap*, and the latter *inter-store overlap*.

3.4 Views and data stores

The distribution can now be defined as a mapping between the view descriptions and the set of data stores. Let $Ext_G(V)$ denote the global extension of view description V, and $Ext_s(V)$ the extension of V in data store s.

If all statements in the data store are within one or more views, the mapping for this data store is said to be *sound*. If only view descriptions consisting of a single statement `(?a, p, ?b)` are used, for all properties p in data store s, then the mapping is sound, for all possible databases in s. For example, data store s_1 can be described by the following view descriptions, denoted by mapping M_1.

1. $V_n =$ `(?a, name, ?b)`.

2. $V_p =$ `(?c, paints, ?d).`
3. $V_e =$ `(?e, exhibitedAt, ?f).`
4. $V_c =$ `(?g, country, ?h).`

Alternatively, this is also a sound mapping for s_1, denoted by M_2:

1. $V_{np} =$ `(?a, name, ?n).(?a, paints, ?b)`
2. $V_{ec} =$ `(?1e, exhibitedAt, ?1fg).(?1fg, country, ?1h).`

Yet another sound mapping, M_3, is:

1. V_{npecNL}.
2. V_{np}.
3. V_{ec}.

By using a projection on a graph pattern, the following sound mapping, M_4, is possible:

1. V_{npecNL}.
2. $\Pi_{V_{np}}(Ext(V_{npecFR}))$.

The question that now arises is: are some of these mappings better than others?

3.4.1 Vertical completeness

First, note that all subgraphs of a view description are also mapped to the same data store as that view description. View V_{np} is mapped to s_1. This implies that V_n and V_p are also mapped to s_1 (possibly among other data stores). If this was not the case, then it would be possible to have an instance of V_{np} on s_1, but the projection to view V_n, $\Pi_{V_n}(Ext_{s_1}(V_{np}))$, would contain instances that are not part of $Ext_{s_1}(V_n)$. This is a contradiction.

Graph patterns consisting of multiple triples give information about how individual statements within a data store are connected. So, in order to describe a data store, we are interesed in what the largest views are such that all statements in the data store are covered.

Definition 3.3. A view description V is *vertically complete* if the union of all extensions of subgraphs of V is equal to the extension of V. Formally,

$$\langle \forall_{W \subseteq V} Ext(W) = \Pi_W(Ext(V)) \rangle$$

If a view is not vertically complete, there exist statements that are in in the extension of some subgraph of this view description, but not in the extension of the view description itself. For data store s_1, view V_{npec} is not vertically complete, but V_{npecNL} is vertically complete. This will be demonstrated by the following example.

The following triple patterns are instantiated in s_1: `(?1a, name, ?1b)`, `(?1c, paints, ?1d)`, `(?1e, exhibitedAt, ?1f)`, `(?1g, country, ?1h)`.

It is now investigated which connected graph patterns can be formed, by unifying variables from two triple patterns. We start with the unification of `?1a` and `?1c`. If all instances of `(?1a, name, ?1b)` and all instances of `(?1c, paints, ?1d)` also occur in the extension of `(?1ac, name, ?1b).(?1ac, paints, ?1d)`, then this graph pattern is vertically complete. This is indeed the case for this data store.

The following unifications can be made:

Vertically complete (always path-forming): `?1a` = `?1c`, `?1f` = `?1g`.
Vertically incomplete (sometimes path-forming): `?1d` = `?1e`.

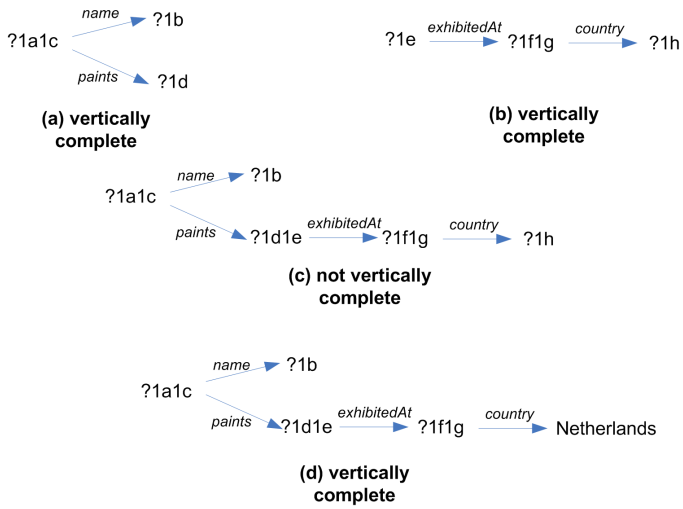


Figure 3.12: Graph patterns for data store s_1

The graphs that are generated by these specifications are shown in Figures 3.12a, 3.12b and 3.12c. For a specific instance of variable `?1h`, namely, `?1h` = `Netherlands`, the vertically incomplete unification `?1d` = `?1e` becomes vertically complete. The resulting graph pattern is shown in Figure 3.12d.

If a view description V_1 is vertically complete, and view description V_2 is an instance of V_2, then V_2 is also vertically complete. This follows from the fact that $Ext(V_2) \subseteq Ext(V_1)$.

3.4.2 Horizontal completeness

The mapping of a view to a data store can be further characterized. If view V is mapped onto data store s, does this mean that the complete extension of V is in s, or is there another data store s' that has instances of V that s does not have? In the latter case, V is horizontally fragmented among multiple data stores.

Definition 3.4. The mapping from V to s is *horizontally complete* if $Ext_s(V) = Ext_G(V)$.

Otherwise, it must be the case that $Ext_s(V) \subset Ext_G(V)$, and the mapping is horizontally incomplete.

Example 3.1. *The mapping of graph pattern V_{npecNL} to s_1 is horizontally complete. There are no instances of this graph pattern that are not on s_1.*

The mapping of graph pattern V_{np} to s_1 is not horizontally complete. There is an instance of this graph pattern that is not on s_1, namely `(VanGogh, name, "Vincent van Gogh").(VanGogh, paints, Zonnebloemen)`.

3.4.3 Example

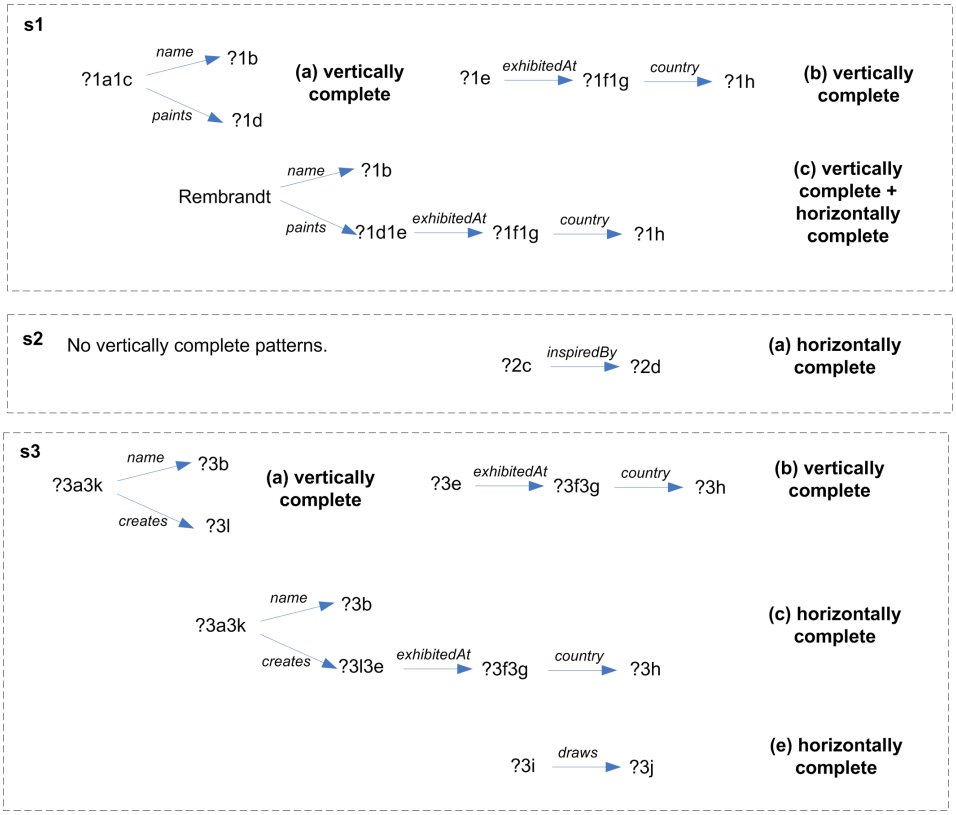


Figure 3.13: Vertically and horizontally complete graph patterns

For data store s_2, a sound mapping consists of the following triple patterns:

```
(?2a, name, ?2b),
(?2c, inspiredBy, ?2d).
```

For data store s_3, a sound mapping consists of the following triple patterns:

```
(?3a, name, ?3b),
(?3c, paints, ?3d),
(?3e, exhibitedAt, ?3f),
(?3g, country, ?3h),
```

`(?3i, draws, ?3j),`
`(?3k, creates, ?3l).`

Horizontally and vertically complete pattern graphs for the running example are shown in Figure 3.13. For s_2, no unification of variables results in local vertical completeness. The graph pattern `(?2c, inspiredBy, ?2d)` is horizontally complete. For s_3, local vertical completeness is obtained for the graphs that follow from these unifications: `?3f` $=$ `?3g`, and `?3a` $=$ `?3k`.

3.4.4 Inter-store overlap

The overlap types that were identified in Section 3.3 are now considered for inter-store overlap.

A graph is *replicated* if it occurs on multiple data stores. In our context, this is the case when a graph is part of two views that are mapped onto different data stores. For example, the triple `(DaVinci, name, "Leonardo Da Vinci")` is part of the extension of V_{np}, mapped to s_1, as well as V_n, mapped to s_2.

In some cases, replication can be modelled as a subset relationship between two views. In the running example,

$$Ext_{s_1}(\texttt{(?1a, name, ?1b)}) \subseteq Ext_{s_2}(\texttt{(?2a, name, ?2b)})$$

For the other types of overlap, vertical fragmentation is analyzed. Suppose views V_1 and V_2 are mapped to different data stores, s_1 respectively s_2. A variable from V_1 is unified with a variable from V_2, which results in view V'. Inter-store overlap exists between $Ext_{s_1}(V_1)$ and $Ext_{s_2}(V_2)$ if the union of these extensions contains matchings of V'.

In the running example, there exists a non-empty extension for the view descriptions resulting from the following unifications:

?2a =?3i (overlapping resource: DaVinci)
?2a =?3c (overlapping resource: VanGogh)
?1d =?3e (overlapping resource: MonaLisa)

The following unifications are implied by the subset relationship mentioned above:
?1a =?2a and ?1b =?2b.

From these unifications, graph patterns from multiple data stores may be combined. This results in global graph patterns, of which some examples are shown in Figure 3.14.

3.5 Conclusion

This chapter presented a number of techniques to describe fragments of an RDF graph. To describe fragments, views were introduced. View descriptions are graph patterns, possibly with selection and projection operators. A view is vertically complete if any instance that is

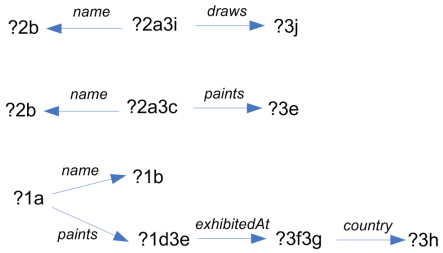


Figure 3.14: Some of the global graph patterns that result from the specifications of inter-store overlap.

part of a vertical fragment of the view, is also part of the whole view.

For the relationships between data stores, the notions of horizontal completeness of a view, and of inter-store overlap were defined. If a view mapping to a data store is horizontally complete, then this data store contains the complete extension of the view description.

Chapter 4

Query processing in distributed RDF graphs

This chapter researches how queries can be processed in a distributed RDF database, and how the quality of this process can be assessed.

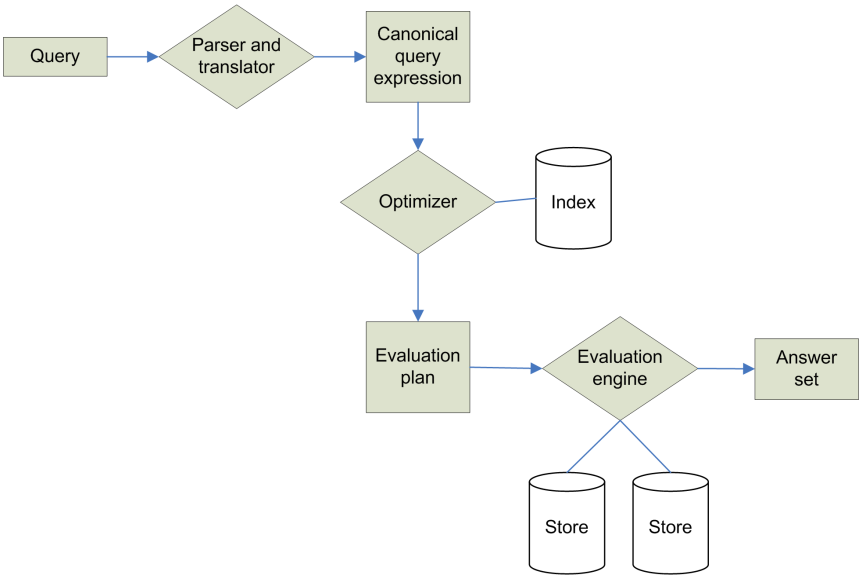


Figure 4.1: Query processing steps [SKS02]

The steps to process a query in a traditional database system are shown in Figure 4.1. The input query is parsed and translated into some internal form. Hereafter, the query optimizer produces a plan, called a *query evaluation plan* or query plan. This plan specifies how to evaluate the query, i.e. to obtain the answers. Generally, there are multiple ways to evaluate the query. The query optimizer determines the "best" way to do so. This can be done by assigning a cost to the evaluation plan. The plan with the lowest cost is elected as the best

plan. This plan is executed and the results are delivered back to the user.

In a distributed setting, there are multiple data stores that can be used during query processing. This is illustrated in Figure 4.2. The optimizer in a mediator (cf. Section 1.1) performs the tasks of data store selection and query translation. A query that is sent to a data store is called a *source query*. The *index structure* of the mediator consists of information about the data in the data stores, and can be used for the generation of source queries.

Section 4.1 introduces a cost model that is used to determine the quality of a query evaluation plan. In Section 4.2 several approaches for the generation of query plans are discussed. For the evaluation engine, an additional task is to merge the results from multiple data stores into a final answer set. This is discussed in Section 4.3.

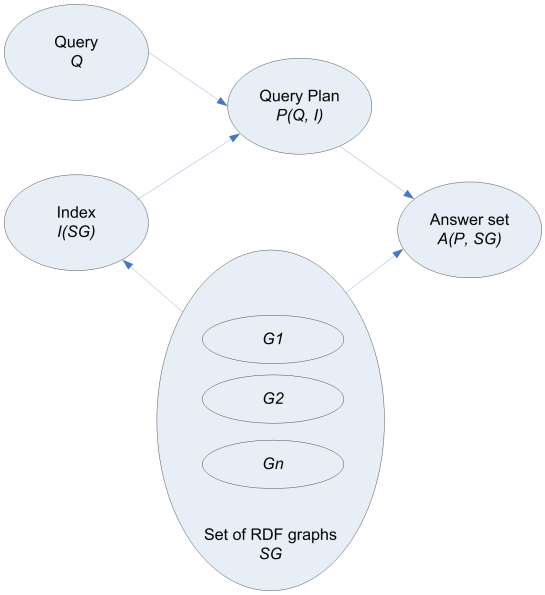


Figure 4.2: Dependencies in the distributed setup. The RDF graph has been replaced by a set of RDF graphs. The construction of the query plan can now take advantage of an optional index structure that was constructed on the basis of the set of RDF graphs.

4.1 Cost model

The goal of a cost model for our purposes is to compare the quality of query plans to one another. The task of the query optimizer is to choose the query plan with the lowest cost. The cost metric that is used by the optimizer is called the *objective function*. The following are examples of objective functions:

1. Complete answer response time: the time that elapses between the moment the user sends the query, and the moment the user has received the complete answer set.
2. Top-k answer response time: the time that elapses between the moment that the user sends the query, and the moment the user has received the first k answers.

Since both have important applications, a cost function for both goals will be researched.

There are many factors that influence the response time of a system that uses a distributed set-up. Stockinger et al. [SSSW01] distinguish static influences, that do not change over time, and dynamic influences. The static influences include bandwidth of the network link, size of data to be transferred, physical distance between two endpoints (measured in round trip time (RTT)), and network protocol overhead. The dynamic influences include network load and server load. Only the static influences will be modelled in the cost function, since the dynamic changes to the environment, including network and server load, were considered out of scope.

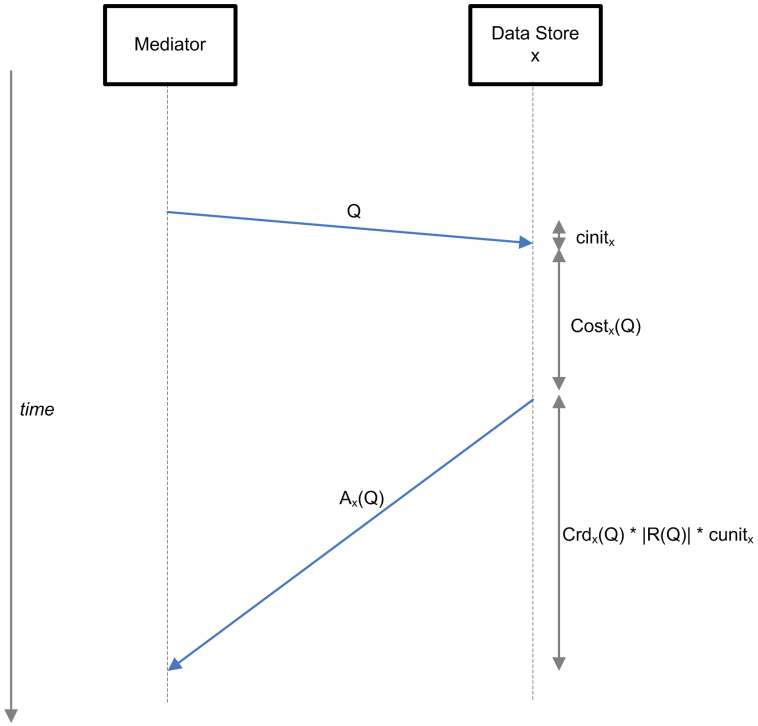


Figure 4.3: The processing of a source query Q, and associated costs in the cost model

A cost model should be validated by performing experiments. These should be performed in such a way that it is possible to determine for each variable what its impact is on the response time. Such a validation is not performed now. Hence, the cost model that is presented here is based on a number of intuitions and assumptions. These will be enumerated here as precisely as possible. Hereafter, cost predicates for each operator in the query plan will be formulated.

