Distributed XML Query Processing

by

Patrick Kling

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

While centralized query processing over collections of XML data stored at a single site is a well understood problem, centralized query evaluation techniques are inherently limited in their scalability when presented with large collections (or a single, large document) and heavy query workloads. In the context of relational query processing, similar scalability challenges have been overcome by partitioning data collections, distributing them across the sites of a distributed system, and then evaluating queries in a distributed fashion, usually in a way that ensures locality between (sub-)queries and their relevant data. This thesis presents a suite of query evaluation techniques for XML data that follow a similar approach to address the scalability problems encountered by XML query evaluation.

Due to the significant differences in data and query models between relational and XML query processing, it is not possible to directly apply distributed query evaluation techniques designed for relational data to the XML scenario. Instead, new distributed query evaluation techniques need to be developed. Thus, in this thesis, an end-to-end solution to the scalability problems encountered by XML query processing is proposed.

Based on a data partitioning model that supports both horizontal and vertical fragmentation steps (or any combination of the two), XML collections are fragmented and distributed across the sites of a distributed system. Then, a suite of distributed query evaluation strategies is proposed. These query evaluation techniques ensure locality between each fragment of the collection and the parts of the query corresponding to the data in this fragment. Special attention is paid to scalability and query performance, which is achieved by ensuring a high degree of parallelism during distributed query evaluation and by avoiding access to irrelevant portions of the data.

For maximum flexibility, the suite of distributed query evaluation techniques proposed in this thesis provides several alternative approaches for evaluating a given query over a given distributed collection. Thus, to achieve the best performance, it is necessary to predict and compare the expected performance of each of these alternatives. In this work, this is accomplished through a query optimization technique based on a distribution-aware cost model. The same cost model is also used to fine-tune the way a collection is fragmented to the demands of the query workload evaluated over this collection.

To evaluate the performance impact of the distributed query evaluation techniques proposed in this thesis, the techniques were implemented within a production-quality XML database system. Based on this implementation, a thorough experimental evaluation was performed. The results of this evaluation confirm that the distributed query evaluation techniques introduced here lead to significant improvements in query performance and scalability both when compared to centralized techniques and when compared to existing distributed query evaluation techniques.

Acknowledgements

I would like to take the opportunity to acknowledge those who have made this work possible. First and foremost, I would like to thank my advisor, M. Tamer Özsu, not only for the expert advice he has given me throughout my studies, but also for his unwavering support and relentless encouragement, which have guided me through the past few years.

I would also like to thank Khuzaima Daudjee for the many productive discussions we have had throughout my studies. His input (both on technical details and on the big picture of this field of research) has played an important part in shaping this work.

I am also grateful to Anisoara Nica for her support in developing the cost model for this work. Her expertise and guidance have allowed me to develop this aspect of my work into a significant component of my thesis.

Finally, I would like to thank my external examiner, Martin Kersten, and the remaining members of my examining committee, Frank Wm. Tompa, Ashraf Aboulnaga, and Paul A. S. Ward, for the valuable questions and comments the have given me, which have helped me significantly improve the quality of this thesis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DEP	distributed execution plan
FTP	fragmentation tree pattern
LQP	local query plan
QTP	query tree pattern
XML	Extensible Markup Language

List of Symbols

Documents and Fragments

d	document
D	set of documents
f_i	fragment
f_i^{H}	horizontal fragment
f_i^{V}	vertical fragment
f_ρ	root fragment
$f_{\max}$	fragment with highest cost
$P_b^{i \to j}$	proxy node representing edge from fragment f_i^{V} to fragment f_j^{V}
$RP_b^{i \to j}$	root proxy node representing edge from fragment f_i^{V} to fragment f_j^{V}
s	sub-tree
o	node in document
$\leq_{\mathrm{doc}}$	document order
$<_{\mathrm{doc}}$	strict document order
$\mathrm{subtset}(f_i)$	set of sub-trees in fragment f_i
$\mathrm{nsubt}(f_i)$	number of sub-trees in fragment f_i

Schemas

S	schema
$\langle \Sigma, \Psi, s, m, \rho \rangle$	schema
σ	node type
Σ	set of node types
ψ	schema edge
Ψ	set of schema edges
$s(\psi)$	cardinality of a schema edge
$m(\sigma)$	domain of a node type
ρ	root node type

XQ Queries

c_{str}	string constant
c_{num}	numeric constant
θ_{str}	string constant
θ_{num}	string constant
/	child step
//	descendant step
$*$	wildcard node test

Tree Patterns

$\langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$	tree pattern
n	pattern node
N	set of pattern nodes
l	logic node
L	set of logic nodes
x	pattern or logic node
$\text{parent}(x)$	parent node of x

r	root node
e	pattern edge
E	set of pattern edges
$\nu(n)$	node test of n
$c(n)$	value constraint of n
$\varepsilon(e)$	axis of e
$\lambda(l)$	logic operator of l ($\wedge$, $\vee$, or $\neg$)
t	extraction point
T	set of extraction points
μ	pattern match
M	set of pattern matches
q_k	QTP query
q_k^u	local QTP corresponding to query q_k