A complete parallel query evaluation plan is beyond the scope of this research. In particular, the optimal scheduling of the operations is not performed. Instead, a simplified schedule is assumed for the query plan. This will be discussed in more detail further on.

First, recall the process of query processing in a distributed database system. Four activities can be distinguished:

1. Transferring a query from the mediator to a data store.
2. Locally retrieving data from the database. Performed only at a data store.
3. Transferring results from a data store to the mediator.
4. Processing data by performing operations on it. Performed at the mediator and at the data stores.

The steps that are executed for each source query are graphically depicted in Figure 4.3. The cost predicates shown on the right of the figure will be introduced later.

4.1.1 Assumptions

1. The time to initiate the connection to a data store and to send the source query is constant. The time to transport the results back to the mediator are assumed to depend only on the size of the data that is to be transferred.
2. All RDF terms have the same size. This is expected to make the estimation of the the variable transport cost easier.
3. The data stores have one or more instance indexes that allow it to find resources in the data set quickly. The mediator does not have such an index.

   Several instance indexes have been proposed for RDF. Sesame 2, for example, uses three index types: for a fixed subject, predicate and object respectively. Yet Another RDF Store (YARS) [HD05] is a prototype system that uses a complete index structure including full-text indices for RDF triples. The goal of this index is to support evaluation of select-project-join queries. Using this index, any triple pattern can be mapped to a single index lookup operation.
4. Pipelining techniques can be applied to the operations that are executed on a data store. In pipelining, the output tuples of one operation are consumed by a second operation at the time they are being generated by the first operation. After the initial tuple has been output by such a pipeline, the throughput of the pipeline depends on the slowest operation in the pipeline.
5. Whenever a source query occurs more than once in the query plan, during query evaluation, the mediator can store the result of the first evaluation and re-use this result at subsequent occurences of the source query. This technique has been investigated in the area of multi-query optimization [RSSB00]. A possible disadvantage of this technique is that it may temporarily require additional storage space.
6. Two source queries can be performed in parallel if they are not sent to the same data store. If two source queries are sent to the same data store, one source query is sent first. Hereafter, the results of this query are received. This process is then repeated for the second source query.

4.1.2 Query evaluation plan

Based on the assumptions, a number of evaluation primitives is introduced now. An evaluation primitive is an algebraic operation annotated with instructions on how to evaluate it [SKS02, Cha. 13].

Answers to a query can be represented as tuples, as described in Section 2.2. This means that in the query plan, many relation algebra operations can be used on these answer sets. As was also shown in Section 2.2, the combination of triple patterns that share a variable or term is the same as a natural join performed on the extensions of these triple patterns. In addition to the algebraic operations that were used in Chapter 3, a new operation is used to indicate the sending of a source query to a data store. The notation ${}^{x}[Q]$ is used to indicate a source query Q that is sent to data store x.

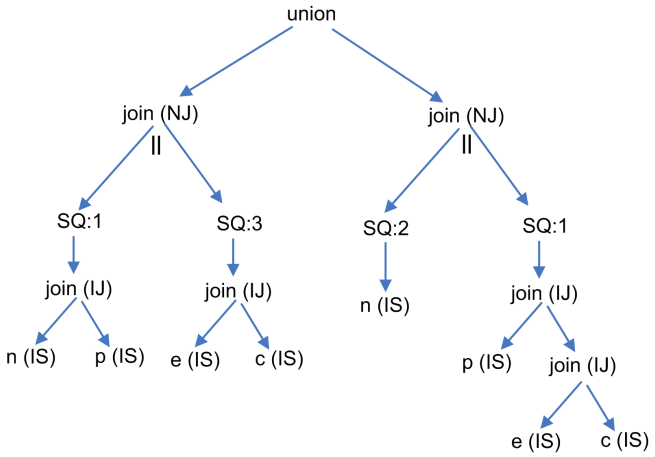


Figure 4.4: Example query evaluation plan. || indicates that the subtrees can be executed in parallel.

Algorithms

Only equality is considered as selection operator. Because data stores are assumed to have instance indexes, locating statements in the database can be done in a time that depends much less than linearly on the total number of statements in the database. Locating data through an instance index will be referred to as Index Scan *(IS)*.

Join operations can be performed on data stores as well as the mediator. Data stores have instance indices and can therefore perform indexed join operations, such as a hash join. This will be referred to as Indexed Join (IJ). No indexed join algorithms can be used on the mediator. In addition, the incoming answer sets from data stores cannot be assumed to be sorted. Therefore, it is assumed that the simplest join operation, nested-loop join (NJ), is used on the mediator.

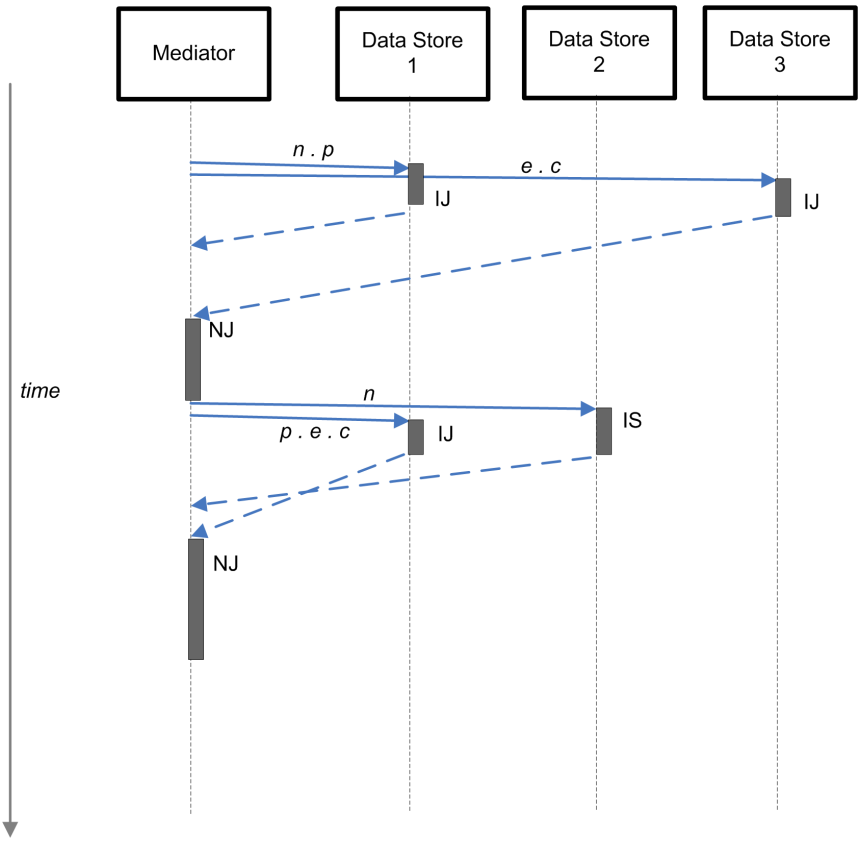


Figure 4.5: The evaluation of the example plan. Arrows pointing to the right indicate the transmission of a query. Arrows pointing to the left indicate the transmission of query results. The origin of the arrow indicates the start of the transmission, and the target of the arrow indicates the moment all result tuples have been received by the mediator.

Parallelism

Source queries that are not sent to the same data store can be executed in parallel. However, since the scheduling of source queries is not considered, it is not possible to determine the exact level of parallelism that can be obtained. Instead, a simplification of actual parallel execution is made.

Let $S(Q)$ denote the set of data stores that are used in source queries in plan Q. That is,

$$S(Q) = \{x \mid {}^{x}[Q'] \in Q\} \tag{4.1}$$

Now, $P \cdot Q$ can be executed in parallel if and only if $S(P) \cap S(Q) = \emptyset$, where $\cdot$ can be any binary operator; in our case: $\bowtie$ or $\cup$. A predicate par is defined to indicate the possibility of parallelism. It is assumed that only the mediator can exploit parallelism.

$$par_x(P, Q) = \begin{cases} S(P) \cap S(Q) = \emptyset & \text{if } x = 0 \\ \text{false} & \text{if } x \neq 0 \end{cases} \tag{4.2}$$

Some examples of this predicate:

$$par_0({}^2[\text{name}], {}^1[\text{paints} \bowtie \text{exhibitedAt} \bowtie \text{country}]) = \text{true}$$
$$par_0({}^2[\text{name}], {}^1[\text{paints}] \bowtie {}^3[\text{exhibitedAt} \bowtie \text{country}]) = \text{true}$$
$$par_0({}^1[\text{name}], {}^1[\text{paints}] \bowtie {}^3[\text{exhibitedAt} \bowtie \text{country}]) = \text{false}$$

Query evaluation plan example

To illustrate the query execution steps that are obtained by the assumptions we made, consider the following query plan expression. The letters represent triple patterns.

$$({}^1[n \bowtie p] \bowtie {}^3[e \bowtie c]) \cup ({}^2[n] \bowtie {}^1[p \bowtie e \bowtie c]) \tag{4.3}$$

The evaluation plan is depicted as a tree in Figure 4.4, annotated with the algorithms that are used for each operation. The execution of this plan, according to the rules of parallelism that were used above, is shown in Figure 4.5.

Observe that there are two subsequent branches that are themselves executed in parallel. The union operator (at the root of the tree) is not executed in parallel, because the set of data stores used in both operands is not disjunct. Let lhs and rhs denote the left-hand side respectively right-hand side of the query plan. Then $S(lhs) = \{1, 3\}$ and $S(rhs) = \{2, 1\}$; data store 1 participates in both branches.

As noted earlier, by choosing the ordering of the source queries in a smart way, it would be possible to obtain a higher degree of parallelism.

4.1.3 Cost predicates

For each action that is part of the process of query processing, a cost predicate is defined. Hereafter, these predicates are related to the algebraic operations of the query expression.

The predicate *Cost*(P) models the total response time to evaluate query plan P. All cost predicates will be given a location-specific suffix. For example, *Cost*$_x(P)$ is the cost of P at data store x. For the costs at the mediator, $x = 0$ is chosen.

To determine the variable costs in the cost model, the expected size of intermediate results should be known or estimated. Let *Crd*$_x(P)$ represent the estimate of the cardinality of the answer $A_x(P)$, i.e. of the answer that is obtained by performing query P at data store x. How this cardinality estimate is obtained is discussed in Section 4.1.6.

Transport costs

In RDF query languages such as SeRQL and SPARQL, there are two result formats for the answer to a query. In the *Select result format*, a list of mappings, from the set of variables in the query to RDF terms of the RDF graph, is returned. For SPARQL, this format is defined

in an XML syntax[1]. In the second result format, the *Construct result format*, an RDF graph is returned, in one of the many possible representations.

For transport cost calculation purposes, it is attractive if the variable cost are related proportionally with the number of tuples being returned. In the Construct result format each statement in the result set is unique. As a result, each additional answer tuple consists of one or more RDF statements. Hence, only an an upper bound can be given for this format. This is elaborated in Section 2.2. For simplicity, the Select result format is assumed.

Let $ctrans_x(Q)$ be the cost of sending query Q to data store x, and transferring answer $A_x(Q)$ back to the mediator. This consists of a constant and a variable part. The constant part $cinit_x$ is the cost of initializing communication with source x and sending the source query. The variable part is proportional to the number of tuples, $Crd_x(Q)$, times the number of mapping variables, $|R(Q)|$, to be transferred. Hence,

$$ctrans_x(Q) = cinit_x + Crd_x(Q) \cdot |R(Q)| \cdot cunit_x \tag{4.4}$$

where $cunit_x$ is the transmission cost per data unit, i.e. RDF term, from x to the mediator.

Local lookup costs

The time needed to retrieve all statements that match a triple pattern from the disk, after finding their location on disk, depends mainly on the disk throughput. However, when pipelining is used, the next operation can start as soon as the first statement has been read from disk. Therefore, a constant cost is used that represents the localization of the first statement. This means that there is a cost associated with the lookup of the first statement, but the reading of all subsequent statements is free. The predicate $clookup_x$ denotes these constant costs.

Join operation costs

Because the data stores have an instance index, a join operation can be performed with an indexed join algorithm *(IJ)*. By participating in the pipeline from the lookup on a data store to the reception on the mediator, it is the question which is the slowest operator in this pipeline.

In Section 2.2.1 it was shown that without any information about the cardinality constraints, the number of tuples resulting from a join can vary greatly. When no information about the size of the resulting answer set is known, also the cost to generate answer tuples for the pipeline cannot be estimated. When many tuples from the operands can be joined, the transfer is likely to be the bottleneck in the pipeline. When few tuples can be joined, the join operator will be the bottleneck.

With no additional information, we assume that there is a cost to determine the first answer of a join operation on a data store, that depends on the sizes of the operands. Hereafter, the

[1] In "SPARQL Query Results XML Format", W3C Proposed Recommendation 12 November 2007, http://www.w3.org/TR/rdf-sparql-XMLres/

throughput of the pipeline is delimited by the transfer of the answers to the mediator.
For $x \neq 0$, in addition to the lookup costs of a triple pattern, the following join cost is added:

$$cjoin_x(Q_1, Q_2) = (Crd_x(Q_1) + Crd_x(Q_2)) \cdot cproc_x \tag{4.5}$$

where $cproc_x$ is the processing cost per tuple.
For join operations on the mediator, it is not possible to use an index. It is assumed that a nested loop join *(NJ)* must be performed:

$$cjoin_0(Q_1, Q_2) = Crd_0(Q_1) \cdot Crd_0(Q_2) \cdot cproc_0 \tag{4.6}$$

Other operation costs

Let $cunion_x(Q_1, Q_2)$ be the cost of the union operation over two answers. It is assumed that the sets of mapping variables in Q_1 and Q_2 are the same. This means that this operation does not depend on the cardinalities of Q_1 and Q_2. The cost is therefore neglected.

$$cunion_x(Q_1, Q_2) = 0 \tag{4.7}$$

Let $cproj_x(Q)$ be the cost of the projection operation $\Pi_V(Q)$. Since this operation only consists of leaving out variable mappings that are not mentioned in V, this does not depend on the cardinalities of Q_1 and Q_2. The cost is therefore neglected.

$$cproj_x(Q) = 0 \tag{4.8}$$

Source query result re-use

The assumption was made that the results of a source query can be re-used when the same source query occurs multiple times in the query plan. For such duplicate queries, the result set of the first execution should be stored in memory or on disk for later re-use. We assume that the I/O activity of storing the results on the mediator can be performed in parallel with the regular processing activities. Therefore, the cost to store and retrieve the result set is neglected.

$$Cost_x({}^y[Q]) = \begin{cases} Cost_y(Q) + ctrans_y(Q) & \text{if first occurence of } {}^y[Q] \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

Query costs

The cost predicates for each query plan operator are now formulated as follows. Observe that the maximum of the costs of the operands is taken when parallelism is possible. Other-

wise, the sum of the operands is used.

$$Cost_x({}^y[Q']) = \begin{cases} Cost_y(Q') + ctrans_y(Q') & \text{if first occurence of } {}^y[Q] \\ 0 & \text{otherwise} \end{cases}$$

$$Cost_x(p_{AB}) = clookup_x(p_{AB})$$

$$Cost_x(Q_1 \bowtie Q_2) = \begin{cases} Cost_x(Q_1) + Cost_x(Q_2) + cjoin_x(Q_1, Q_2) & \text{if } \neg par(Q_1, Q_2) \\ \max(Cost_x(Q_1), Cost_x(Q_2)) + cjoin_x(Q_1, Q_2) & \text{if } par(Q_1, Q_2) \end{cases}$$

$$Cost_x(Q_1 \cup Q_2) = \begin{cases} Cost_x(Q_1) + Cost_x(Q_2) & \text{if } \neg par(Q_1, Q_2) \\ \max(Cost_x(Q_1), Cost_x(Q_2)) & \text{if } par(Q_1, Q_2) \end{cases}$$

$$Cost_x(\Pi_V(Q')) = Cost_x(Q')$$

4.1.4 First answer query cost

The second objective function that was mentioned at the beginning of Section 4.1 is that of Top-k answers response time. To accomodate for this in the cost model, a second cost predicate, *CostFA*, is introduced. This predicate designates the cost to construct the first answer tuple and deliver it back to the user. Hence, it effectively models the Top-1 answer response time, and will be referred to as first answer cost from now on. The assumption that is made is that the time to construct and return the first answer is significantly larger than the time to construct and return a subsequent answer.

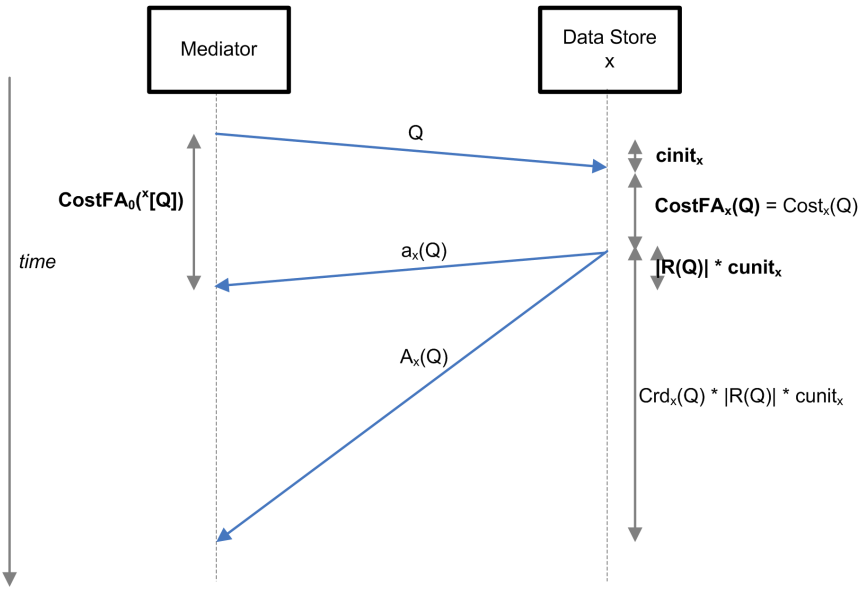


Figure 4.6: Execution of a source query, annotated with total costs and first answer costs. $a_x(Q)$ indicates the first answer tuple for query Q at data store x. Bold text indicates costs that contribute to the *CostFA* predicate.