Plans and Their Properties

θ	comparison operation
p_k^u	LQP corresponding to query q_k's local QTP q_k^u
$\bar{p}_k^u$	remainder plan of p_k^u
$\bar{p}_k^{u,A}$	modified remainder plan of p_k^u
P	set of LQPs
G_P	distributed execution plan consisting of local plans in P
G_P^{P}	physical distributed execution plan corresponding to G_P
$R(G_P)$	sequence of tuples produced by G_P
$A\left(G_P\right)$	set of attributes comprising tuples in $R(G_P)$
$O\left(G_P^{\mathrm{P}}\right)$	set of attributes that are in document order in $R(G_P)$
a_u^{rp}	attribute in $R(G_P)$ containing root proxy nodes matched by p_k^u

a_u^p	attribute in $R(G_P)$ containing proxy nodes corresponding to root proxy nodes matched by p_k^u
a_i^e	attribute in $R(G_P)$ containing extraction point nodes in the global QTP
A	set of attributes
$\mathrm{id}(a_u^p)$, $\mathrm{id}(a_u^{rp})$	ID of proxy/root proxy node
$\mathrm{ntpath}(a_u^{rp})$	node type path of root proxy node
$\mathrm{card}(G_{P_u})$	cardinality of G_{P_u}
$\mathrm{cost}(G_{P_u})$	response time of G_{P_u}
$\mathrm{cost\text{-}first}(G_{P_u})$	time to first tuple of G_{P_u}
$\mathrm{nsubt}(p_k^u)$	number of sub-trees accessed by LQP p_k^u
$\mathrm{subtcost}({}^a p_k^u)$	cost of evaluating the physical LQP ${}^a p_k^u$ over a single sub-tree
$\mathrm{docdelay}(G_{P_u})$	time until G_{P_u} returns all tuples derived from an entire document
$\mathrm{tupfirst}(G_{P_u}, G_{P_u} \bowtie G_{P_v})$	number of tuples needed from G_{P_u} to produce one join tuple
$\mathrm{tupdelay}(G_{P_u}, G_{P_u} \bowtie G_{P_v})$	time taken by G_{P_u} to produce $\mathrm{tupfirst}(G_{P_u}, G_{P_u} \bowtie G_{P_v})$ tuples
$\mathrm{nullprob}(G_{P_v})$	probability that there is no match in G_{P_v} for a given proxy node
$\mathrm{samegroup}(G_{P_u}, a_v^p)$	average number of tuples returned by G_{P_u} that differ only in their value of a_v^p
$\mathrm{groupdelay}(G_{P_u}, a_v^p)$	time to obtain $\mathrm{samegroup}(G_{P_u}, a_v^p)$ tuples from G_{P_u}
probecost	cost of probing a hash table

Logical Operators

$\rhd$	anti-join
$\mathbb{G}$ $\mathbb{A}$	grouping with aggregation
$\bowtie$	join
$\odot$	merge
⟕	outer join
π	projection
scan	scan
σ	selection
$\Box$	singleton scan
$\mathbb{S}$	sorting
Υ	unnest map

Physical Operators

$\bowtie^{\mathrm{H}}$	hash join
$\bowtie^{\mathrm{SH}}$	hash join
$\bowtie^{\mathrm{I}}$	index join
$\bowtie^{\mathrm{M}}$	merge join
$\odot^{\mathrm{C}}$	concatenation-based merge
$\odot^{\mathrm{DI}}$	merge with document-based interleaving
$\odot^{\mathrm{FI}}$	merge with full interleaving
$\odot^{\mathrm{SC}}$	stable concatenation-based merge

Chapter 1

Introduction

Over the past decade, XML [99] has become a commonly used format for storing and exchanging data in a wide variety of systems. Due to this widespread use, the problem of effectively and efficiently managing large XML collections has attracted significant attention in both the research community and in commercial products. One can claim that techniques for the management of XML data residing on a single system and for the centralized evaluation of queries over these data are now well understood. However, because these techniques are inherently based on centralized execution on a single machine, their scalability is limited when presented with large collections (or single, large documents) and heavy query workloads.

In relational database systems, these scalability challenges have been successfully addressed by partitioning data collections and distributing the resulting fragments across a distributed system. This makes it possible to distribute and parallelize query processing [115]. This work is focused on similarly exploiting distribution in the context of an XML database system. While there are some similarities between the way relational database systems can be distributed and the opportunities for distributing XML database systems, the significant differences in both data and query models make it impossible to directly apply relational techniques to XML. Therefore, new solutions need to be developed to distribute XML database systems.

While there has been research interest in distributed XML query processing for some time, much of the existing work has focused on the problem of integrating multiple repositories into a single XML view (e.g., [2, 6, 43, 101]). While some of the work in this area deals with the problem of optimizing queries over distributed collections of XML data, the goals and constraints faced in the data integration scenario are decidedly different from those seen in a scenario where distribution is used to improve scalability. For instance, whereas data integration requires a fragmentation model that can express the complex ways in which individual and possibly redundant data sources might need to be integrated, the fragmentation model used in this work does not need to meet this requirement and can therefore be optimized entirely for the purpose of improving query performance.

A few publications have focused on distribution as a means to improve scalability. These either rely heavily on replicated index structures that complicate the handling of updates [31] or they focus primarily on minimizing network communication cost [33, 39, 40, 124]. In contrast to this, the work presented here is concerned with finding an end-to-end solution that takes into account the overall cost of distributed query evaluation.

The primary focus of this work is on a scenario in which a collection of XML data is distributed across multiple machines within a single data centre (ensuring high-throughput, low-latency communication). Experimental results show that the techniques presented here, which are specifically designed for improving the overall cost of query evaluation, outperform techniques that focus on communication cost alone.

1.1 Focus and Motivation

This thesis focuses on the following aspects of the problem of improving the scalability of XML query evaluation through distribution:

- First, a *distribution model* for XML is developed. This model is based on a fragmentation approach that partitions the collection based on characteristics of its content and structure. A key advantage of this model is that it is simple and yet sufficiently powerful to significantly improve the scalability of distributed query evaluation. This simplicity makes it easier to identify a suitable fragmentation for a given query workload.

The distribution model supports horizontal fragmentation (based on selection operators and predicates) and vertical fragmentation (based on a partitioning of the set of element types in a schema). Both types of fragmentation are designed to be orthogonal, which means that they can be used together to achieve hybrid fragmentation. While the semantics of this model are inspired by relational fragmentation techniques, it is important to point out that the characteristics of XML, such as its nested data model and structure-based queries, lead to a set of challenges and optimization opportunities that differ significantly from what is encountered in the relational context.