The *CostFA* predicate is defined like the *Cost* predicate, with three differences:

1. The first answer cost for a source query ${}^{x}[Q]$ consists of the cost to set up the network connection (*cinit*), plus the processing cost to construct a single answer tuple, plus the cost to send this tuple back to the mediator. The processing cost for the construction of a single answer tuple is the same as for the *Cost* predicate, because it is independent of the number of tuples in the answer set.

   The variable transport costs depend on the number of tuples to be transferred. Therefore, in the cost for $ctrans_x$, the predicate $Crd_x(Q) \cdot |R(Q)| \cdot cunit_x$ is replaced by $|R(Q)| \cdot cunit_x$. This yields the following predicate for ${}^{y}[Q']$:

$$CostFA_x({}^{y}[Q']) = CostFA_y(Q') + cinit_x + |R(Q)| \cdot cunit_x$$

2. For the union operation, the cost to obtain the first answer is the minimum of the left and right hand side's cost to obtain the first answer. This can be seen intuitively as follows. Consider the query plan expression ${}^{1}[Q_1] \cup {}^{2}[Q_2]$ that is to be evaluated by the mediator. Suppose that it takes less time to compute ${}^{2}[Q_2]$ than to compute ${}^{1}[Q_1]$. Then, at some point in time, the mediator receives the first tuple of answer $A({}^{2}[Q_2])$. This answer is, without modification, part of the complete answer $A({}^{1}[Q_1] \cup {}^{2}[Q_2])$. The answer tuples $A({}^{2}[Q_2])$ can be returned immediately to the user, at no additional time penalty.

$$CostFA_x(Q_1 \cup Q_2) = \min(CostFA_x(Q_1), CostFA_x(Q_2))$$

3. The nested loop join depends on the number of tuples that are involved in both operands. Without any statistics on the number of matching tuples, it is also unknown when the first match is found. As an average case, we require that the inner set must be retrieved completely, while from the outer set, only a single tuple is required.

   Hence, $CostFA(Q_1 \bowtie Q_2)$ uses *CostFA* for Q_1, and *Cost* for Q_2:

$$CostFA_0(Q_1 \bowtie Q_2) = \begin{cases} CostFA_0(Q_1) + Cost_0(Q_2) + cjoinFA_0(Q_1, Q_2) & \text{if } \neg par(Q_1, Q_2) \\ \max(CostFA_0(Q_1), Cost_0(Q_2)) + cjoinFA_0(Q_1, Q_2) & \text{if } par(Q_1, Q_2) \end{cases}$$

   In the predicate $cjoin_0$, the expression $Crd_0(Q_1)$ is replaced by the number 1.

$$cjoinFA_0(Q_1, Q_2) = 1 \cdot Crd_0(Q_2) \cdot cproc_0$$

cinit	100
clookup	100
cunit	0.5
cproc	0.015

Table 4.1: Cost values chosen for the cost model

The complete specification of the *CostFA* predicate is:

$$\begin{aligned}
\mathit{CostFA}_x({}^{y}[Q']) &= \mathit{CostFA}_y(Q') + \mathit{cinit}_x + |R(Q)| \cdot \mathit{cunit}_x \\
\mathit{CostFA}_x(p_{AB}) &= \mathit{clookup}_x(p_{AB}) \\
\mathit{CostFA}_0(Q_1 \bowtie Q_2) &= \begin{cases} \mathit{CostFA}_0(Q_1) + \mathit{Cost}_0(Q_2) + \mathit{cjoinFA}_0(Q_1, Q_2) & \text{if } \neg par(Q_1, Q_2) \\ \max(\mathit{CostFA}_0(Q_1), \mathit{Cost}_0(Q_2)) + \mathit{cjoinFA}_0(Q_1, Q_2) & \text{if } par(Q_1, Q_2) \end{cases} \\
\mathit{CostFA}_x(Q_1 \bowtie Q_2) &= \mathit{CostFA}_x(Q_1) + \mathit{CostFA}_x(Q_2) + \mathit{cjoinFA}_x(Q_1, Q_2) \\
\mathit{CostFA}_x(Q_1 \cup Q_2) &= \min(\mathit{CostFA}_x(Q_1), \mathit{CostFA}_x(Q_2)) \\
\mathit{CostFA}_x(\Pi_V(Q')) &= \mathit{CostFA}_x(Q')
\end{aligned}$$

4.1.5 Choosing values

The processing costs in a data store consist of retrieving data from disk to main memory, reading from main memory, performing calculations on the data, and writing the results to main memory, and possibly to disk, again. The time it takes to retrieve data from disk or from memory depends on the access time (latency) and the data transfer rate (throughput). Disks are much slower than memory, both in access time and throughput, and hence are the dominating cost.

The *clookup* predicate models the time to look up the first instance of a triple pattern. A lookup through an instance index generally takes a number of disk access. Let *disklatency* denote the latency of the disk. Now, $\mathit{clookup} = 10 \cdot \mathit{disklatency}$ is chosen. Disk access times are usually around 10 ms. Hence, $\mathit{clookup} = 10 \cdot 10 = 100$ is chosen.

For the network access time, *cinit*, a network connection must be established, which takes a number of data exchanges between mediator and data store. Let *networklatency* denote the latency of the communication channel. Now, $\mathit{cinit} = 10 \cdot \mathit{networklatency}$ is chosen. Network latency is around 1 ms on a LAN, while it can be 100 to 1000 ms on a WAN. Assume that *networklatency* = 10 ms. Then, $\mathit{cinit} = 100$.

We assume that each RDF term has a size of 50 bytes. Assume that the application-level throughput of the network link is 100 kB per second. That is, each second 100 kB consisting only of data units (RDF terms) are transferred. This is 0.5 millisecond per RDF term. Then, $\mathit{cunit} = 0.5$.

For *cproc* the combined I/O and CPU cost are taken. Assume that the application-level throughput of the disk is 10,000 kB per second. This is 0.05 millisecond per RDF term. Assuming that a tuple consists on average of 3 RDF terms, $\mathit{cproc} = 0.015$.

4.1.6 Join size estimation

The cardinality of the answer to query Q at data store x is equal to $|A_x(Q)|$. Generally though, before evaluating a query, the mediator does not know this cardinality for every possible source query Q. Instead, some form of estimation for the cardinality of query Q should be provided. This estimation is represented as a predicate $Crd_x(Q)$, for the cardinality estimate of the answer $A_x(Q)$. As tuples are used to represent the answer set, we can formulate an upper bound on the number of tuples in the result of a join.

We assume that no additional information is available about the cardinality constraints of the graphs. In Section 2.2.1 it was demonstrated that there can be a huge variation in the size of the answer set that results from a join between triple patterns. As an "average case" estimate for the number of tuples in $Q_1 \bowtie Q_2$ is taken:

$$Crd_x(Q_1 \bowtie Q_2) = \max(Crd_x(Q_1), Crd_x(Q_2)) \tag{4.10}$$

For the union of two answer sets, the cardinality is estimated as the sum of the two answer sets:

$$Crd_x(Q_1 \cup Q_2) = Crd_x(Q_1) + Crd_x(Q_2) \tag{4.11}$$

It is assumed that $Crd_x(p)$ for every triple pattern `(?a, p, ?b)` is known, and that for other query forms, an estimate is constructed.

4.2 Query optimization

Two basic requirements of query processing in traditional database systems are *correctness* and *completeness* of the answer set. In some applications, especially on the Web, the completeness requirement is of lesser importance [VP98]: users are often satisfied with a partial, but nevertheless correct answer set.

The transfer of data over the network is the greatest bottleneck in distributed setups. Hence, reducing necessary communication between mediator and data stores has the highest potential for optimization [AS05]. For the objective function minimization of the response time, the following proposition is therefore made:

Proposition 4.1. A query plan should generally avoid source queries that produce an answer set that:

1. is empty, or
2. contains answers that are not part of the final query answer, or
3. contains answers that are duplicates of answers produced by other source queries.

Ouzzani and Bouguettaya [OB04] propose five dimensions along which query optimization for Web-based data integration systems can be compared. These are:

1. Scalability: a measure for the degradation of the query optimization technique when the number of data stores grows.
2. Autonomy preservation: how much information is required from the data stores for deploying the optimization technique.
3. Optimization performance: performance evaluation of the objective function under different scenarios.
4. Adaptiveness: the ability of the optimization technique to take into account unexpected changes.
5. Capabilities: whether the optimization technique takes into account data stores with different and limited query capabilities.

The last two dimensions are not relevant for this research, as was pointed out in Section 1.2. Hence, the optimization techniques discussed in this chapter are compared on the first three dimensions. For the performance dimension, the cost model of Section 4.1 is used.

4.2.1 No index

No knowledge is available of how the graph is distributed among the data stores. This means that each pair of instances can have a common resource reference. If source queries of multiple triple patterns are used, some statements may inadvertedly not be retrieved. This means that to obtain a complete and correct answer set, every triple pattern in the query graph must be evaluated on every data store. This approach will be called the *naive method*.

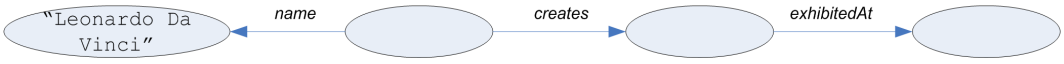


Figure 4.7: Query graph

Example 4.1. *Consider the query graph Q in Figure 4.7, which will be used on the distribution in Figure 3.2. Introduce the following abbreviations:*

$$n = \texttt{(?A, name, "Leonardo Da Vinci")}$$
$$p = \texttt{(?A, paints, ?C)}$$
$$e = \texttt{(?C, exhibitedAt, ?D)}$$
$$c = \texttt{(?D, country, ?E)}$$

The query plan is now expressed as:

$$A(Q) = ({}^1[n] \cup {}^2[n] \cup {}^3[n]) \bowtie ({}^1[p] \cup {}^2[p] \cup {}^3[p]) \bowtie ({}^1[e] \cup {}^2[e] \cup {}^3[e]) \bowtie ({}^1[c] \cup {}^2[c] \cup {}^3[c])$$

This query plan consists of $3 \times 4 = 12$ *distinct source queries.*

Evaluation

First, the time complexity is investigated. Let s be the number of data stores; t the size of the input query (in number of triple patterns); T the longest time between the moment a source query is sent, and the reception of the complete answer set to it.

For each triple pattern, s data stores are queried. The total number of source queries is therefore $\Theta(s \cdot t)$. The query answering time per data store is $O(T)$. The source queries that are performed per triple pattern can be performed in parallel (cf. Section 4.1.2). The mediator performs $t - 1$ join operations on the results. Let T_{join} be the longest time to perform such a join. The total query answering time is then: $O((T + T_{join}) \cdot t)$.

For the complete answer response cost, let $Q = \{q_1, \cdots, q_t\}$ be the input query. Let P be the query plan expression generated by this approach. It is of the form $P = P_1 \bowtie \cdots \bowtie P_t$, where each P_i for $1 \leq i \leq t$ is of the form $P_i = ({}^1[q_i] \cup \cdots \cup {}^s[q_i])$.

$$Cost_0(P) = Cost_0(P_1) + \sum_{i=2}^{t}(Cost_0(P_i) + cjoin_0(P_{i-1}, P_i))$$

$$Cost_0(P_i) = \max_{j=1}^{s}(clookup_j(q_i) + ctrans_j(q_i))$$

The first answer can only be given when all parts P_i of the join query have an answer.

Since no index structure is utilized, the goal of preservation of the data store autonomy is reached. However, in terms of scalability and optimizer performance, this approach is very weak:

- All data stores are queried for every triple pattern in the query graph, while some of the data stores may not contain instances of a triple pattern. Hence, the approach is not scalable with respect to number of data stores.

- No effort is taken to combine source queries for the same data store. Joins are only performed by the mediator. Hence, the approach is not scalable with respect to the input query size.

4.2.2 Property index

The first of these problems is now addressed. Suppose the mediator has a mapping from triple patterns to a set of data stores that have instances of this triple pattern. For now, all triple patterns of the form `(?a, p, ?b)`, for each predicate p, are considered.

The index structure consists of a set of mappings from a property name to the set of sources, $I : P \rightarrow S$, in which the property name occurs at least once:

$$(p \mapsto s) \in I \Leftrightarrow |Ext_s(\texttt{(?a,}p\texttt{, ?b)})| > 0 \tag{4.12}$$

Property	$\in s_i$	$\notin s_i$
country	1, 3	2
creates	3	1, 2
draws	3	1,2
exhibitedAt	1, 3	2
inspiredBy	2	1, 3
name	1, 2, 3	
paints	1	2, 3

Table 4.2: Index structure: mapping from property name to set of data stores

Example 4.2. *The set of mappings from Table 4.2 is taken.*

$$A(Q) = ({}^1[n] \cup {}^2[n] \cup {}^3[n]) \bowtie ({}^1[p]) \bowtie ({}^1[e] \cup {}^3[e]) \bowtie ({}^1[c] \cup {}^3[c])$$

There are 8 distinct source queries, which is 4 fewer than with the naive approach.

Evaluation

Again, the time complexity is investigated. In addition to the predicates defined for the previous approach, let s' be the average number of data stores that have instances of a triple pattern. The total number of source queries is $\Theta(s' \cdot t)$. The query answering time per data store is again $O(T)$. Also the total query answering time is the same: $O((T + T_{join}) \cdot t)$.

The worst-case answering time complexity is thus the same as in the naive approach, even though fewer source queries may be sent. There are more opportunities for parallelism however. When the data stores used in one join operand are all different from the data stores used in the other join operand, then all these source queries can all be performd in parallel.

The complete answer response cost is the same, except that s is replaced by s'. This results in the following predicate for P_i:

$$Cost_0(P_i) = \max_{j=1}^{s'}(clookup_j(q_i) + ctrans_j(q_i))$$

Scalability in terms of the size of the user query is the same as in the previous approach. Scalability w.r.t. the number of data stores is better, because source queries that produce empty answers are avoided (cf. Proposition 4.1.) The approach assumes the presence of statistics from the data stores about which properties they contain.

4.2.3 Graph index

The second problem with the naive approach is considered now: when can joins be performed by a data store, rather than by the mediator? To do so, the concept of vertical completeness of Section 3.4.1 is applied to source queries.

A	C
DaVinci	MonaLisa

(a) $A_{s_1}(V_{npLDV})$

A
DaVinci

(b) $A_{s_2}(V_{nLDV})$

A	C
VanGogh	Zonnebloemen

(c) $A_{s_3}(V_p)$

C	D	E
Staalmeesters	Rijksmuseum	Netherlands
Nachtwacht	Rijksmuseum	Netherlands

(d) $A_{s_1}(V_{ec})$

C	D	E
MonaLisa	Louvre	France
Oneness	GroningerMuseum	Netherlands

(e) $A_{s_3}(V_{ec})$

Table 4.3: Answer sets for the source queries used in the example.

Proposition 4.2. A source query that represents a view description that is vertically complete does not have to be split.

Example 4.3. *Consider again the query from Figure 4.7. There is no data store on which this query is mapped completely. Instead, subgraphs of the query graph are attempted. View V_{np} is vertically complete for data store s_1. This means that the instantiation of V_{np} as V_{npLDV} =*`(?A, name, "Leonardo Da Vinci").(?A, paints, ?C)` *must also be vertically complete for s_1 (cf. Section 3.4.1.) By Proposition 4.2, source query ${}^1[V_{npLDV}]$ can be used.*

For data stores s_2 and s_3 no vertical completeness is known for V_{npLDV}. Hence, separate source queries for each triple pattern are used. Let view V_{nLDV} = `(?A, name, "Leonardo Da Vinci")` *and $V_p =$* `(?A, paints, ?C)`*. Assume that V_{nLDV} is mapped to data store s_2 (note that V_n is also mapped to s_3 though), and V_p to s_3.*

View V_{ec} is vertically complete for both s_1 and s_3. All subgraphs (i.e. V_{ec} itself, and V_e and V_c) are only mapped to s_1 and s_3.

With this information, the following source queries can potentially contribute to the final query answer:

$$
\begin{aligned}
{}^1[V_{npLDV}] &= {}^1[\texttt{(?A, name, "Leonardo Da Vinci").(?A, paints, ?C)}] \\
{}^2[V_{nLDV}] &= {}^2[\texttt{(?A, name, "Leonardo Da Vinci")}] \\
{}^3[V_p] &= {}^3[\texttt{?A, paints, ?C}] \\
{}^1[V_{ec}] &= {}^1[\texttt{?C, exhibitedAt, ?D).(?D, country, ?E)}]
\end{aligned}
$$

$$^3[V_{ec}] = {}^3[\texttt{?C, exhibitedAt, ?D).(?D, country, ?E)}]$$

The answer set for each source query is shown in Table 4.3.

It cannot be seen which resource overlaps exist between the answer sets of these source queries. If the mediator has meta-information about inter-store overlap, it can be used to avoid unnecessary source queries.

Proposition 4.3. Joins between the answers of source queries that produce an empty answer should be avoided. If a source query cannot particpate in any join, it should be removed.

Example 4.4. *No overlap on* `?C` *exists between* $^3[V_p]$ *and* $^1[V_{ec}]$*, nor between* $^3[V_p]$ *and* $^3[V_{ec}]$*. This implies that*

$$^3[V_p] \bowtie {}^1[V_{ec}] = \emptyset$$
$$^3[V_p] \bowtie {}^3[V_{ec}] = \emptyset$$

As at least one of $^1[V_{ec}]$ *and* $^1[V_{ec}]$ *must be used in the query plan,* $^3[V_p]$ *cannot provide any part of an answer tuple. Hence, it is unnecessary.*

The same argument can be used for $^1[V_{ec}]$*. If it is known that there is no overlap on* `?C` *between this source query and* $^1[V_{npLDV}]$*, or between* $^1[V_{ec}]$ *and* $^3[V_{ec}]$*, then it is unnecessary. It can be verified in Table 4.3 that* $^1[V_{ec}]$ *cannot contribute to an answer to the query.*

The extension of V_{nLDV} *is replicated on data stores* s_1 *and* s_2*: the answer to* $^1[V_{nLDV}]$ *is equal to* $^2[V_{nLDV}]$*. This means that* $^2[V_{nLDV}]$ *is unnecessary.*

Hence, the following source queries are necessary to obtain a correct and complete answer set:

$$^1[V_{npLDV}], {}^3[V_{ec}]$$

4.3 Result merging

In Section 2.2.1 it was argued that an answer set can be represented as a set of graphs, or a set of tuples. Each representation has a different approach in combining the answers of the source queries:

1. For graphs: combine the statements that result from each source query. Find instances of the query graph in the resulting graph.
2. For tuples: try all natural join combinations between source query answer sets. This means that for each variable, a join should be attempted for all pairs of source queries that share this variable.

The latter has similar semantics as in the relational model, and is assumed in the cost model. Investigation of an approach better suited for RDF is future work.

If the data store databases contain blank nodes, there are multiple ways in which the answers from the data stores can be combined [GHM04].

1. Union of answer tuples. The blank node identifiers are preserved, and therefore multiple data stores may refer to the same blank node.
2. Merge of answer tuples, which means to rename blank nodes if necessary avoid name clashes.

Blank nodes have, by definition, a scope limited to one RDF graph. This means that in the context of data integration, two data stores cannot refer to the same blank node. Therefore, the merge semantics are appropriate.

In contrast, for a parallel database system, multiple data stores logically represent a single RDF graph, and they should be able to refer to the same blank node. Therefore, the union semantics are appropriate. Note that in this case, the data stores themselves should also use union semantics in constructing the answer. This means that they should not rename blank node identifiers in an answer tuple if it clashes with a blank node identifier in another answer tuple.

These semantics are also relevant for the elimination of duplicate answer tuples. Gutierrez et al. [GHM04] show that deciding whether there are duplicate tuples in the answer set is coNP-complete when union semantics are used, and can be done in polynomial time if merge semantics are used.

4.4 Conclusion

Two cost functions were defined to model the performance of different query processing approaches. The performance is measured as total response time, and as first answer response time. Validating and improving this model based on actual measurements is future work.

Due to the exploitation of parallelism in the cost model, a query processing approach that is aware of which predicates are instantiated on each data store (property-based approach) has only a minor advantage over a naive processing approach.

A source query in a query plan should be avoided if: 1) the answer set is empty; 2) the answers are not part of the final answer set; 3) the answers are also produced by other source queries in the plan. Avoiding an empty answer set can be done if the index structure stores for each data store which graph patterns are instantiated on it. Avoiding answers that are not part of the final answer can be done if the mediator is aware of (the absense of) overlap between two source queries. To avoid retrieving duplicate answers, subset relations between the extensions of views can be used. A larger source query can be used if it represents a vertically complete view description. If it represents a vertically incomplete view description, the final answer set may not be complete.

Chapter 5

Overlap-based query planning

This chapter introduces a query processing approach that utilizes some of the types of meta-information that were described in Chapter 3. The ideas from the previous chapter are used to formulate a concrete approach.

In Section 5.1, an index structure is introduced that represents knowledge of resource overlap between data stores. Hereafter, it is researched how this index structure can be employed during query processing tasks. The initial query plan that is constructed on this index structure may be of low quality, as we will see later. To better assess the merits of the approach, it should also be researched how the initial plan can be transformed into a high-quality plan. The Planning by Rewriting (PbR) [Amb99] paradigm fits into this scenario. In this approach, it is assumed that an initial plan can be generated efficiently, and that this plan can be transformed by rewrite rules. The applicability of this paradigm to the current query planning problem is analyzed. The generation of the initial plan is described in Section 5.2. Section 5.3 researches the transformations of the generated plans. A local search method is used to control the plan rewriting. This is discussed in Section 5.4. The cost model that was described in Section 4.1 is used to determine the quality of the generated plans.

5.1 Indexing overlap

Let $D = (G_1, \ldots, G_n)$ be a distribution of RDF graph G in n subgraphs. This is not necessarily a partitioning, since replication of statements is possible. Each graph G_i represents the database of a data store.

5.1.1 Bigraphs

Definition 5.1. A *bigraph* $B = (t_1, t_2)$ is a connected RDF graph consisting of two statements $t_1 \neq t_2$, containing at least one blank node.

Since the set of blank node identifiers is infinite, there are also infinitely many ways to describe the same bigraph. The actual blank node identifier that is used is only relevant when

Description	Bigraph	Overlap specification
One resource overlaps	`(_:A,p1,_:B).(_:A,p2,_:C)`	$p_1 \overleftarrow{\bowtie} p_2 \mapsto (1,2)$
	`(_:A,p1,_:C).(_:B,p2,_:C)`	$p_1 \overrightarrow{\bowtie} p_2 \mapsto (1,2)$
	`(_:A,p1,_:B).(_:B,p2,_:C)`	$p_1 \overline{\bowtie} p_2 \mapsto (1,2)$
	`(_:A,p1,_:B).(_:C,p2,_:A)`	$p_2 \overline{\bowtie} p_1 \mapsto (2,1)$
Two resources overlap	`(_:A,p1,_:B).(_:A,p2,_:B)`	$p_1 \overleftarrow{\bowtie} p_2 \mapsto (1,2), p_1 \overrightarrow{\bowtie} p_2 \mapsto (1,2)$
	`(_:A,p1,_:B).(_:B,p2,_:A)`	$p_1 \overline{\bowtie} p_2 \mapsto (1,2), p_2 \overline{\bowtie} p_1 \mapsto (2,1)$

Table 5.1: Overlap table types. Instances of the first bigraph statement (containing $p1$) are on data store s_1, while instances of the second bigraph statement are on s_2.

it is used more than once in the same graph: it then refers to the same blank node. The scope of a blank node identifier is limited to the bigraph. This means that the bigraph `(_:A,p1,_:B).(_:A,p2,_:C)` is equivalent with `(_:A,p1,_:C).(_:A,p2,_:B)`. As with regular RDF graphs, two bigraphs are equivalent if and only if there exists a bijection between the sets of blank node identifiers.

A statement with two blank node identifiers `(_:A, p, _:B)` is abbreviated to p_{AB}, or if no ambiguity arises, to p. For bigraphs, the following notation is used when the actual blank node identifiers are not relevant:

1. path-forming bigraph (object-subject join):
   `(_:A,p,_:B).(_:B,q,_:C)` is abbreviated to $p \overline{\bowtie} q$
2. branching by subjects (subjects join):
   `(_:A,p,_:B).(_:A,q,_:C)` is abbreviated to $p \overleftarrow{\bowtie} q$
3. branching by objects (objects join):
   `(_:A,p,_:B).(_:C,q,_:B)` is abbreviated to $p \overrightarrow{\bowtie} q$

Note that it is not necessary to define a subject-object join, as this is the same as an object-subject join with the operands reversed: `(_:A,p,_:B).(_:C,q,_:A)` is equal to `(_:C,q,_:A).(_:A,p,_:B)`, which is written as $q \overline{\bowtie} p$. The operators $\overleftarrow{\bowtie}$ and $\overrightarrow{\bowtie}$ are commutative, i.e. $a \overleftarrow{\bowtie} b = b \overleftarrow{\bowtie} a$. The operator $\overline{\bowtie}$ is not cummutative, because it indicates a path from left to right, which is not the same as a path from right to left.

5.1.2 Overlap specification

As was demonstrated in Section 4.2.3, the query answering process can be improved if the mediator is aware of the presence or absence of overlap in the distribution. An overlap specification is used to describe this knowledge.

Definition 5.2. Let $B = (t_1, t_2)$ be a bigraph, and s_1 and s_2 be two data store identifiers. Then $O = \{t_1 \mapsto s_1, t_2 \mapsto s_2\}$ is an *overlap specification*.

Suppose we take the instances of t_1 in s_1, and the instances of t_2 in s_2. There exists resource overlap if the merge of these instances will contain instances of B. In that case, overlap specification O is *valid*.

Example 5.1. *Consider the distribution of the running example in Figure 3.2. The distribution consists of graphs G_1, G_2 and G_3 which are mapped to data stores s_1, s_2, s_3 respectively. Consider bigraph $B =$* `(_:A, name, _:B).(_:A, paints, _:C)` *and overlap specification $O_1 = \{$*`(_:A, name, _:B)` $\mapsto$ s_2, `(_:A, paints, _:C)` $\mapsto s_1\}$.
Data store s_2 contains 3 instances of `(_:A, name, _:B)`*, and data store s_1 contains 3 instances of* `(_:A, paints, _:C)`*. The merge of these instances comprises 6 statements. From this merge, three instances of B can be formed:*

{ `(DaVinci, name, "Leonardo Da Vinci"), (DaVinci, paints, MonaLisa)` *}*
{ `(Rembrandt, name, "Rembrandt van Rijn"), (Rembrandt, paints, Nachtwacht)` *}*
{ `(Rembrandt, name, "Rembrandt van Rijn"), (Rembrandt, paints, Staalmeesters)` *}*

Now, consider another overlap specification: $O_2 =$ `{(_:A, name, _:B)` $\mapsto s_3$, `(_:A, paints, _:C)` $\mapsto s_1\}$*. Data store s_3 contains 1 instance for* `(_:A, name, _:B)`*, data store s_1 contains 3 instances for* `(_:A, paints, _:C)`*. The merge of these instances contains 4 statements. But this time, there are zero instances for B. This is the case because there is no resource for which blank node* `_:A` *can be instantiated. That is, there is no overlapping resource between* `(_:A, name, _:B)` *on s_3 and* `(_:A, paints, _:C)` *on s_1. Hence, this overlap specification is not valid.*

Let $B = (t_1, t_2)$ be a bigraph in which the subjects and objects of both statements are blank node identifiers. Since a bigraph is a connected graph, there must be at least one blank node identifier that is used in both statements. Let Y denote the set of shared blank node identifiers. Overlap specification $O = \{t_1 \mapsto s_1, t_2 \mapsto s_2\}$ for B is valid if and only if for every blank node identifier $y \in Y$, there exists a resource r that can be instantiated for y in both answer sets $A_{s_i}(t_1)$ and $A_{s_j}(t_2)$. That is, if

$$\Pi_Y(A_{s_i}(t_1)) \cap \Pi_Y(A_{s_j}(t_2)) \neq \emptyset \tag{5.1}$$

where Π and $\cap$ are the projection respectively intersection operators from relational algebra. The equation can be simplified by using a natural join operator: Overlap specification $O = \{t_1 \mapsto s_1, t_2 \mapsto s_2\}$ is valid if and only if:

$$A_{s_1}(t_1) \bowtie A_{s_2}(t_2) \neq \emptyset \tag{5.2}$$

Definition 5.3. An *overlap table* for bigraph B, $OT(B)$, is a set of overlap specifications that have B as their domain.

Hence, the overlap table for B is defined as follows:

$$OT(B) = \{(t_1 \mapsto s_i, t_2 \mapsto s_j) \mid (A(t_1, s_i) \bowtie A(t_2, s_j)) \neq \emptyset\} \tag{5.3}$$

An overlap specification can be abbreviated to a pair of data store identifiers (i, j) if no ambiguity arises. The set of overlap pairs for B in D is defined by $\{(i, j) \mid (t_1 \mapsto s_i, t_2 \mapsto s_j) \in OT(B)\}$. Due to the commutativity of some types of bigraphs, this can only be done

	(_:A, name, _:B)	(_:A, paints, _:C)
1	1	1
2	2	1
3	2	3

(a) T_{np}: Overlap table for (name $\overleftarrow{\bowtie}$ paints)

	(_:A, paints, _:B)	(_:B, exhibitedAt, _:C)
1	1	1
2	1	3

(b) T_{pe}: Overlap table for (paints $\overline{\bowtie}$ exhibitedAt)

	(_:A, exhibitedAt, _:B)	(_:B, country, _:C)
1	1	1
2	3	3

(c) T_{ec}: Overlap table for (exhibitedAt $\overline{\bowtie}$ country)

Table 5.2: Examples of overlap tables for the distribution of the running example in Figure 3.2. The numbers i in the tables themselves refer to the data stores s_i.

if it is clear which statement in the bigraph is mapped to which data store. For example, if $a \overleftarrow{\bowtie} b$ has overlap specification (i, j), then $b \overleftarrow{\bowtie} a$ has overlap specification (j, i).

If and only if $(\exists_{(i,j)\in OT(B)}\ i = j)$, there exists intra-store overlap for B in D; i.e. there are instances of the whole bigraph in the data store. If and only if $(\exists_{(i,j)\in OT(B)}\ i \neq j)$, there exists inter-store overlap for B in D.

Table 5.1 shows for the resource overlap types of Section 3.3 how they can be represented by overlap tables. Table 5.2 shows the feasible overlap specifications for a number of bigraphs that occur in the running example.

5.1.3 Replication

When statements are replicated over multiple data stores, redundancy may exist in an overlap table. More concretely, the instances that can be retrieved for one overlap specification, can also be retrieved for another overlap specification. This was described in Section 3.4.4.

Let $A(B, O)$ denote the set of instances of B according to overlap specification O. Overlap specification O is redundant if:

$$A(B, O) \subseteq A(B, O') \tag{5.4}$$

Consider bigraph B = (name $\overleftarrow{\bowtie}$ paints), for which the overlap is described in Table 5.2a. Consider the first two overlap specifications in the table: $O_1 = (1, 1)$ and $O_2 = (2, 1)$. The instance sets $A(B, O_1)$ and $A(B, O_2)$ are, in both cases:

```
{
{ (DaVinci, name, “Leonardo Da Vinci”), (DaVinci, paints, MonaLisa) },
{ (Rembrandt, name, “Rembrandt van Rijn”), (Rembrandt, paints, Nachtwacht) },
{ (Rembrandt, name, “Rembrandt van Rijn”), (Rembrandt, paints, Staalmeesters) }
}
```

Hence, the set of instances for O_1 is contained in the set of instances for O_2, and vice versa. This means that only one of these overlap specifications needs to be taken into account to obtain all instances for this bigraph. This annotation is added in the overlap table for the bigraph in Table 5.3.

r	`(_:A, name, _:B)`	`(_:A, paints, _:C)`	
1	1	1	$r_1 \subseteq r_2$
2	2	1	$r_2 \subseteq r_1$
3	2	3	

Table 5.3: T'_{np}: Overlap table for (name $\overleftarrow{\bowtie}$ paints), annotated with replication information.

5.1.4 Complexity

Let b be the number of different bigraphs that can be formed in a graph G. When only bigraphs of the form `(_:a, p, _:b)`, i.e. with blank nodes as subject and object, the number of different bigraphs is equal to the number of predicates in the graph. Let r be the average number of rows in an overlap table. An overlap table for a bigraph consists of 2 columns. The size complexity of the set of overlap tables for G is then: $\Theta(2 \cdot b \cdot r)$. Factor b depends on the structure of the graph (G) and the choice of how specific the bigraphs are chosen. Factor r depends on the distribution (D) of the data set.

5.1.5 Overlap table construction

One approach is to incrementally construct these overlap tables during regular query processing. Assume that during query processing, two source queries ${}^1[p_{AB}]$ and ${}^2[q_{AC}]$ are executed. Whenever the evaluation of ${}^1[p_{AB}] \bowtie {}^2[q_{AC}]$ is the empty answer set, the knowledge is obtained that for bigraph $p \overleftarrow{\bowtie} q$ the overlap specification (1,2) should not be used. In case the evaluation does not result in the empty answer set, the mediator knows that overlap (1,2) exists on this bigraph. The next time the mediator is given a query containing $p \overleftarrow{\bowtie} q$, the previously obtained knowledge can be utilized. Over time, this means that better query plans will be generated.

5.2 Initial plan generation

In this section it is researched how the index structure from Section 5.1 can be utilized to construct a query plan.

5.2.1 Graph isomorphism

An overlap table describes how the instances of a bigraph can be retrieved.

If we take two adjacent edges (triple patterns) from the query graph, we can check if there is an overlap table that uses a bigraph that is isomorph to these two edges. A bigraph is

isomorph with a graph pattern if the blank nodes of the bigraph can be mapped to the variables of the graph pattern.

Let V_B and V_Q be the sets of vertices in B respectively Q.

Definition 5.4. A bigraph B is *applicable* to graph pattern Q if and only if there is a mapping $M : V_B \rightarrow V_Q$ such that for all $l \in L$ (literals) and $u \in U$ (URIs): $M(l) = l$ and $M(u) = u$, and:

$$\langle \forall_{(m,p,m') \in B} (M(m), p, M(m')) \in Q \rangle$$

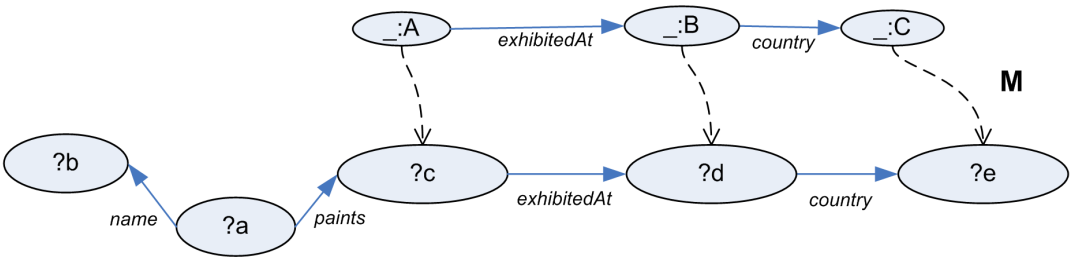


Figure 5.1: Query graph and mapping of an applicable bigraph

Example 5.2. *Consider query graph* $Q =$ `(?a, name, ?b).(?a, paints, ?c).(?c, exhibitedAt, ?d).(?d, country, ?e)}`
Bigraph (exhibitedAt $\overline{\bowtie}$ *country) is applicable to this query graph. This is shown graphically in Figure 5.1.*

Put differently, a bigraph B is applicable to a query graph Q if there exists a graph B' such that B' is a subgraph of Q and B' is isomorph to B.