- Next, the focus is on techniques for evaluating queries over a collection that has been distributed according to the proposed distribution model. These techniques begin by *localizing* the query, i.e., by transforming the overall query into sub-queries that can be evaluated independently and in parallel at the sites that hold the fragments that are relevant to the query. Each sub-query is then transformed into a local query plan, with each site choosing the most appropriate local query evaluation strategy independently. Next, a *distributed execution plan* is specified, which describes how and in which order these local query plans are to be executed and how their results are combined to form the overall query result.

- Then, methods for improving the performance of distributed execution plans are discussed. Several different techniques for this are introduced. One such technique focuses on *pruning* the set of fragments that need to be visited to answer a given query. A novel technique is introduced that can be used to detect fragments that can be omitted from a distributed execution plan without compromising the correctness of the query result. Experiments show that pruning leads to a significant performance improvement in many realistic use cases.

- Further techniques for improving the performance of distributed query evaluation are discussed next. For instance, a suite of rewrites are presented that make it possible to improve query performance by *pushing join operators* from the distributed execution plan into the local query plans. This increases parallelism (an important contributor to good query performance in the distributed scenario considered in this work) and makes it possible to skip large portions of the data stored within a fragment in many cases. Experiments validate that these techniques further improve query performance beyond the level achieved by pruning alone.

- Based on the distributed query evaluation techniques proposed in this thesis, there are usually many different distributed plans that can be used to evaluate a given query over a given distributed collection. While all of these plans yield the correct query result, they may vary widely in their performance. This makes it important to choose the best plan for a given query and distributed collection. This problem is addressed by a *cost-based optimization technique*, which enumerates all possible plans and compares them based on their estimated performance. Since communication costs are usually low within the context of a data centre, the cost of distributed query evaluation is dominated by the cost of evaluating sub-queries over individual fragments. The cost model used in this work exploits this and estimates the overall cost of a distributed plan by composing the costs of its constituent local plans.

- Based on the query evaluation techniques and the cost model, a *workload-aware fragmentation technique* is then proposed. This technique is designed to fragment a given collection in a way that will result in good performance for a given set of queries.

When combined, these techniques represent a complete solution to the problem of increasing the scalability of XML query evaluation. To validate this, an extensive set of experiments are performed. These confirm that a combination of the techniques presented in this thesis leads to a significant improvement in query performance and scalability, both when compared to centralized techniques and to existing distributed approaches. The contribution of each individual technique to this performance improvement is analyzed thoroughly, both to validate the cost model and to gain a further understanding of the performance characteristics of this work.

It is important to point out that all of the distributed query evaluation techniques described in this work are designed to work without relying on a globally replicated index structure, because using such a structure could limit the scalability of a distributed system and negatively affect the performance of updates. In addition, all of these techniques work independently of the local query evaluation strategies used for evaluating sub-queries at the individual sites in the system, allowing for maximum flexibility.

To illustrate the challenges and optimization opportunities of distributed XML query evaluation, consider two sample collections, which will be used as examples throughout this thesis: a collection that is horizontally fragmented and one that is vertically fragmented. For both collections, a brief summary is given of how the techniques presented in this thesis improve query performance.

1.1.1 Horizontal Fragmentation

Figure 1.1 shows a horizontally fragmented data collection consisting of four documents representing information about authors and their publications. The horizontal fragmentation is defined based on the first character of the authors' last names, placing 'John Adams' in fragment f_1^{H}, 'Jane Dean' in fragment f_2^{H} and 'John Smith' as well as 'William Shakespeare' in fragment f_3^{H}.

Consider evaluating the following XPath query (q_1):

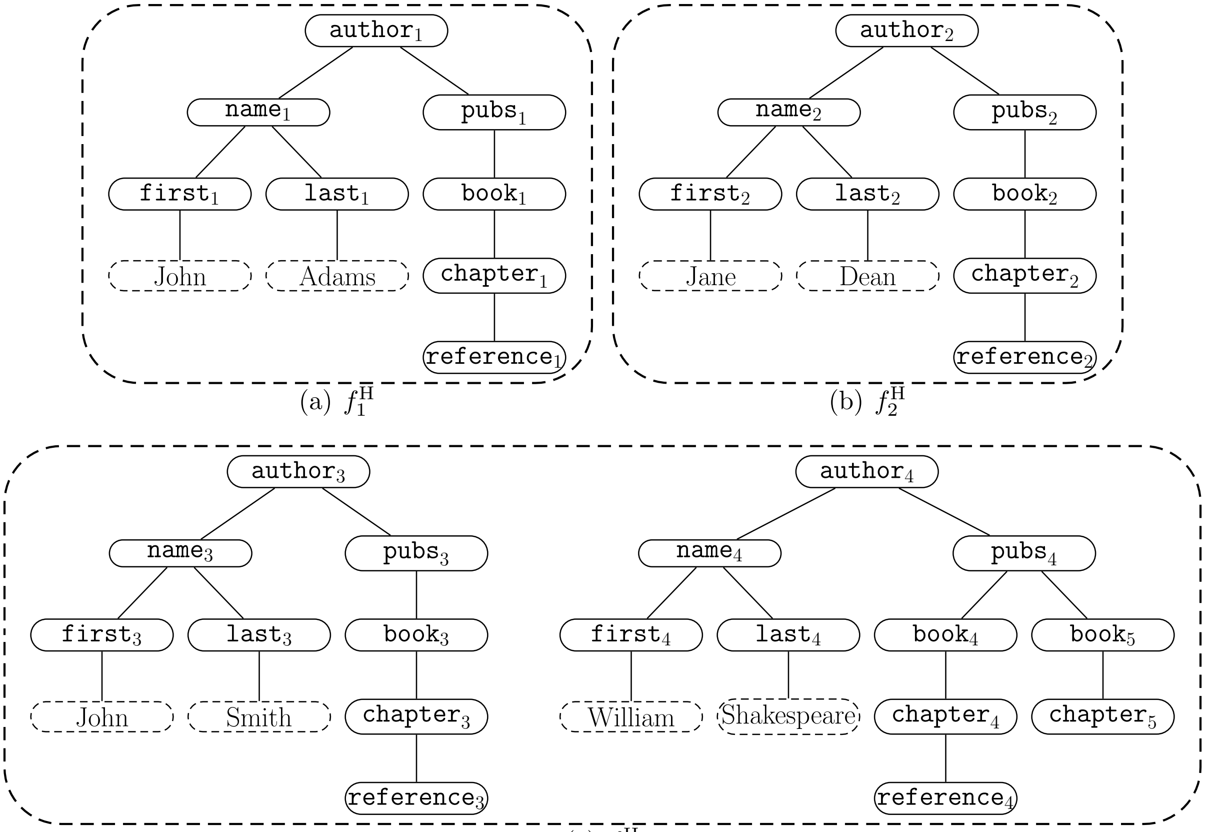
```
/author[name[first[.= 'William'] and last[.='Shakespeare']]]//reference
```