Observe that the only difference between B' and B can be the vertices that represent a blank node in B' and a variable in B. In other words, B' must be an edge-induced subgraph of Q: "an edge-induced subgraph is a subset of the edges of a graph together with any vertices that are their endpoints."[1] The induced subgraph isomorphism problem is NP-complete [Epp94]. But since a bigraphs has at most 3 vertices, the time complexity of deciding if the bigraph matches an induced subgraph of Q is at most $O(n^3)$, where n is the number of edges in Q. In addition, if Q is planar, deciding if the bigraph matches an induced subgraph of Q can be decided in $O(n)$ [Epp94].

5.2.2 Source lookup table construction

The goal is now to construct a table for the input query that indicates which combinations of source queries result in a non-empty answer. This table will be called the Source Lookup Table (SLT), and is a generalization of the overlap table.

[1] Copes, L. "Subgraphs.", `http://web.hamline.edu/~lcopes/SciMathMN/concepts/csubgr.html`

r	name_{ab}	paints_{ac}
1	1	1
2	2	1
3	2	3

(a) T'_{np}

r	paints_{ac}	exhibitedAt_{cd}
1	1	1
2	1	3

(b) T'_{pe}

r	exhibitedAt_{cd}	country_{de}
1	1	1
2	3	3

(c) T'_{ec}

Table 5.4: Source lookup tables based on the overlap tables from Table 5.2.

The SLT consists of a list of complete mappings $r : Q \to S$, where Q is a set of triple patterns, and S is a set of data stores. Each row in the SLT represents a procedure for obtaining answers to the query.

r	name_{ab}	paints_{ac}	exhibitedAt_{cd}
1	1	1	1
2	2	1	1
3	1	1	3
4	2	1	3

Table 5.5: T'_{npe}: Source lookup table that is constructed by $T'_{np} \bowtie T'_{pe}$

r	name_{ab}	paints_{ac}	exhibitedAt_{cd}	country_{de}
1	1	1	1	1
2	2	1	1	1
3	1	1	3	3
4	2	1	3	3

Table 5.6: T'_{npec}: Source lookup table for query graph depicted in Figure 5.1, constructed by $T'_{npe} \bowtie T'_{ec}$

The procedure to construct an SLT for query graph $Q = (V, E)$ is then as follows. Let E' initially be the set of edges in the query graph, i.e. $E' := E$. Pick an edge e_1 from E' and an adjacent edge e_2 from E. Find the overlap table for the bigraph that is equivalent with the graph consisting of the edges (e_1, e_2). Add this overlap table to the result table by performing a natural join between the two tables. Remove e_1 from E'. Repeat until E' is empty.

Example 5.3. *Consider the query graph from the previous example, and shown in Figure 5.1. Tables 5.4a, 5.4b, and 5.4c present the SLTs for every pair of adjacent edges in the query graph. These SLTs are based on the overlap tables in Table 5.2, and are constructed as follows. Map the blank node identifiers in the bigraph to the query graph's variables, according to the mappings M that were*

used to choose the bigraph. For overlap table T_{np} this means that mapping M is $\{\_:\texttt{A} \mapsto ?a, \_:\texttt{B} \mapsto ?b, \_:\texttt{C} \mapsto ?c\}$. Let $T'(T)$ denote overlap table T in which the statements of the overlap specification are replaced according to M.

Perform a natural join on all selected overlap tables:

$$T'_{npec} = T'_{np} \bowtie T'_{pe} \bowtie T'_{ec}$$

The first join yields the SLT shown in Table 5.5. The final SLT is shown in Table 5.6.

Let n be the number of edges in query graph Q. Finding a matching bigraph is $O(b \cdot n^3)$ when there are b overlap tables available. Performing the join between two SLTs takes $O(r^2)$ when r is the number of rows in each SLT. The total time complexity to construct an SLT for Q is then $O(n^4 \cdot b \cdot r)$.

5.2.3 Query plan construction

Pick a row from the constructed SLT. Each edge in the query graph can now be annotated with the corresponding data store in the row. For adjacent edges that are annotated with the same data store (intra-store overlap) a larger source query can be sent to that data store. The answers to source queries to different data stores are combined (joined) at the mediator. The complete answer set is obtained by performing a union over the answers obtained from each row.

Observe that the answer is constructed by performing a union of several correct answer tuples. It is, in other words, a "union of joins", instead of a "join of unions" as the earlier approaches were. This means that already during the query processing phase, answer tuples can be delivered back to the user.

For r rows in the SLT and n edges in the query graph, the translation of the SLT to a query plan can be performed in $\Theta(r \cdot n)$ time. This follows from the description of the translation above.

Example 5.4. *The SLT in Table 5.6 can be directly translated into the following source queries:*

$$A(Q) = \cup \begin{cases} {}^{1}[\mathit{name} \bowtie \mathit{paints} \bowtie \mathit{exhibitedAt} \bowtie \mathit{country}] \\ {}^{2}[\mathit{name}] \bowtie {}^{1}[\mathit{paints} \bowtie \mathit{exhibitedAt} \bowtie \mathit{country}] \\ {}^{1}[\mathit{name} \bowtie \mathit{paints}] \bowtie {}^{3}[\mathit{exhibitedAt} \bowtie \mathit{country}] \\ {}^{2}[\mathit{name}] \bowtie {}^{1}[\mathit{paints}] \bowtie {}^{3}[\mathit{exhibitedAt} \bowtie \mathit{country}] \end{cases} \tag{5.5}$$

This query plan consists of 6 distinct source queries. Evaluation of this query plan yields the answer shown in Table 5.8.

a	b	c	d	e
Rembrandt	"Rembrandt van Rijn"	Nachtwacht	Rijksmuseum	Netherlands

(a) Result of source query ${}^{1}[n_{ac} \bowtie p_{ac} \bowtie e_{cd} \bowtie c_{de}]$

a	c	d	e
Rembrandt	Nachtwacht	Rijksmuseum	Netherlands

(b) Result of source query ${}^{1}[p_{ac} \bowtie e_{cd} \bowtie c_{de}]$

a	b
Rembrandt	"Rembrandt van Rijn"
DaVinci	"Leonardo Da Vinci"
VanGogh	"Vincent van Gogh"

(c) Result of source query ${}^{2}[n_{ab}]$

a	b	c
Rembrandt	"Rembrandt van Rijn"	Nachtwacht
DaVinci	"Leonardo da Vinci"	MonaLisa

(d) Result of source query ${}^{1}[n_{ab} \bowtie p_{ac}]$

a	c
Rembrandt	Nachtwacht
DaVinci	MonaLisa

(e) Result of source query ${}^{1}[p_{ac}]$

c	d	e
MonaLisa	Louvre	France
Oneness	GroningerMuseum	Netherlands

(f) Result of source query ${}^{3}[e_{cd} \bowtie c_{de}]$

Table 5.7: Tabular representations of the results of the six source queries described by Equation (5.5)

b	a	c	d	e
"Rembrandt van Rijn"	Rembrandt	Nachtwacht	Rijksmuseum	Netherlands
"Leonardo da Vinci"	DaVinci	MonaLisa	Louvre	France
"Leonardo da Vinci"	DaVinci	MonaLisa	Louvre	France
"Rembrandt van Rijn"	Rembrandt	Nachtwacht	Rijksmuseum	Netherlands

Table 5.8: Tabular representation of the answer described by Equation (5.5).

5.3 Transforming plans

The query plan that is generated by the procedure described in Section 5.2 may not be optimal. In particular, the low costs in the cost model for re-used source queries are an opportunity in the query optimization process. In this section, rewrite rules on the query plan are investigated. The answer set that is the result of the execution of a rewritten query plan should be the same as for the original plan. Equation (5.5) is used as running example query plan.

5.3.1 Transformations from relational algebra

Since the answer sets for a query plan expression are modelled in the same way as relations in relational algebra, many of the rules of the relational algebra operators can be applied in this case as well. A few examples are listed here. For a more thorough list, see e.g. [SKS02, Sec. 14.3].

The natural join operator is associative and commutative:

$$(P \bowtie Q) \bowtie R \Leftrightarrow P \bowtie (Q \bowtie R) \tag{5.6}$$

$$P \bowtie Q \Leftrightarrow Q \bowtie P \tag{5.7}$$

The projection operator distributes over the natural join operator as follows. Let V_P and V_Q denote the set of variables in P and Q respectively. Let $L_1 \subseteq V_P$ and $L_2 \subseteq V_Q$.

$$\Pi_{L_1 \cup L_2}(P \bowtie Q) \Leftrightarrow \Pi_{L_1}(P) \bowtie \Pi_{L_2}(Q) \text{ (join-proj-distributive)} \tag{5.8}$$

5.3.2 Transformations in a distributed environment

If a join or union operator are used on two source queries to the same data store, then the source queries may be combined by pushing the operator "inside" the source query. Hence, this causes the operator to be evaluated on the data store itself. Generally, this reduces the total amount of data that will be received from the data store, because only relevant tuples will be returned.

The notation $R \vdash R'$ is used to describe the rewrite step of replacing all occurences of R by R' in the query plan expression.

$$^x[\alpha] \bowtie {}^x[\beta] \vdash {}^x[\alpha \bowtie \beta] \text{ (remote-join-eval)} \tag{5.9}$$

$$^x[\alpha] \cup {}^x[\beta] \vdash {}^x[\alpha \cup \beta] \text{ (remote-union-eval)} \tag{5.10}$$

In some cases, it may be beneficial to apply rule (remote-join-eval) in the other direction:

$$^x[\alpha \bowtie \beta] \vdash {}^x[\alpha] \bowtie {}^x[\beta] \text{ (local-join-eval)} \tag{5.11}$$

I.e. replace a single source query by two source queries and execute the join operation on the mediator instead of at a data store. Normally, this would result in an increase in the amount of irrelevant data that is returned by a data store. But if the two new source queries are already part of the whole query plan, it may be possible to re-use their answers. This technique was described in more detail in Section 4.1.3.

Example 5.5. *The source query on the first line of the query plan in Equation* (5.5) *can be replaced by two queries that both already occur in the query plan. Apply rule (local-join-eval) for* $\alpha = \textit{name} \bowtie \textit{paints}$ *and* $\beta = \textit{paints} \bowtie \textit{exhibitedAt} \bowtie \textit{country}$ *to obtain* ${}^{1}[\alpha] \bowtie {}^{1}[\beta]$*. The rewritten query plan becomes:*

$$A(Q) = \cup \begin{cases} {}^{1}[\textit{name} \bowtie \textit{paints}] \bowtie {}^{1}[\textit{paints} \bowtie \textit{exhibitedAt} \bowtie \textit{country}] \\ {}^{2}[\textit{name}] \bowtie {}^{1}[\textit{paints} \bowtie \textit{exhibitedAt} \bowtie \textit{country}] \\ {}^{1}[\textit{name} \bowtie \textit{paints}] \bowtie {}^{3}[\textit{exhibitedAt} \bowtie \textit{country}] \\ {}^{2}[\textit{name}] \bowtie {}^{1}[\textit{paints}] \bowtie {}^{3}[\textit{exhibitedAt} \bowtie \textit{country}] \end{cases} \tag{5.12}$$

This reduces the number of distinct source queries by one, resulting in 5 distinct source queries.

In some cases multiple data stores can be used to evaluate a part of the query graph. This was demonstrated for the case of replication in Section 5.1.3: overlap specification O_1 can be replaced by overlap specification O_2 if the evaluation by means of O_1 delivers all instances that O_2 would also deliver. In the generic case, the predicate *alternative-source*(α, x, y) is true if and only if the evaluation of query α on x can be replaced by the evaluation of α on y, without losing correctness or completeness. This corresponds to the following rule:

$$\frac{\textit{alternative-source}(\alpha, x, y)}{{}^{x}[\alpha] \vdash {}^{y}[\alpha]} \text{ (source-swap)} \tag{5.13}$$

Projection equivalents

In another scenario, it is sometimes possible to perform a larger source query and do a projection afterwards. As explained above, this can be useful if the larger source query already occurs in the plan, and an additional source query does not contribute to the cost.

When inter-store overlap exists for a bigraph, the evaluation of the corresponding query graph fragment results in two source queries. In case both inter-store overlap (x, y), for $x \neq y$, and intra-store overlap (x, x) exists, at least two source queries are sent to data store x: one containing only the first query graph statement, and the second one containing both query graph statements.

For example, Table 5.2 indicates that (name $\overleftarrow{\bowtie}$ paints) has inter-store overlap (2,1) and intra-store overlap (1,1). A complete answer for this graph pattern can be obtained through

$$({}^{2}[\textit{name}_{ab}] \bowtie {}^{1}[\textit{paints}_{ac}]) \cup {}^{1}[\textit{name}_{ab} \bowtie \textit{paints}_{ac}]$$

Thus, three source queries are sent, two of which to source 1. However, when the graph pattern of a larger source query is vertically complete, source queries that use a subgraph

of this graph pattern are redundant. This follows from Proposition 4.2 in Section 4.2.3. In this case, ${}^1[name_{ab} \bowtie paints_{ac}]$ uses graph pattern that is vertically complete for data store 1. Source query ${}^1[paints_{ac}]$ can therefore be replaced by a projection on the answer to the larger source query:

$$({}^2[name_{ab}] \bowtie \Pi_{a,c}({}^1[name_{ab} \bowtie paints_{ac}])) \cup {}^1[name_{ab} \bowtie paints_{ac}]$$

Definition 5.5. Query plan $P \bowtie Q$ is *vertically complete* for data store x, denoted $vc(x, P \bowtie Q)$, if and only if:

$$A({}^x[P]) = \Pi_{V_P}(A({}^x[P \bowtie Q])), \text{ and}$$
$$A({}^x[Q]) = \Pi_{V_Q}(A({}^x[P \bowtie Q]))$$

where V_P and V_Q denote the variables that occur in expression P respectively Q. As was demonstrated in Section 3.4.4, vertical completeness is orthogonal to overlap. A data store can be vertically complete for a query plan that has overlap, as in the previous example. Or it can have no overlap with any other store, yet also be locally incomplete.

When vertical completeness holds for a join expression, a projection can be performed to either the left-hand-side or the right-hand-side:

$$\frac{vc(x, P \bowtie Q)}{{}^x[P] \vdash \Pi_{V_P}({}^x[P \bowtie Q]} \text{ (proj-intro-1)} \tag{5.14}$$

$$\frac{vc(x, P \bowtie Q)}{{}^x[Q] \vdash \Pi_{V_Q}({}^x[P \bowtie Q]} \text{ (proj-intro-2)} \tag{5.15}$$

Example 5.6. *Using rule (proj-intro-2) on $vc(1, name_{ab} \bowtie paints_{ac})$ results in the following equivalence for data store s_1:*

$$\Pi_{a,c}(A({}^1[name_{ab} \bowtie paints_{ac}])) = A({}^1[paints_{ac}]) \tag{5.16}$$

This equivalence can be used to remove the source query ${}^1[paints]$ from the query plan:

$$A(Q) = \cup \begin{cases} {}^1[name \bowtie paints] \bowtie {}^1[paints \bowtie exhibitedAt \bowtie country] \\ {}^2[name] \bowtie {}^1[paints \bowtie exhibitedAt \bowtie country] \\ {}^1[name \bowtie paints] \bowtie {}^3[exhibitedAt \bowtie country] \\ {}^2[name] \bowtie \Pi_{a,c}({}^1[name_{ab} \bowtie paints_{ac}]) \bowtie {}^3[exhibitedAt \bowtie country] \end{cases} \tag{5.17}$$

This results in another reduction by one of the number of distinct source queries. In this query plan, there are 4 distinct source queries.

5.4 Search strategy

Many of the local search algorithms available in the literature can be adapted to work within the PbR framework [Amb99]. The characteristics of the planning domain, the initial plan generator, and the rewriting rules determine which algorithm performs best. Local search methods include:

1. First improvement: rewritten plans are generated incrementally and the first plan with a lower cost than the current one is selected.
2. Best improvement: all possible rewritings are calculated, and the best (i.e. lowest cost) plan is selected.
3. Simulated annealing: a set of rewritten plans is calculated, and one plan is randomly randomly selected. If it has a lower cost, it is chosen as next step. If it has a higher cost is still chosen, with some probability. This probability is decreased as the algorithm progresses.

It is expected that some transformation rules result in a higher plan cost, such as the (local-join-eval) rule. Such a step may be necessary however to make other rules applicable to the plan. E.g. the two projection rules may only be applicable after the (local-join-eval) rule has been executed. Hence, it is expected that an algorithm that selects only plans with a lower cost than the current one is inappropriate. Therefore, the simulated annealing algorithm may be a good candiate as search strategy for this optimization problem.