A simple approach to answering this query might evaluate the original query over each fragment independently and then gather the results. However, in the example shown here, it is easy to see that the fragments f_1^{H} and f_2^{H} cannot possibly contribute to the result of this query since they correspond to authors whose last names start with the letters 'A' and 'D', respectively, whereas the query is searching for authors whose last name is 'Shakespeare'. Pruning these fragments makes it possible to answer the query without contacting the sites at which they are stored. One of the contributions of this thesis is a technique that detects irrelevant fragments and avoids accessing them during distributed query evaluation.

Once the irrelevant fragments have been eliminated, the original query is evaluated over each remaining fragment. Each fragment will yield a sequence of results, which are then combined to the overall sequence of results, while paying attention to the overall ordering of the query results (which may require sorting).

1.1.2 Vertical Fragmentation

Figure 1.2 shows a collection that has been fragmented vertically. Ignoring the nodes labeled as $P_b^{i \to j}$ and $RP_b^{i \to j}$ for now, it can be seen see that `author` and `agent` nodes are stored in fragment f_1^{V}, the nodes related to the author's name are stored in fragment f_2^{V}, `pubs` and `book` nodes are stored in fragment f_3^{V} and `chapter` and `reference` nodes are stored in fragment f_4^{V}.

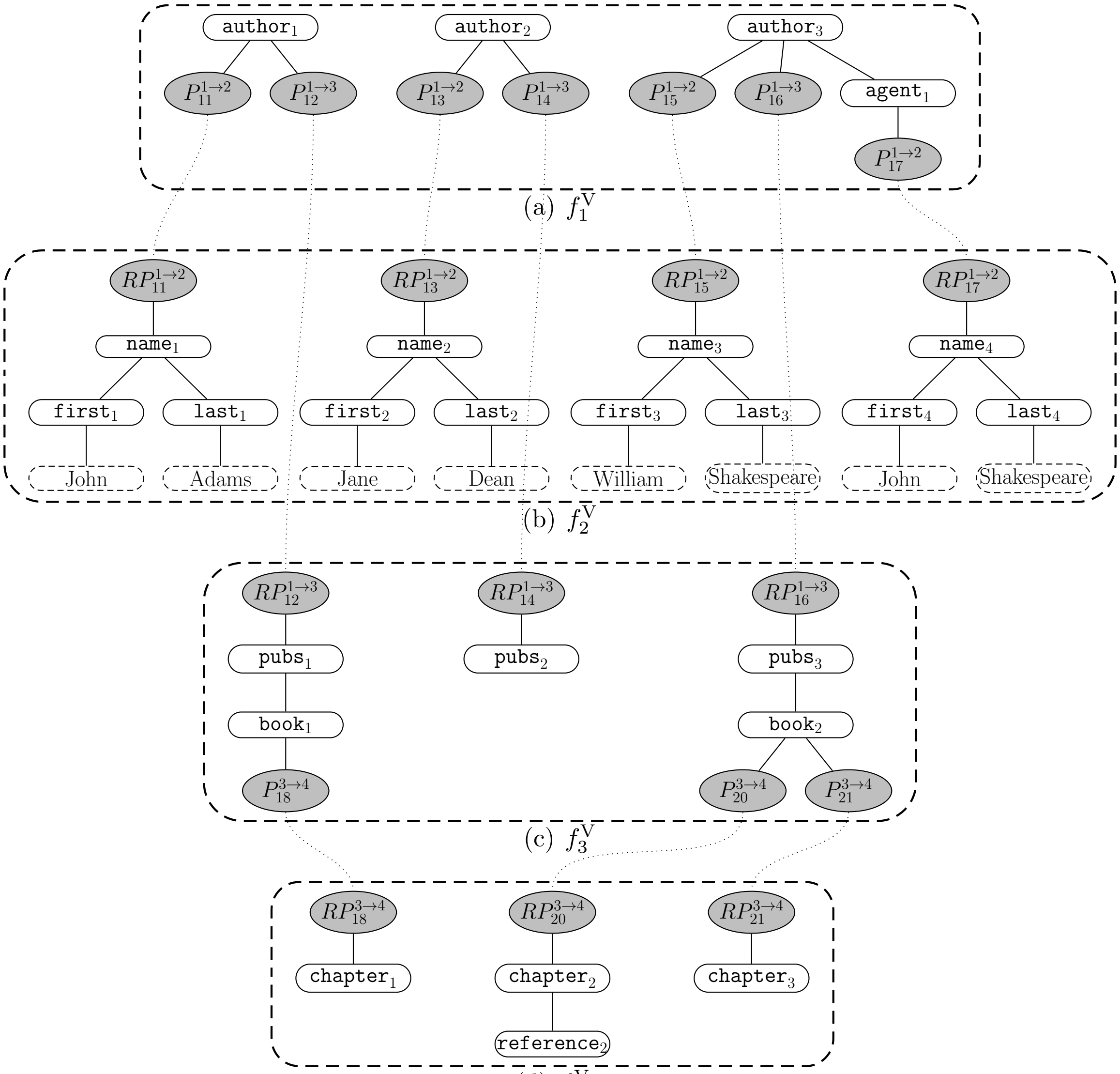


(a) f_1^{H}

(b) f_2^{H}

(c) f_3^{H}

Figure 1.1: A horizontally fragmented collection



(a) f_1^{V}

(b) f_2^{V}

(c) f_3^{V}

(d) f_4^{V}

Figure 1.2: A vertically fragmented collection

When evaluating q_1 over the vertically fragmented collection, in the general case, all four fragments need to be accessed. Fragment f_2^{V} is needed to evaluate the value constraint predicates, fragment f_4^{V} is needed to obtain result nodes and fragments f_1^{V} and f_3^{V} are needed to evaluate structural constraints. One of the contributions of this thesis is a technique that avoids accessing certain fragments that are only needed for evaluating structural constraints. In the example shown here, it is possible to avoid accessing f_3^{V}. Conceptually, this is possible because it can be inferred that the sub-trees in f_3^{V} are only needed to establish an ancestor-descendant relationship between sub-trees in f_1^{V} and sub-trees in f_4^{V}.

Once the fragments that are irrelevant for a given query have been eliminated, the sub-queries that need to be evaluated over each remaining fragment are determined. With vertical fragmentation, this step is complicated by the fact that each fragment covers a different portion of the overall schema. Therefore, a technique is proposed that decomposes the query into a set of sub-queries corresponding to the portions of the schema covered by the individual fragments.

After a sub-query for each relevant fragment has been obtained, it is necessary to determine how to best execute these sub-queries and how to combine their results to the overall query result. How this is done is specified in a distributed execution plan. Whereas with horizontal fragmentation the sequences of results corresponding to each fragment can simply be concatenated, with vertical fragmentation joins need to be performed to combine the results derived from individual fragments. For a given fragmentation and query, there are usually a large number of distributed execution plans that all lead to the correct result but that differ in factors such as join order and join strategy. As will be shown, different distributed execution plans lead to vastly different levels of query performance. Therefore, a major focus of this thesis is on the generation of good execution plans. Particular attention is paid to achieving a high level of parallelism, reducing the sizes of intermediate results, and decreasing the cost of evaluating sub-queries.

1.2 Contributions

The specific contributions of this work are the following:

1. A formal definition of a fragmentation model for XML is provided. This model makes it possible to fragment and distribute a collection in order to improve query performance. Along with this model, a succinct method for specifying the horizontal or vertical fragmentation of a collection of XML documents is proposed. This can then be used as the basis for distributed query optimization.

2. Using this specification, a technique is introduced that decomposes a query into sub-queries, each of which corresponds to a single fragment. A distributed execution plan describes how the results of sub-queries are combined to form the overall query result.

3. Based on the fragmentation model, a complete suite of techniques for identifying irrelevant fragments and pruning them from a distributed execution plan is proposed.

4. The benefits and drawbacks of different types of execution plans are analyzed thoroughly. A novel query evaluation technique is proposed that makes it possible to skip irrelevant sub-trees within a fragment by pushing cross-fragment joins into local query plans. By pipelining aggressively, a negative impact on parallelism is avoided.

5. A cost model for distributed execution plans is proposed based on the end-to-end response time of query evaluation. The focus of this model is on composing the costs of local query plans (which dominate the cost of distributed query execution in a data centre environment) while taking into account parallelism and the query evaluation techniques presented in this thesis. Using this cost model, the best distributed execution plan for a given query and collection can be determined.

6. Building on the cost model, a technique is proposed that can fragment an XML collection such that the performance of a given query workload (when evaluated using the techniques presented in this thesis) is improved.

7. All of the distributed query evaluation techniques presented in this thesis have been implemented within the XML database system Natix and deployed within a virtualized data centre. This has made it possible to verify that they significantly improve the performance and scalability of query evaluation, both compared to centralized query evaluation and to existing distributed techniques.

Figure 1.3 shows how these individual contributions, when taken together, yield an end-to-end solution to the problem of distributed query evaluation over fragmented collections of XML data. First, the query dispatcher decomposes an incoming query into sub-queries that can be evaluated over individual fragments. Then, irrelevant fragments (and the sub-queries corresponding to them) are pruned. Next, each of the remaining sub-queries is sent to the site that holds the corresponding fragment. At this site, a local query plan for this sub-query is determined and cost estimates for this local query plan are sent back to the dispatcher. The dispatcher then uses these cost estimates to determine the distributed execution plan with the lowest overall cost. Finally, this distributed execution plan (along with the local query plans contained in it) is evaluated over the sites of the distributed system. This yields the overall query result, which is shipped to the query dispatcher and returned.

1.3 Organization

The remainder of this thesis is structured as follows. Chapter 2 describes a data model and query model for XML. Chapter 3 discusses related work, both in the context of XML data processing and distributed databases in general. Chapter 4 discusses in detail how the fragmentation and distribution of XML collections is modeled. Chapter 5 describes how queries can be evaluated over a distributed collection in a way that maximizes scalability. The main focus of this chapter is on the localization of data that are relevant to the query and the generation of a distributed execution plan. Together, these techniques represent the foundation for Chapter 6, which focuses on techniques for improving the performance of distributed execution plans. In Chapter 7, all of this is then tied together by a cost model that helps to identify the best distributed plan for a given query and fragmented