5.5 Example

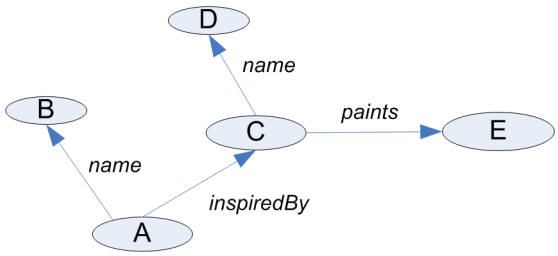


Figure 5.2: Example query

Consider the query graph `(?a, name, ?b).(?a, inspiredBy, ?c).(?c, name, ?d).(?c, paints, ?e)`, graphically depicted in Figure 5.2. The following bigraphs are applicable to this query graph:

1. name $\overleftarrow{\bowtie}$ inspiredBy
2. name $\overleftarrow{\bowtie}$ paints

r	name_{ab}	inspiredBy_{ac}
1	2	2

(a) T_{ni}: SLT for (name $\overleftarrow{\bowtie}$ inspiredBy)

r	inspiredBy_{ac}	name_{cd}
1	2	2
2	2	1

(b) T_{in}: SLT for (inspiredBy $\overleftarrow{\bowtie}$ name)

r	name_{cd}	paints_{de}
1	1	1
2	2	1

(c) T_{np}: SLT for (name $\overleftarrow{\bowtie}$ paints)

r	name_{ab}	inspiredBy_{ac}	name_{cd}	paints_{de}
1	2	2	2	1
2	2	2	1	1

(d) T_{ninp}: SLT for the query graph depicted in Figure 5.2

Table 5.9: Overlap tables for three bigraphs, and the source lookup table that results from joining these tables

3. inspiredBy $\overline{\bowtie}$ name
4. inspiredBy $\overline{\bowtie}$ paints

The SLTs T_{ni}, T_{in}, T_{np} are shown in Table 5.9. The SLT for the whole query graph T_{ninp} is constructed from these tables and is shown in Table 5.9d. The construction of the answer for the query graph can now be directly translated from Table 5.9d, as:

$$A(Q) = \cup \begin{cases} {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}] \bowtie {}^{1}[\text{paints}_{ce}] \\ {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac}] \bowtie {}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}] \end{cases} \tag{5.18}$$

Now the transformation rules are applied together with the additional information about vertical completeness.

$$\begin{aligned} A(Q) =& \{\textit{Apply (proj-intro-2), together with vc}(1, \text{name}_{AB} \bowtie \text{paints}_{AC}) \textit{ to line 1}\} \\ & \{{}^{1}[\text{paints}_{ce}] \vdash \Pi_{c,e}({}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}])\} \\ & \cup \begin{cases} {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}] \bowtie \Pi_{c,e}({}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}]) \\ {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac}] \bowtie {}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}] \end{cases} \\ =& \{\textit{Apply (local-join-eval) to line 2}\} \\ & \cup \begin{cases} {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}] \bowtie \Pi_{c,e}({}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}]) \\ {}^{2}[\text{name}_{ab}] \bowtie {}^{2}[\text{inspiredBy}_{ac}] \bowtie {}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}] \end{cases} \end{aligned}$$

a	b	c	d
VanGogh	"Vincent van Gogh"	Rembrandt	"Rembrandt"

(a) Result of source query ${}^{2}[n_{AB} \bowtie i_{AC} \bowtie n_{CD}]$

c	d	e
Rembrandt	"Rembrandt van Rijn"	NachtWacht
DaVinci	"Leonardo Da Vinci"	MonaLisa

(b) Result of source query ${}^{1}[n_{CD} \bowtie p_{CE}]$

a	b	c	d	e
VanGogh	"Vincent van Gogh"	Rembrandt	"Rembrandt"	NachtWacht
VanGogh	"Vincent van Gogh"	Rembrandt	"Rembrandt van Rijn"	NachtWacht

(c) Tabular representation of the answer

Table 5.10: Tabular representations of the results of the queries described by the final query plan

$$
\begin{aligned}
&= \{\textit{Apply (proj-intro-1), together with vc}(2, \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}) \textit{ to line 2}\} \\
&\quad \{{}^{2}[\text{inspiredBy}_{ac}] \vdash \Pi_{a,c}({}^{2}[\text{inspiredBy}_{ac} \bowtie \text{name}_{cd}])\} \\
&\quad \cup \begin{cases} {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}] \bowtie \Pi_{c,e}({}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}]) \\ {}^{2}[\text{name}_{ab}] \bowtie \Pi_{a,c}({}^{2}[\text{inspiredBy}_{ac} \bowtie \text{name}_{cd}]) \bowtie {}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}] \end{cases} \\
&= \{\textit{Apply (join-proj-distributive) to line 2}\} \\
&\quad \cup \begin{cases} {}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}] \bowtie \Pi_{c,e}({}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}]) \\ \Pi_{a,b,c}({}^{2}[\text{name}_{ab} \bowtie \text{inspiredBy}_{ac} \bowtie \text{name}_{cd}]) \bowtie {}^{1}[\text{name}_{cd} \bowtie \text{paints}_{ce}] \end{cases}
\end{aligned}
$$

The answers to these source queries, as well as the final answer to the input query are shown in Table 5.10.

5.6 Conclusion

An index structure was defined for: 1) intra- and inter-store overlap of two triple patterns, and 2) vertical completeness of two triple patterns in a data store.

A query processing method (the overlap-based method) that uses this index structure was introduced. The query plans generated by this method support the fast production of first answers in many situations. Transformations of the initial plan reduce the number of source queries by applying the index information about vertical completeness.

Chapter 6

Prototype evaluation

In the previous chapter, a new query processing approach was discussed. Only very small data sets (databases) were used to describe the appraoch. In this chapter, we wish to evaluate the approach on larger distributed data sets, and compare it with other query planning approaches. To do so, three query planners are implemented: the naive query planner (Section 4.2.1), the property-based query planner (Section 4.2.2), and the new overlap-based planner (Chapter 5).
The approaches will be evaluated on the basis of the cost of the generated query plans. As was discussed in Section 4.1, cost is a model for the estimated time that is required to compute the answer to the input query. Two different cost metrics are distinguished:

1. cost to return the complete answer set;
2. cost to return the first answer tuple.

Note that the query plans are not actually executed; only the costs of each approach are compared. Not all ideas from Chapter 5 are implemented in the current prototype, including replication in the index structure, and the automated optimization of the query plan by rewriting parts of it.
The following three parameters influence the cost of the query plan produced by a query planner:

1. The data set. The size of the data set (in number of statements), and the structure of the graph that it represents.
2. The distribution of the data set over data stores. The size of the data set in each data store, and which fragments of the data set are contained in each data store.
3. The input query. The size and structure of the query graph.

In Section 6.1 an overview of the implementation of the components of the prototype is given. Section 6.2 describes the implementation of the index structure for the overlap-based

approach, and in Section 6.3 the implementation of query planners is discussed. Section 6.4 gives details about the experiments that will be performed with the help of the prototype, and Section 6.5 presents the results of these experiments.

6.1 Implementation overview

The prototype is implemented in Java 6.0.

6.1.1 Query input

The tool ANTLR 3[1] is used to construct a query parser. ANTLR is a parser generator tool and allows one to define language grammars. A very simple query language grammar is used: the input query is a set of triple patterns. These triple patterns represent a query graph, in the way that is described in Section 2.2.

All subjects and objects are interpreted as variables, and all properties as URIs. Hence, selection of values by specifying literals or URIs as the subject or object is not possible. Also, no queries with a variable as property can be used.

The JGraphT[2] library is used to represent the query graph. This library directly supports labeled, directed graphs, and also provides various graph walking algorithms. These algorithms are useful for the construction of the source lookup table and the query planning algorithm that were discussed in the previous chapter.

A query graph uses String objects as vertices, and JoinGraphEdge objects as edges. A JoinGraphEdge has a label, which is also a String object. Such an edge represents an RDF triple, where the source vertex of the edge is the subject, the target vertex is the object, and the label is the property of the triple.

In pseudo-code:

```
JoinGraphEdge:  ( String source, String label, String target )
```

6.2 Implementation of the index structure

6.2.1 Overlap tables

Overlap tables and bigraphs were defined in Section 5.1. An overlap table is a list of mappings from a graph edge to a data store. A data store's identifier is a String. In pseudo-code:

```
DataStore:  ( String )
SLTRow:  ( Map<JoinGraphEdge, DataStore> )
SourceLookupTable:  ( List<SLTRow> )
```

[1] http://www.antlr.org/
[2] http://jgrapht.sourceforge.net/

An overlap table can be retrieved by specifying the bigraph that it represents. This is implemented as `OverlapTableMapping`: This class should take the commutativity property of bigraphs into account when looking up the SLT given a bigraph.

```
JoinType:  [ SubjectSubject, ObjectSubject, ObjectObject ]
Bigraph:  ( String property1, String property2, JoinType jointype )
OverlapTableMapping:  ( Map<Bigraph, SourceLookupTable> )
```

6.2.2 Cardinality tables

A cardinality table for a data store contains the number of occurences for each property. It is used in the cost model to estimate join sizes. In addition, this index is also used by the property-based planner, to determine which data stores have instances for a certain property.

```
CardinalityIndex:  ( Map<DataStore, Map<String, Integer> > )
```

6.2.3 Index construction

The indexing process consists of two parts: construction of overlap tables, and the construction of cardinality lists for property names. The results of this process are stored for later use.

To simulate the use of multiple data stores, the notion of *context* in a Sesame 2 repository is used. Contexts provide a way to group sets of statements together through a single group identifier. In this case, each statement in a data store is given the name of that data store as context identifier. The use of contexts in this way provides a more direct way to extract the data to build the overlap tables. The use of contexts for this purpose is clearly artificial: in practice, data stores are separate, and other ways must be persued to obtain the necessary index information.

With the help of contexts, it is possible to use only three queries to obtain the information from all available data stores. For example, overlap caused by Subject-Subject joins can be determined by executing the following query:

```
select distinct s1,p1,s2,p2
from context s1 {a} p1 {b}
from context s2 {a} p2 {c}
where p1 != p2
```

For the determination of Object-Subject and Object-Object joins, the query after "`from context s2`" is changed to `{b} p2 {c}` and `{c} p2 {b}` respectively. The construction of these tables is performed in the `OverlapTableBuilder` class.

For the cardinality index, the list of properties in each context must be retrieved. This is done with the following query:

```
select s,p
from context s {a} p {b}
```

The construction of the cardinality table is done in the `CardinalityTableBuilder` class.

6.3 Implementation of the query planners

Let S denote the set of data stores that are used in the distribution.

6.3.1 Plan tree representation

The plan tree that is constructed is a rooted binary tree. The reason for this is that the cost and cardinality estimates for the query plan operators are defined for at most two operands, which corresponds to a left and right subtree of a node in the query plan tree.

```
PlanTree:
( org.jgrapht.graph.DefaultDirectedGraph<PlanNode, PlanEdge> )
```

A `PlanNode` is an abstract class, that has the following (instantiateable) subclasses:

1. `JoinPlanNode`, representing $P \bowtie Q$. This node has exactly two child nodes.
2. `UnionPlanNode`, representing $P \cup Q$. This node has exactly two child nodes.
3. `SQPlanNode`, representing: ${}^{x}[Q]$, where x is a data store identifier. This node has exactly one child node.
4. `ProjectionPlanNode`, representing: $\Pi_L(Q)$, where L is a set of variable names. This node has exactly one child node.
5. `LeafPlanNode`, representing p_{AB}. This node has exactly zero child nodes.

Every node object is able to return the total cost, the cost to retrieve the first answer, and the cardinality estimate of the subtree of which it is the root node.

6.3.2 Planner 1: naive planner

The naive planner uses no index structure. Based on the query graph that is given as input, a query plan is constructed.

6.3.3 Planner 2: property-based planner

The property-based planner uses the cardinality index to determine which data stores have at least one instance of a property.

6.3.4 Planner 3: overlap-based planner

For the overlap-based planner, first a source lookup table (SLT) is constructed. The procedure to construct such a table is based on the input query and the available overlap tables. Hereafter, an initial query plan is constructed. These procedures are illustrated with a few key algorithms here. The complete procedure is described in Section 5.2.

Algorithm 1: FindSources: construction of the source lookup table for a query graph

Input: Query graph $G = (V, E)$
Result: Source lookup table SLT for G

```
1  SLT := [];
2  E' := E;
3  while E' ≠ ∅ do
4      Pick edge e ∈ E';
5      A := Adj(E, e);
6      lookupSucceeded := false;
7      while ¬lookupSucceeded ∧ (A ≠ ∅) do
8          Pick edge e' ∈ A;
9          g := ({s(e), t(e), s(e'), t(e')}, {e, e'});
10         g' := FindIsomorphBigraph(g);
11         ot := FindOverlapTable(g');
12         lookupSucceeded := (ot ≠⊥);
13         A := A \ {e'};
14     end
15     ot' := MapToQueryEdges(ot, g', g);
16     E' := E' \ {e};
17     JoinSLT(SLT, ot');
18 end
19 return SLT;
```

FindSources (Algorithm 1) is used to construct an SLT for a query graph Q. The predicate Adj(E, e) returns a set of edges in E that are adjacent to edge e. FindIsomorphBigraph(g) returns a bigraph g' in the index structure that is isomorph to g. If no such bigraph exists in the index structure, $\perp$ is returned instead. FindOverlapTable(g') returns the overlap table for bigraph g', according to OverlapTableMapping. The predicate MapToQueryEdges(ot, g', g) replaces the edges $e \in g'$ in the domain of the overlap table ot to isomorph edges in g. Hereafter, the overlap table is merged with the constructed SLT by JoinSLT($slt1, slt2$), which performs a natural join on two SLTs, and returns the result.

MakePlan (Algorithm 2) traverses each row in the SLT, and adds a subtree of joins for each data store that is mentioned on that row. This is done by MakePlanTree (Algorithm 3). Hereafter, these subtrees are combined by union operators.

The function insertIntoPlan(P, e, op) adds a triple pattern e as a leaf plan node to plan tree P.

Algorithm 2: OverlapBasedQueryPlanner.MakePlan

Input: Query graph $G = (V, E)$; source lookup table SLT; set of data stores S
Result: Query plan P

```
1  P := [];
2  for r ∈ SLT do
3      PRow := [];
4      for sn ∈ S do
5          G' := "subgraph of G containing only edges that, according to r, map to data
              store sn";
6          p := MakePlanTree(G');
7          merge(Prow, p, ⋈);
8      end
9      merge(P, Prow, ∪);
10 end
11 return P;
```

Algorithm 3: OverlapBasedQueryPlanner.MakePlanTree

Input: Graph $G = (V, E)$ (expected to be a graph annotated with single data store)
Result: Plan tree for G

```
1  for v ∈ V do
2      for e ∈ E ∧ t(e) = v do
3          insertIntoPlan(P, e, ⋈);
4      end
5  end
6  return P;
```

6.3.5 Cost calculation

The cost for a plan tree as defined in Section 4.1 is recursively calculated by the tree node objects, and also stored within these tree node objects. This same holds for the cost to compute the first answer. The only exception on this is the cost for a source query, ${}^{x}[Q]$. After the regular cost calculation, a procedure is run that subtracts the costs of duplicate source queries from the plan. Because it is run afterwards, the nil cost of a duplicate source query does not render the first answer costs incorrect.

In general, a query planner may generate equivalent query plans that differ only in the order of join operations. These plans may have different costs. These plans are equivalent because the join operator is associative. But all the plan construction algorithms are implemented in such a fashion that the same query plan is generated, whenever the index structure and input query are kept unchanged. By having this guarantee, comparison of the cost results is possible.

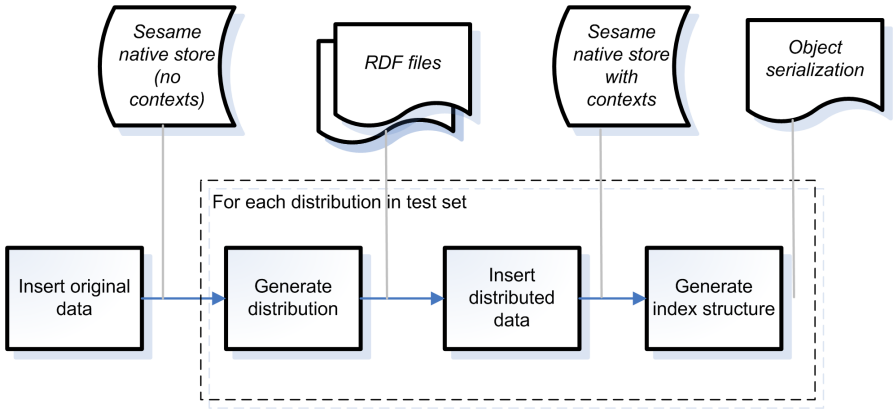


Figure 6.1: Phase 1 in the test process: generation of the index structure. Above the process flow, the output result of each process step is depicted.

6.4 Experiment set-up

The experiments are separated in two phases. Phase 1 consists of the preparation of the data set, the distributions of this data set, and the generation of the index structures that are used by the query planners in phase 2. The steps in this phase are shown in Figure 6.1. In phase 2 the query planners are evaluated. For each distribution that was prepared, a number of input queries is supplied to the planners. Each planner generates a query plan for which the cost is determined. The analysis will be performed on the these cost results. The process is outlined in Figure 6.2.

6.4.1 Data set

Sesame 2 Release Candidate 2[3] is used as back-end RDF storage system. A native repository is used for all experiments. A native repository is a repository that uses an on-disk data structure. The alternative, a memory-based repository requires all data in the repository to be fit in the memory of the Java Virtual Machine, which is problematic for larger data sets.

The data set on which the experiments are conducted is the merge of two repositories, called the Jamendo repostory and the Magnatune repostory. Both repostories contain a large collection of music artists, records and performances. The RDF representations of these repositories are obtained from DBTune[4]. This data set is chosen because 1) it is considerable in size (the Jamendo repository is 86,600 kilobytes in RDF/XML representation, 1,044,941 triples), 2) it has a manageable number of properties (24), and 3) distributions of the data set can be easily formulated and understood, since the concepts of artists, records etc. are known to everyone.

The ontology that is used for both repositories is the same, i.e. in both repositories the same set of properties is used. Figure 6.3 shows how the properties are used in the data set. The

[3] http://openrdf.org/
[4] http://dbtune.org/

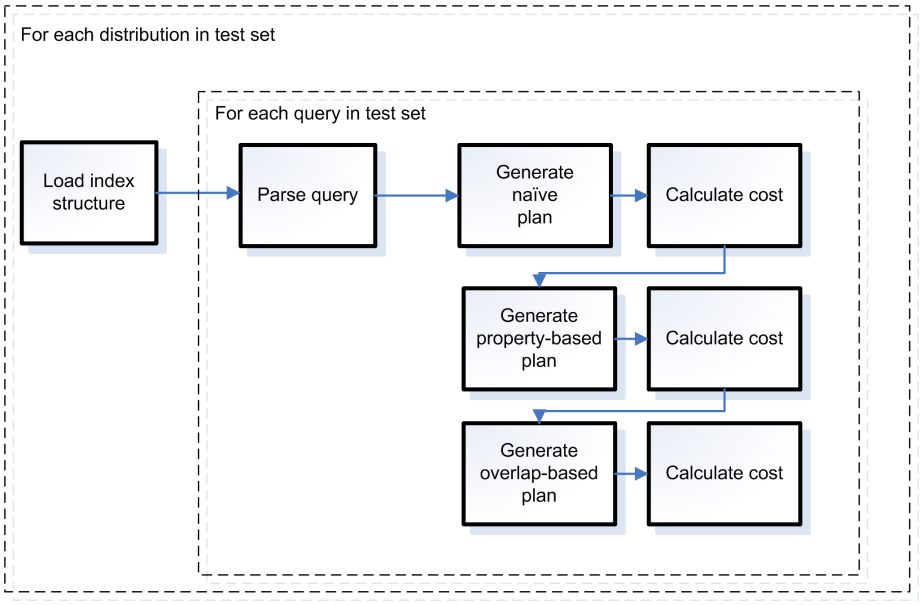


Figure 6.2: Phase 2 in the test process: generation of query plans.

actual data contained in the repositories is different. The Magnatune repository contains information about artists and records from the Magnatune music label. The Jamendo repository does not contain any information about artists or records from this music label.

Two modifications are made to the data from the Jamendo repository:

1. All foaf:homepage elements are removed, since they often referred to malformed URIs.
2. Some tags:taggedWithTag properties are added to artists. In the original data set, these properties were only used on records.

6.4.2 Distribution

An important characteristic of a distribution is the number of data stores participating in it. Besides this, each data store is characterized by the amount of data from the original data set it contains.

A data store is defined as a subgraph of the global graph. A subgraph can be extracted with a CONSTRUCT query in the SeRQL query language. The execution of a CONSTRUCT query results in a set of RDF statements. This set is stored so it can be loaded in phase 2 of the experiment.

In actual query processing systems, it is common to issue queries that contain selection operators. As a result, the cardinality of the answer that a data store returns is possibly much smaller than in queries without such selection operators. It is interesting to research the query processing performance under different answer set sizes. In the prototype system, this cannot be done in the input query, since no selection operators are supported. Instead,

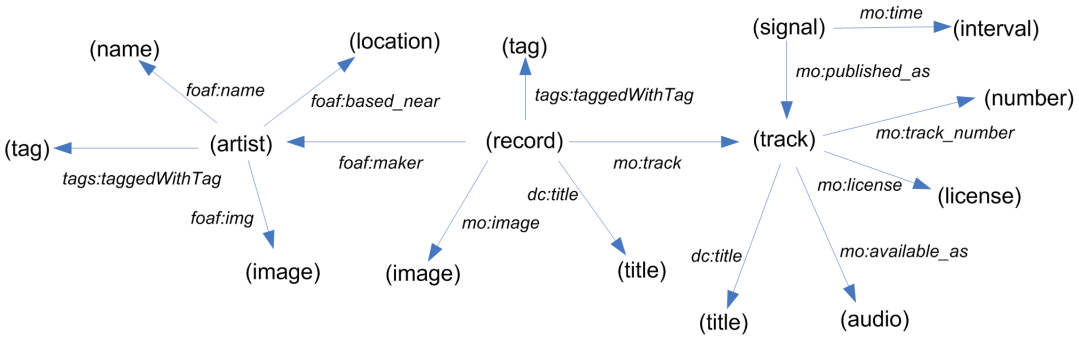


Figure 6.3: Graph showing how properties are used in the data set. The cardinality of the relationship is not depicted; many properties have many-to-many relationships. The nodes in the graph have been given names (between brackets) for clarity. These names do not occur in the data set itself. Not all properties are depicted in the graph. In particular, `rdf:type` properties are left out.

a second variant of each distribution is generated: one that has an artifically reduced size. This is done by supplying a LIMIT clause to the CONSTRUCT queries.

	Data store	# Sts A	# Sts B	# Predicates
1	artists	104	10548	5
2	records	19	2108	4
3	obscure	32	32	6
4	magna	24	551	6

Table 6.1: Overview of the distributions used in the experiment. For each data store, the number of statements are shown for distributions A and B. (Only data stores 1 and 2 are used for distribution 1A and 1B.) On the right is the number of predicates that is used, which is the same for all distributions.

The distributions used in the experiment are summarized below. The actual CONSTRUCT queries that were used to generate the distribution appear in Appendix A. The sizes of the data stores are shown in Table 6.1.

1. Distribution 1. One data store contains data about music artists, the second about music records. A number of records is made by the artists in the first data store.
   - s1, `_:artists`: artists from the Jamendo repository.
   - s2, `_:records`: records and tracks from the Jamendo repository.
2. Distribution 2. One data store contains data about music artists, the second about music records. A number of records is made by the artists in the first data store. A third data store has instances using the same property names as the other two sources, and includes resources that occur also in s1 and s2 as well. This data store is manually constructed by moving a few artists together with their corresponding records from the Jamendo repository to the new data store. The fourth data store contains this information for artists and records from the Magnatune music label. There are no resources in this data store that also occur in any of the other data stores.

- s1, `_:artists`: artists from the Jamendo repository.
- s2, `_:records`: records and tracks from the Jamendo repository.
- s3, `_:obscure`: obscure artists, records and tracks.
- s4, `_:magna`: Artists, records and tracks from the Magnatune repository.

To refer to the small variant of each distribution, the names 1A and 2A will be used. The large variants will be given the names 1B and 2B.

6.4.3 Queries

1. Query 1. Show where artists are based, and which records they made. This query uses properties related to artist resources.

   (?record, foaf:maker, ?artist)
   (?artist, foaf:img, ?c)
   (?artist, foaf:based_near, ?d)

   Contrary to the other planners, the overlap-based planner should be able to send the whole query to data store s4, since there is no resource overlap between s4 and the other data stores.

2. Query 2. Show the tracklisting of records, together with tags. This query uses the `tags:taggedWithTag` property on track resources.

   (?record, mo:track, ?track)
   (?record, tags:taggedWithTag, ?tag)

   Contrary to the other planners, the overlap-based planner should be able to choose the right data store for the `tags:taggedWithTag` property.

3. Query 3. Show name and tags of artists. This query uses the `tags:taggedWithTag` property on artist resources.

   (?artist, foaf:name, ?name)
   (?artist, tags:taggedWithTag, ?tag)

   Again, contrary to the other planners, the overlap-based planner should be able to choose the right data store for the `tags:taggedWithTag` property.

4. Query 4. Show details about records. This query uses properties from artists as well as records.

   (?record, foaf:maker, ?artist)
   (?record, tags:taggedWithTag, ?tag)
   (?record, mo:track, ?track)
   (?record, mo:image, ?img)

6.4.4 Experiments

The following experiments can now be conducted:

- Determine the costs of the generated query plans, per query planner for the four input queries, applied to four distributions. The cost functions that are considered are the total costs, and the first answer costs.

- Investigate the effect and applicability of query plan transformations in the overlap-based query planner. Note that since the rewriting rules are not implemented in the prototype, they will be applied manually on the generated initial plans. These manually transformed plans are then used as input for the cost calculator in the prototype.

6.5 Results

Figures 6.4, 6.5, 6.6, and 6.7 show the results per planner for each distribution. Each column in the chart consists of two parts: the cost to obtain the first answer ($CostFA_0(Q)$) and the cost to obtain the rest of the answer set. Together, the column denotes the total cost ($Cost_0(Q)$).

The total cost and first answer cost for the property-based planner is always equal to the cost of the naive planner. This is due to the parallelism that can always be exploited in the source queries for a triple pattern. The difference between the first answer cost and total cost is much higher in the large result sets (1B and 2B) than in the small result sets (1A and 1B). This is to be expected, since the transfer costs for the large result sets are much higher. For distribution 1, the cost of the naive and property-based planners is always higher than the cost of the overlap-based planner.

The source lookup table for each query are shown in Tables 6.2 and 6.3 for distribution 1 and 2 respectively. The overlap tables on which the SLTs are based, appear in Appendix A. In distribtion 1, each SLT consists of a single line. This explains why the first answer costs are very high in the overlap-based planner: answers from all source queries must be received before a first answer tuple can be produced. As an example, consider the query plan for query 4:

$$A_{D_1B}(Q_4) = {}^{2}[\text{track} \bowtie (\text{image} \bowtie \text{taggedWithTag})] \bowtie {}^{1}[\text{maker}]$$

The total cost is much lower than for the other planners because 3 triple patterns can be combined in a single source query.

As an example of the costs of the subexpressions in a query plan, consider the query plan that is generated by the overlap-based query planner for query 1 in distribution 2B (Figure 6.7.) After each line in the union expression the total cost and first answer cost of the

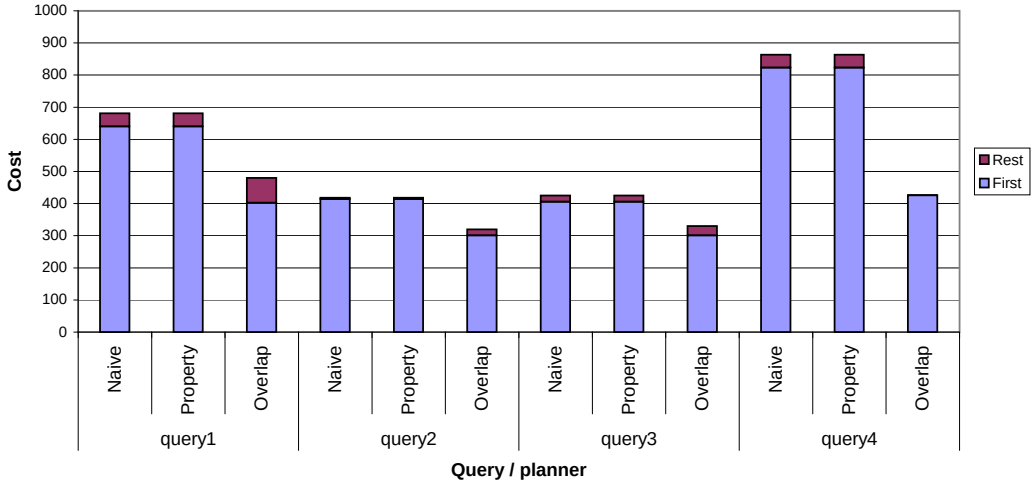


Figure 6.4: Cost per query for distribution 1A with 2 data stores, separated in the cost for the first answer and the cost for the remaining answers.

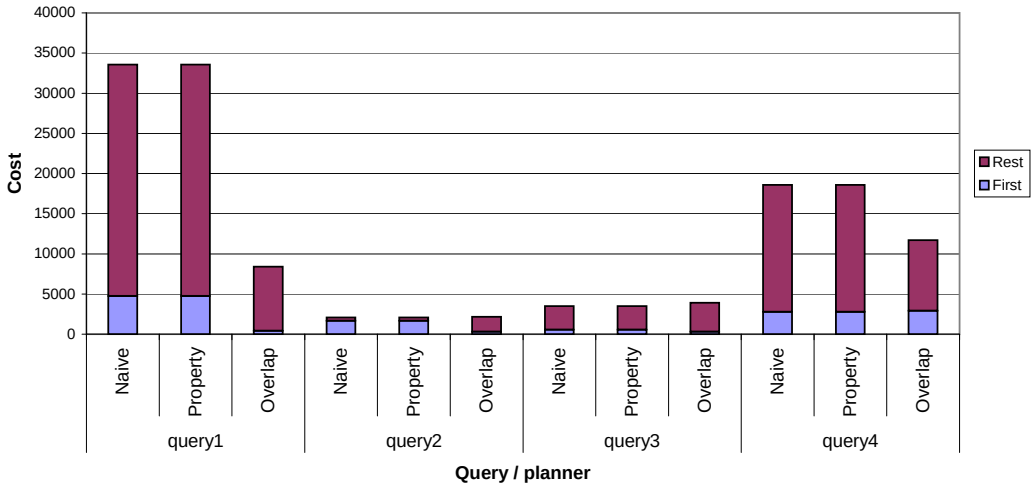


Figure 6.5: Cost per query for distribution 1B with 2 data stores.

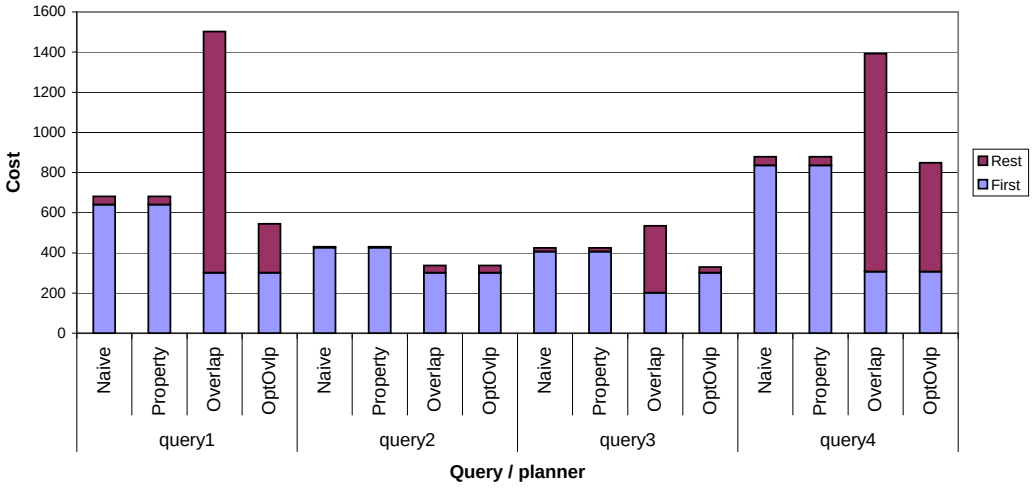


Figure 6.6: Cost per query for distribution 2A with 4 data stores.

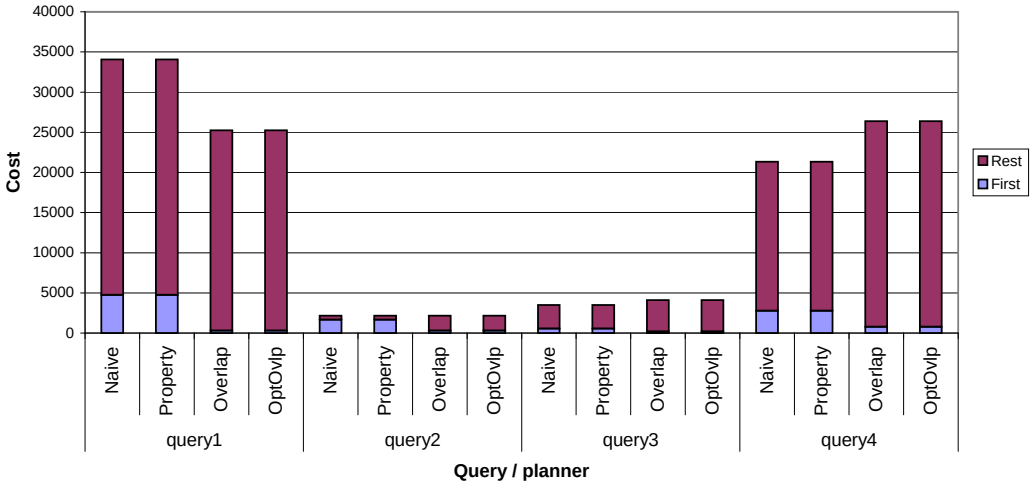


Figure 6.7: Cost per query for distribution 2B with 4 data stores.

#	foaf:maker	foaf:based_near	foaf:img
1	_:artists	_:artists	_:artists

(a) Source Lookup Table for query 1

#	mo:track	tags:taggedWithTag
1	_:records	_:records

(b) Source Lookup Table for query 2

#	foaf:name	tags:taggedWithTag
1	_:artists	_:artists

(c) Source Lookup Table for query 3

#	foaf:maker	tags:taggedWithTag	mo:image	mo:track
1	_:artists	_:records	_:records	_:records

(d) Source Lookup Table for query 4

Table 6.2: Source lookup table for each query, in distribution 1

#	foaf:maker	foaf:based_near	foaf:img
1	_:artists	_:artists	_:artists
2	_:artists	_:artists	_:obscure
3	_:artists	_:obscure	_:artists
4	_:artists	_:obscure	_:obscure
5	_:magna	_:magna	_:magna

(a) Source Lookup Table for query 1

#	mo:track	tags:taggedWithTag
1	_:records	_:records
2	_:obscure	_:obscure

(b) Source Lookup Table for query 2

#	foaf:name	tags:taggedWithTag
1	_:artists	_:artists
2	_:obscure	_:artists

(c) Source Lookup Table for query 3

#	foaf:maker	tags:taggedWithTag	mo:image	mo:track
1	_:artists	_:obscure	_:records	_:obscure
2	_:artists	_:obscure	_:records	_:records
3	_:artists	_:records	_:records	_:obscure
4	_:artists	_:records	_:records	_:records

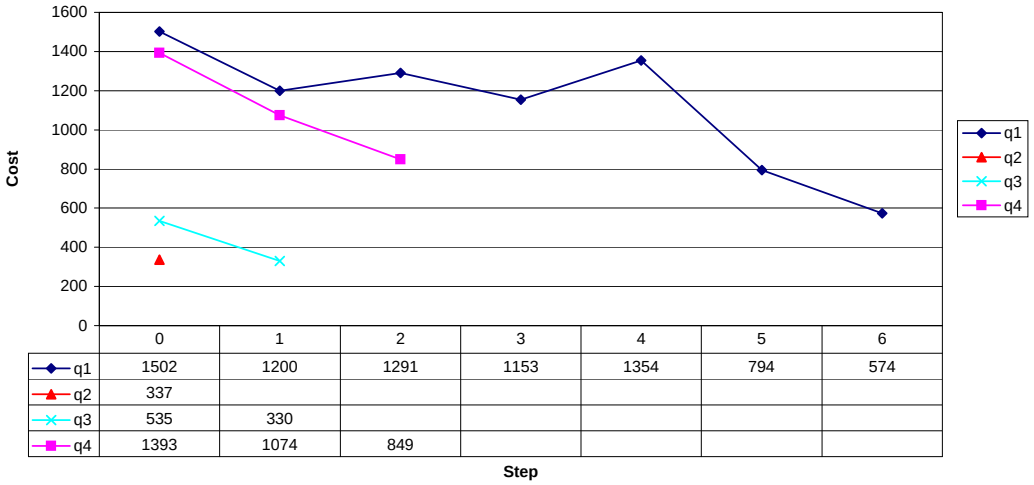
(d) Source Lookup Table for query 4

Table 6.3: Source lookup table for each query, in distribution 2

evaluation of that line are shown.

$$A_{D_{2B}}(Q_1) = \cup \begin{cases} {}^{1}[(\text{maker} \bowtie \text{based\_near}) \bowtie \text{img}] & 8418; 420 \\ {}^{1}[\text{maker} \bowtie \text{based\_near}] \bowtie {}^{3}[\text{img}] & 6315; 310 \\ {}^{3}[\text{based\_near}] \bowtie {}^{1}[\text{maker} \bowtie \text{img}] & 6314; 309 \\ {}^{1}[\text{maker}] \bowtie {}^{3}[\text{img} \bowtie \text{based\_near}] & 4206; 302 \\ {}^{4}[\text{img} \bowtie (\text{maker} \bowtie \text{based\_near})] & 462; 402 \end{cases}$$

Subexpression ${}^{1}[\text{maker}] \bowtie {}^{3}[\text{img} \bowtie \text{based\_near}]$ has the lowest first answer cost (302). To return the the first answer as early as possible, this subexpression should be executed first.