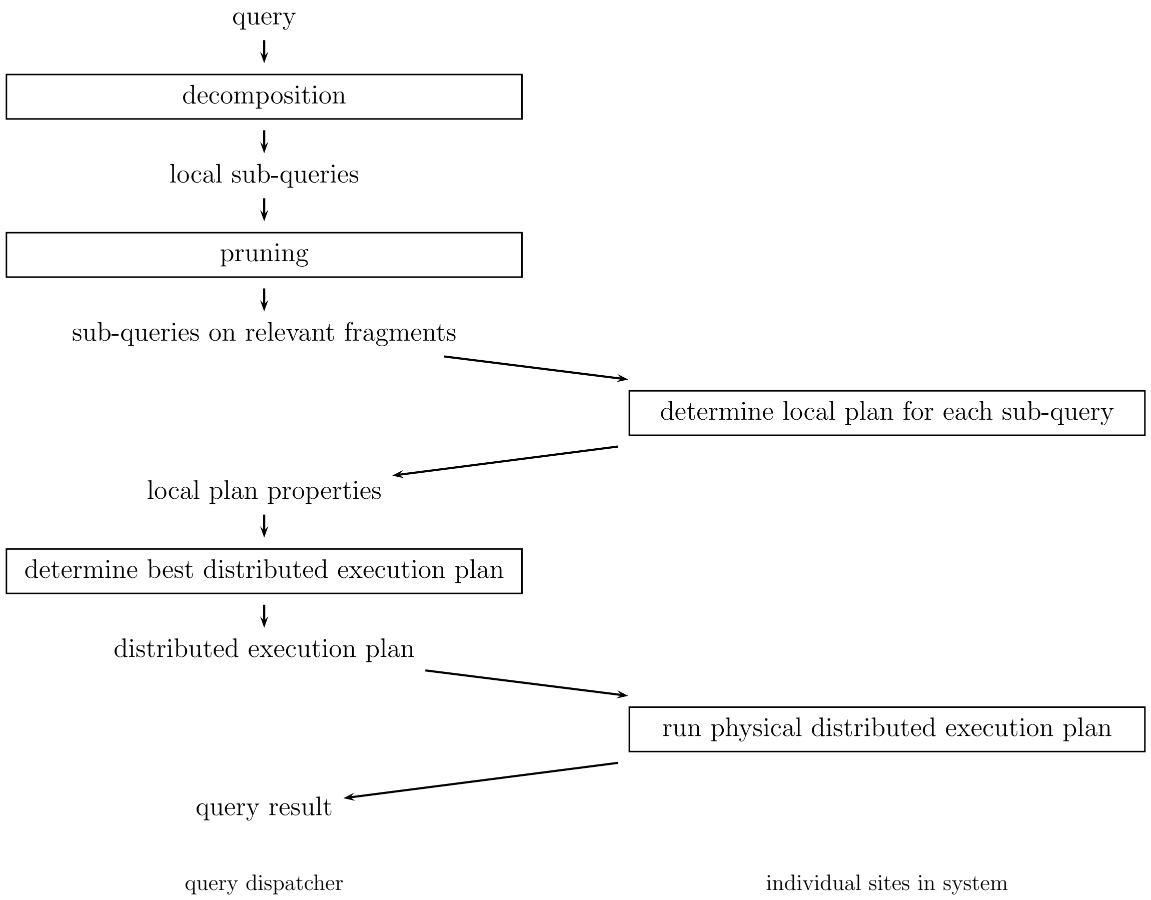

query
decomposition
local sub-queries
pruning
sub-queries on relevant fragments
determine local plan for each sub-query
local plan properties
determine best distributed execution plan
distributed execution plan
run physical distributed execution plan
query result
query dispatcher
individual sites in system

Figure 1.3: Distributed query processing overview

collection. Chapter 8 introduces a technique that can determine the best fragmentation for a given query workload. To validate the effectiveness of this work, a thorough experimental evaluation is presented in Chapter 9. Chapter 10 then presents a summary of the thesis, conclusions, and areas of interest for future work.

Chapter 2

XML Data and Query Model

This chapter describes the XML data model used in this work. It also introduces XQ, the class of queries supported.

2.1 Data Model

An XML collection can be described as a set of labeled, ordered trees. While XML is a self-describing format that can be used without a schema, in practice, the structure of document trees is usually constrained by a schema that specifies how elements may be nested and what the domain of their textual content is. This work exploits the schema definition (which is assumed to be the same for all documents in the collection) in order to improve the performance of distributed query evaluation.

A schema is usually defined in a language such as DTD [99] or XML Schema [133, 132, 100]. For simplicity, here, a simple directed graph representation is used that covers only those aspects of the schema that are important for the purpose of distributed query evaluation. For example, the distinction between XML elements and attributes is ignored and both are treated uniformly as *nodes*. To avoid ambiguity, these nodes will sometimes be referred to as *collection nodes* to better distinguish them from nodes in other tree or

```
author(name, pubs, agent?)
pubs(book*, article*)
book(chapter*)
article(chapter*)
chapter(reference?)
reference(chapter)
agent(name)
name(initial?, first, last, title?)
initial(#text)
first(#text)
last(#text)
title(#text)
```

Figure 2.1: A schema

graph structures. In analogy with the treatment of nodes, both element types and attribute names are referred to as *node types*.

Assuming that the original schema definition does not contain unspecified portions (such as those defined using the DTD keyword `ANY`, which would lead to an incomplete schema graph), it is straightforward to extract the information needed for this graph representation from a DTD[1] or an XML Schema. It is important to realize that the schema graph may be less restrictive than the original DTD or XML Schema. However, since the schema graph is never used to validate documents, this does not pose a problem (cf. [122]).

Definition 2.1. An XML *schema graph* is defined as a 5-tuple $\langle \Sigma, \Psi, s, m, \rho \rangle$ where Σ is an alphabet of node types, $\rho \in \Sigma$ is the root node type, $\Psi \subseteq \Sigma \times \Sigma$ is a set of directed edges between node types, for each $\psi \in \Psi$, $s(\psi) \in \{\text{ONCE}, \text{OPT}, \text{MULT}\}$ denotes the cardinality of ψ, and for each $\sigma \in \Sigma$, $m(\sigma) \in \{\{\texttt{string}\}\}$ denotes the domain of σ's content. ■

[1] Note that a DTD does not explicitly specify the root element type of a document. Instead, the root element type is specified in the DOCTYPE declarations of documents conforming to a DTD.

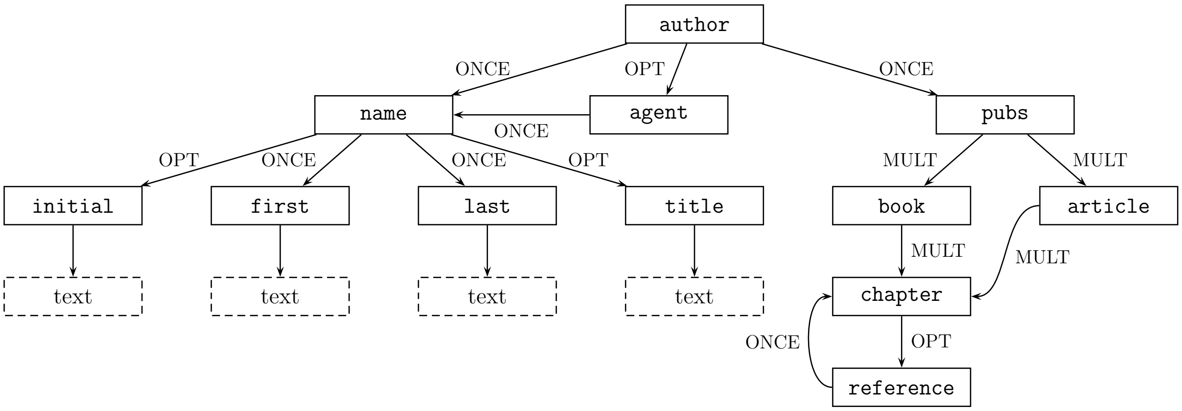


Figure 2.2: An XML schema graph

The semantics of this definition are as follows: An edge $\psi = \langle \sigma_1, \sigma_2 \rangle \in \Psi$ denotes that a node of type σ_1 may contain a node of type σ_2. $s(\psi)$ denotes the cardinality of the containment represented by this edge: If $s(\psi) = \text{ONCE}$, then a node of type σ_1 must contain exactly one node of type σ_2. If $s(\psi) = \text{OPT}$, then a node of type σ_1 may optionally contain a node of type σ_2. If $s(\psi) = \text{MULT}$, then a node of type σ_1 may contain zero or more nodes of type σ_2. $m(\sigma)$ denotes the domain of the text content of a node of type σ, represented as the set of all strings that may occur inside such a node.

Figure 2.1 shows an example of a schema represented as a simplified DTD; Figure 2.2 shows the same schema represented as a schema graph. Note how the node types `initial`, `first`, `last` and `title` each contain an edge to a node with a dashed outline, which is referred to as a *domain node*. The domain nodes represent the domain of the text content of their parent nodes. As can be seen, all four nodes may contain arbitrary text as their content (denoted as “text”). The other node types in the schema may not contain any text content and are therefore drawn without domain nodes.

When translating a DTD or an XML Schema into the graph representation, attributes are always assigned a cardinality of either ONCE or OPT, corresponding to mandatory and optional attributes, respectively. Elements, on the other hand, may occur with any of the three cardinalities, since both DTD and XML Schema allow for the specification of elements with exactly one, zero or one, or multiple occurrences. In addition to these three cases, XML Schema allows a more fine-grained specification of the number of occurrences of an element. This is handled by assigning a cardinality of MULT whenever the XML Schema definition allows for an element to occur more than once.

2.2 Query Model

The query evaluation techniques proposed in this work support a query model that consists of a subset of XPath [24], referred to as XQ. This section first describes the XPath primitives that are supported in XQ. Then, a tree-pattern representation of XQ queries is defined. This representation will be used throughout this thesis.

2.2.1 XQ

Definition 2.2. An *XQ query* is an expression that conforms to the following grammar such that $\sigma \in \Sigma$ is a node type, c_{str} is a string constant and c_{num} is a numeric constant.

$$
\begin{aligned}
XQ &:= \texttt{/}PATH \\
PATH &:= STEP \mid PATH\texttt{/}STEP \mid PATH\texttt{//}STEP \mid PATH\texttt{/self::}STEP \\
STEP &:= NTEST\ \texttt{[}PPRED\texttt{]}?\ \texttt{[}VC\texttt{]}? \\
NTEST &:= \sigma \mid * \\
PPRED &:= PATH \mid \texttt{(}PPRED\,\texttt{and}\,PPRED\texttt{)} \mid \texttt{(}PPRED\,\texttt{or}\,PPRED\texttt{)} \mid \texttt{not(}PPRED\texttt{)} \\
VC &:= TC \mid NC \mid\mid \texttt{(}VC\,\texttt{and}\,VC\texttt{)} \mid \texttt{(}VC\,\texttt{or}\,VC\texttt{)} \mid \texttt{not(}VC\texttt{)} \\
TC &:= \texttt{.}\ \theta_{\text{str}}\, c_{\text{str}} \mid \texttt{starts-with(.,}c_{\text{str}}\texttt{)} \mid \texttt{contains(.,}c_{\text{str}}\texttt{)} \mid \ldots \\
NC &:= \texttt{.}\ \theta_{\text{num}}\, c_{\text{num}} \\
\theta_{\text{str}} &:= \texttt{=} \mid \texttt{!=} \\
\theta_{\text{num}} &:= \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{!=}
\end{aligned}
$$

■

As shown in the grammar, XQ queries are absolute location paths ($PATH$) consisting of child (`/`), descendant (`//`) and self (`/self::`) steps. Each step consists of a node test ($NTEST$), either for a node type $\sigma \in \Sigma$ or the wildcard $*$. Additionally, each step may optionally contain one or more predicates. The following two types of predicates are supported:

path predicates ($PPRED$) are relative location paths with the same restrictions as the absolute location paths.

value constraints (VC) are constraints on the content of a collection node. Two types of value constraints are supported:

- **textual constraints** (TC) compare the textual content of a node to a string constant (c_{str}). This is done using a comparison based on equality, inequality or arbitrary string comparison functions such as `starts-with()` and `contains()`.
- **numeric constraints** (NC) treat the content of a node as a number and perform a comparison to a numeric constant (c_{num}) based on the operators $<$, $\leq$, $=$, $\geq$, $>$, and $\neq$.