	0	1	2	3	4	5	6
q1	1502	1200	1291	1153	1354	794	574
q2	337						
q3	535	330					
q4	1393	1074	849				

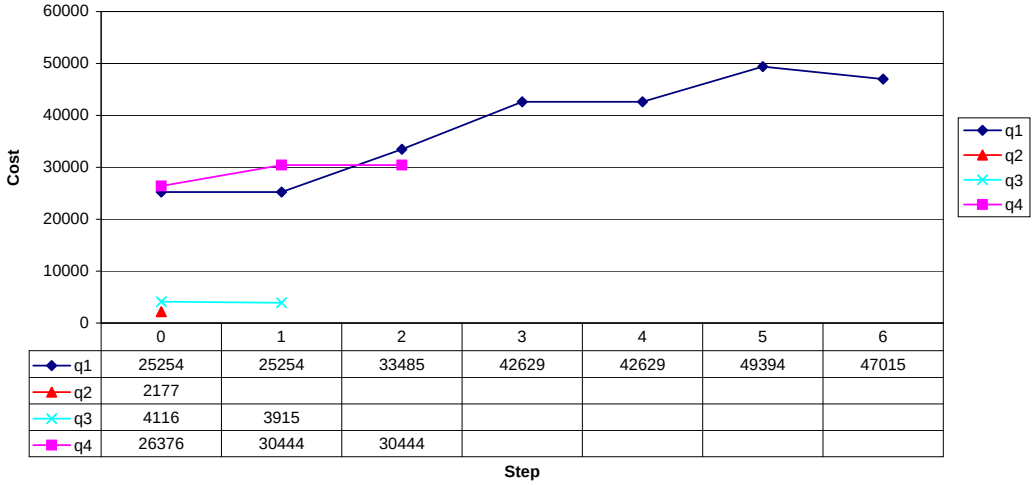
Figure 6.8: Cost of applying transformations to the initial query plan, in distribution 2A.

Transformation of query plans

In distribution 2 (Figures 6.6 and 6.7) the overlap-based planner has a higher cost than the other two planners for some queries. The reason for this is that all overlap combinations between sources are translated in the query plan. As was demonstrated in Chapter 5, redundancies may exist between source queries. Some of them can be removed altogether, others can be replaced by other expressions, such as a projection of other source queries in the plan. The costs for planner *optOvlp* represent the costs of a plan after applying rewriting steps on it.

The following source queries represent vertically complete mappings from graph pattern to data store:

- ${}^{1}[\text{foaf:img}_{AB} \bowtie \text{foaf:based\_near}_{AC}]$
- ${}^{1}[\text{foaf:name}_{AB} \bowtie \text{tags:taggedWithTag}_{AC}]$



	0	1	2	3	4	5	6
q1	25254	25254	33485	42629	42629	49394	47015
q2	2177						
q3	4116	3915					
q4	26376	30444	30444				

Figure 6.9: Cost of applying transformations to the initial query plan, in distribution 2B.

- ${}^{2}[\text{mo:track}_{AB} \bowtie \text{mo:image}_{AC}]$

Figure 6.8 shows the costs after a number of rewrite steps for all queries. For query 2 no meaningful rewriting is possible. For the other queries, a better (i.e. lower cost) plan could be obtained. The cost of query 1 was reduced by 62%, of query 3 by 38% and of query 4 by 39%.

Note that for query 1, there are two steps where the cost of the query plan increases. Such an increase is followed by a decrease at the subsequent step. This supports the proposition that the first or best improvement algorithms as a search strategy are not a good choice for this optimization problem.

The rewriting steps that are performed on query 1 are:

1. local-join-eval of ${}^{3}[\text{img,based\_near}]$
2. local-join-eval of ${}^{1}[\text{maker,based\_near,img}]$
3. local-join-eval of ${}^{1}[\text{maker,based\_near}]$
4. local-join-eval of ${}^{1}[\text{maker,img}]$
5. proj-intro-2; use vertical completeness of ${}^{1}[\text{foaf:img}_{AB} \bowtie \text{foaf:based\_near}_{AC}]$
6. proj-intro-1; use vertical completeness of ${}^{1}[\text{foaf:img}_{AB} \bowtie \text{foaf:based\_near}_{AC}]$

Introduce abbreviation $B = {}^{1}[(\text{img}_{AB} \bowtie \text{based\_near}_{AC}]$. The query plan for query 1 after

applying the rewrite steps is:

$$A_{D_2}(Q_1) = \cup \begin{cases} {}^{1}[\text{maker}] \bowtie B \\ {}^{1}[\text{maker}] \bowtie \Pi_{A,D}(B) \bowtie {}^{3}[\text{img}] \\ {}^{1}[\text{maker}] \bowtie \Pi_{A,C}(B) \bowtie {}^{3}[\text{based\_near}] \\ {}^{1}[\text{maker}] \bowtie {}^{3}[\text{img}] \bowtie {}^{3}[\text{based\_near}] \\ {}^{4}[\text{img} \bowtie (\text{maker} \bowtie \text{based\_near})] \end{cases} \tag{6.1}$$

The following rewrite step is performed on query 3:

1. proj-intro-2; use vertical completeness of ${}^{1}[\text{foaf:name}_{AB} \bowtie \text{tags:taggedWithTag}_{AC}]$

The following rewrite steps are performed on query 4:

1. proj-intro-2; use vertical completeness of ${}^{2}[\text{mo:track}_{AB} \bowtie \text{mo:image}_{AC}]$
2. local-join-eval of ${}^{2}[\text{taggedWithTag,img,track}]$

In Figure 6.9, the same rewriting steps are used on the query plans for distribution 2B. This time however, the cost of the plan for query 1 after these transformations is significantly higher than the cost of the original plan. The reason for this is the number of join operations that are executed on the mediator. In the original plan, 3 join operations were performed on the mediator, while 7 join operations were performed in the rewritten plan. In the cost model, join operations on the mediator have a much higher cost for operands with a high cardinality than join operations on the data stores. For re-used source queries, this cost is expected to be much lower, since an indexed join algorithm can be used in that case.

Since the cost of the plan was not decreased after performing transformations, the original plan costs are shown for the optimized overlap-based planner in Figure 6.7.

Chapter 7

Conclusion

The research questions that were raised in Chapter 1 are recapitulated here, and an overview of the answers that were presented in the different chapters of this thesis is given.

How can the distribution of an RDF database be described, and how can the data in the data stores be related?

Although the RDF model is different in many ways from the relational data model, fragments of an RDF graph can be specified in a way very much like the specification of fragments of a relation. The main difference is that fragments are not defined on a pre-defined schema, but on a newly defined one, called a view description. Such a view description is a graph pattern, possibly with selection and projection operators. A relation then corresponds to the extension of this view.

Because URIs provide a global resource identification mechanism, relationships between resources are explicit in the data itself. The forms that these relationships can take were analyzed. As a structural property of a graph, vertical completeness was defined. For a distribution, horizontal completeness was defined.

How can queries be answered in a distributed RDF database? How can the quality of a query processing approach be assessed?

A number of query optimization approaches were compared on the basis of scalability, the preservation of autonomy and the optimization performance. Two performance goals were identified: total response time, and the first answer response time. A cost function for both goals was defined to enable the comparison of the performance of different query processing approaches. Validating and improving this model based on actual measurements is future work. One area where the cost model is imprecise is the estimation of the size of the result set after a join operation.

When parallelism in the execution of source queries can be exploited, an index structure that only stores the instantiated predicates for each data store, is of limited use.

A source query in a query plan should be avoided if: 1) the answer set is empty; 2) the answers are not part of the final answer set; 3) the answers are also produced by other source queries in the plan. Avoiding an empty answer set can be done if the index structure stores

for each data store which graph patterns are instantiated on it. Avoiding answers that are not part of the final answer can be done if the mediator is aware of (the absense of) overlap between two source queries. To avoid retrieving duplicate answers, subset relations between the extensions of views can be used. A larger source query can be used if it represents a vertically complete view description. If it represents a vertically incomplete view description, the final answer set may not be complete.

How can the performance of query processing be improved when information about the distribution of the database is available?

Based on the graph properties and relationships between data stores that were identified for the first research question, a query processing method (the overlap-based method) was defined and implemented. This method and the approaches described for the second research question were applied on a number of database distributions and queries. A comparison of the approaches was made on the basis of the total cost and first answer cost. Because the cost model is not validated by actual measurements, interpretation of the results must be done under reserve.

Utilizing meta-information on the distribution can result in more efficient query processing on distributed data. By using larger source queries where possible, the total cost and the first answer cost can be greatly reduced. When many overlaps between data stores occur, the query plan in the overlap-based approach grows quickly. In many cases this sub-optimal plan can be improved with transformations.

Using overlap tables as an index structure does not scale well with the number of available data stores however, because the number of overlap possibilities grows exponentially with the number of data stores. Data store autonomy is preserved when the index is constructed incrementally during query processing.

7.1 Future work

Empirical performance evaluation

The performance evaluation conducted in this thesis was based on the outcomes of a cost model. To assess the real performance, an actual distributed setup should be used. The data stores in such a setup could consist of Sesame 2 servers. The source queries in the query plan would then be translated to SeRQL or SPARQL queries that are submitted over a TCP/IP communication channel. The mediator orchestrates this process and combines the answers that are returned. Such a setup can be used to calibrate and improve the cost model described in this thesis. Another important area of future work is the estimation of the result size of a join operation.

Also, the transformation rules and search strategy of the overlap-based method should be implemented and evaluated for actual queries.

Support for more query language constructs

The overlap-based approach and index structure were only defined for a simple query language. Many important constructs of SPARQL are not supported, such as selection conditions other than equality and optional graph patterns. A more advanced query language should be investigated for the input query as well as the source queries sent to the data stores.

Optimized distributions

In the usage scenario of a parallel database system, the RDF dataset is distributed for performance reasons. It is often desirable to maximize parallel execution during query processing. More extensive methods to specify the distribution are required; e.g. to specify ranges for the values of graph nodes. In this scenario, the data stores and distribution are controlled by the database administrator. This presents new opportunities to optimize the performance of query answering; e.g. the shipping of parts of the answer sets between data stores (for semi-joins).

Efficient result merging

With an increase of the number of data stores, often the number of join possibilities between source queries grows quickly as well. If the answer sets are represented as tuples, many join operations must be performed to compute the final answer set. A different logical representation of the answer sets of source queries may be required to obtain a more efficient way to merge the results. The effectiveness of such a logical representation is also influenced by the actual representation of answer sets that are transferred.

Adaptivity

The query processing method provides ways to include application-specific or user-specific (personalized) goals. For instance, the application may specify which trade-off should be made between first-answer response time, total response time, and number of answers returned.

Bibliography

[Amb99] J.L. Ambite, *Planning by Rewriting*, Ph.D. thesis, University of Southern California, 1999.

[AS05] G. Adamku and H. Stuckenschmidt, *Implementation and Evaluation of a Distributed RDF Storage and Retrieval System*, Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (2005), 393–396.

[Epp94] D. Eppstein, *Subgraph Isomorphism in Planar Graphs and Related Problems*, Information and Computer Science, University of California, Irvine, 1994.

[GHM04] C. Gutierrez, C. Hurtado, and A.O. Mendelzon, *Foundations of semantic web databases*, Proceedings of the 23th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (2004), 95–106.

[GP98] L. Gravano and Y. Papakonstantinou, *Mediating and Metasearching on the Internet*, Data Engineering Bulletin **21** (1998), no. 2, 28–36.

[Hay04] P. Hayes, *RDF Semantics*, W3C Recommendation, `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`, Feb 2004.

[HD05] A. Harth and S. Decker, *Optimized Index Structures for Querying RDF from the Web*, Proceedings of the 3rd Latin American Web Congress (2005), 71–80.

[KC04] G. Klyne and J.J. Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, Feb 2004.

[OB04] M. Ouzzani and A. Bouguettaya, *Query Processing and Optimization on the Web*, Distributed and Parallel Databases **15** (2004), no. 3, 187–218.

[PAG06] J. Pérez, M. Arenas, and C. Gutierrez, *Semantics and Complexity of SPARQL*, Proceedings of the 5th International Semantic Web Conference (ISWC), Athens, GA, USA (2006), 30–43.

[PS08] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF*, `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`, Jan 2008.

[RSSB00] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, *Efficient and extensible algorithms for multi query optimization*, ACM SIGMOD Record **29** (2000), no. 2, 249–260.

[SKS02] A. Silberschatz, H.F. Korth, and S. Sudarshan, *Database system concepts*, 4th ed., McGraw-Hill Boston, 2002.

[SSSW01] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers, *Towards a cost model for distributed and replicated data stores*, Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (2001), 461–467.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez, *Scaling heterogeneous databases and the design of DISCO*, International Conference on Distributed Computing Systems (1996), 449–457.

[VP98] V. Vassalos and Y. Papakonstantinou, *Using knowledge of redundancy for query optimization in mediators*, Proceedings of the AAAI Workshop on AI and Information Integration (1998), 29–35.

[Wie94] G. Wiederhold, *Interoperation, Mediation, and Ontologies*, Proceedings of the International Symposium on Fifth Generation Computer Systems (FGCSOB94), Workshop on Heterogeneous Cooperative Knowledge-Bases **3** (1994), 33–48.

Appendix A

Experiment details

Each query is supplemented with a `USING NAMESPACES` clause defining the relevant namespaces. Each query is also supplemented with a `LIMIT` clause that restricts the number of answers to 200 for distributions 1A and 2B.

Data store `_:artists`:

```
construct *
from {record} foaf:maker {artist}, {artist} foaf:name {name}; [foaf:based_near
{place}]; [foaf:img img]
[{artist} tags:taggedWithTag {artistTag}]
```

Data store `_:records`:

```
construct *
from {record} tags:taggedWithTag {tag}; mo:track {track}; mo:image {image};
dc:title {title}
```

Data store `_:obscure`:
This data store is manually constructed by moving a few artists together with their corresponding records from the Jamendo repository to the new data store.

tags	http://www.holygoat.co.uk/owl/redwood/0.1/tags/
dc	http://purl.org/dc/elements/1.1/
time	http://www.w3.org/2006/time#
tl	http://purl.org/NET/c4dm/timeline.owl#
mo	http://purl.org/ontology/mo/
foaf	http://xmlns.com/foaf/0.1/
event	http://purl.org/NET/c4dm/event.owl#
xsd	http://www.w3.org/2001/XMLSchema#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#

Table A.1: Namespaces used in the data set

rdf:type
foaf:made
foaf:based_near
foaf:img
foaf:name
tags:taggedWithTag
dc:date
mo:available_as
mo:image
mo:track
dc:title
foaf:maker
tags:tagName
mo:biography
dc:description
tl:onTimeLine
mo:text
mo:recorded_as
mo:published_as
mo:time
event:factor
dc:format
mo:license
mo:track_number

Table A.2: All properties that occur in the data set

Data store `_:magna`:

```
construct *
from {record} foaf:maker {artist}, {artist} foaf:name {name}; [foaf:based_near
{place}]; [foaf:img {img}]
, [{artist} tags:taggedWithTag {artistTag}]
, [{record} tags:taggedWithTag {tag}], {record} mo:track {track}; [mo:image
{image}]; dc:title {title}
```

Properties	Bigraph type	Overlap table
[mo:track,mo:image]	SubjectSubject	[_:artists, _:records]; [_:records, _:records]
[tags:taggedWithTag,mo:image]	SubjectSubject	[_:records, _:artists]
[tags:taggedWithTag,mo:track]	SubjectSubject	[_:artists, _:artists]; [_:records, _:artists]; [_:obscure, _:obscure]
[foaf:based_near,tags:taggedWithTag]	SubjectSubject	[_:artists, _:artists]
[foaf:based_near,foaf:img]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]; [_:magnatune, _:magnatune]
[foaf:img,tags:taggedWithTag]	SubjectSubject	[_:artists, _:artists]
[foaf:made,mo:image]	ObjectSubject	[_:artists, _:records]
[foaf:made,mo:track]	ObjectSubject	[_:artists, _:artists]; [_:records, _:artists]; [_:obscure, _:obscure]
[foaf:made,tags:taggedWithTag]	ObjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]
[foaf:made,tags:taggedWithTag]	SubjectSubject	[_:artists, _:artists]
[foaf:made,foaf:based_near]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]
[foaf:made,foaf:img]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]
[foaf:made,foaf:name]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]
[foaf:name,tags:taggedWithTag]	SubjectSubject	[_:artists, _:artists]
[foaf:name,foaf:based_near]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]; [_:magnatune, _:magnatune]
[foaf:name,foaf:img]	SubjectSubject	[_:artists, _:artists]; [_:obscure, _:obscure]; [_:magnatune, _:magnatune]

Table A.3: Overlap tables (for the overlap-based planner) for distribution 2A and 2B. Only the properties that are used in the queries in the experiments are shown here.