For both types of predicates, arbitrarily nested conjunction, disjunction and negation are supported. To keep the query model simple, conjunction and disjunction involving two predicates of different types (i.e., a path predicate and a value constraint) is not directly supported, although this can easily be emulated by inserting additional self steps.

The semantic of each of the primitives present in XQ is defined to be identical to its XPath counterpart. In particular, this means that if a step contains two predicates (a path predicate and a value constraint), then the implied semantics are a conjunction of these predicates. Also, as in XPath, XQ steps return nodes in document order.

The primitives supported in XQ have been chosen based on two goals:

- The first goal is to include the most commonly used features of XPath in order to maximize the practical applicability of the proposed techniques. These features include nested path queries with predicates, conjunction, disjunction and negation, all of which are supported in XQ.

- The second goal is to focus on features of XPath that can be supported efficiently in a distributed fashion. Since some of the more advanced features of XPath – such as the far-reaching `following` or `preceding` axes – would have severely limited the opportunity for distributing and parallelizing query evaluation, these features have been excluded from XQ.

XQ represents a reasonable trade-off between these two goals. It supports a rich and useful class of queries. This is is supported by a wealth of related work based on query models that are largely equivalent to XQ (e.g., [39]) or even more restrictive (e.g., [31]).

As previously pointed out by Zhang et al. [142, 141], restricting the query model to self, child, and descendant axes is reasonable, as the path expressions in the XQuery use cases [35] consist predominantly of these axes. Furthermore, there exists some work on rewriting certain XPath axes to other axes that are easier to support [113, 114]. These techniques might be useful for supporting a larger class of queries by rewriting axes that are not directly supported in XQ, corroborating the usefulness of XQ as a model for expressing realistic queries.

XQ queries are not only commonly used on their own, but they also represent an important building block of more complex XQuery [26] queries and can be extracted from these queries in many cases [107, 74, 106]. This makes it possible to leverage the techniques presented in this thesis in the context of XQuery evaluation.

In addition to its expressiveness, XQ can be supported efficiently in a distributed system using the techniques proposed in this thesis. This has made it possible to propose a coherent, end-to-end strategy for evaluating XQ queries in a distributed fashion.

2.2.2 Tree Patterns

Rather than relying on the textual representation of XQ presented above, for query processing and optimization, it is more convenient to use a structured representation. For this work, a tree representation was chosen that is inspired by existing work on tree pattern queries [32, 141, 70].

Whereas traditional tree pattern queries support location paths with path predicates and conjunction, this work expands this model by adding support for value constraints, disjunction and negation. This makes it possible to express all of XQ in a convenient tree pattern representation, which will form the foundation of the query evaluation techniques presented in this thesis.

Traditionally, tree patterns are defined as an un-ordered tree consisting of *pattern nodes* with node tests and edges associated with XPath axes. To accommodate predicates with arbitrary combinations of conjunction, disjunction and negation, nodes of a second type are introduced, referred to as *logic nodes*. This leads to the following definition of tree patterns.

Definition 2.3. A *tree pattern* is defined as the 9-tuple $\langle N, L, r, E, \nu, c, \varepsilon, \lambda, T\rangle$ where N is a set of pattern nodes, L is a set of logic nodes, and $E \subseteq (N \cup L) \times (N \cup L)$ is a set of edges. $\langle (N \cup L), r, E\rangle$ is required to be an un-ordered tree rooted at $r \in N$ (the root node). $T \subseteq N$ denotes the set of extraction points.

For each logic node $l \in L$, $\lambda(l) \in \{\wedge, \vee, \neg\}$ determines whether this node represents a conjunction, disjunction, or negation.

If Σ is a set of node types, then for each pattern node $n \in N$, $\nu(n) \in \Sigma \cup \{*\}$ denotes the node test and $c(n)$ determines the value constraint. For each edge $e = \langle x, n\rangle \in E$ with $n \in N$, $\varepsilon(e) \in \{\texttt{/}, \texttt{//}, \texttt{/self::}\}$ determines the axis. ■

Each pattern node in a tree pattern may have at most one child node. Thus, for each pattern node $n \in N$ exactly one of the following must hold:

- $\forall x \in (N \cup L) : \nexists \langle n, x\rangle \in E$ (n is a leaf node with no children),
- $\exists$ unique $n' \in N : \langle n, n'\rangle \in E$ (n has a single pattern node child $n' \in N$), or
- $\exists$ unique $l' \in L : \langle n, l'\rangle \in E$ (n has a single logic node child $l' \in L$).

A logic node l with $\lambda(l) = \neg$ must have exactly one child node, which may be a pattern node or a logic node: $\exists$ unique $x \in (N \cup L) : \langle l, x\rangle \in E$. Similarly, if $\lambda(l) = \wedge$ or $\lambda(l) = \vee$,

l must have at least two child nodes, which may be any combination of pattern and logic nodes: $\exists x_1, x_2 \in (N \cup L), x_1 \neq x_2 : \langle l, x_1 \rangle, \langle l, x_2 \rangle \in E$.

All logic nodes l on the path from the root of the tree pattern r to an extraction point $t \in T$ must have $\lambda(l) = \wedge$ (i.e., there must not be any $\vee$ or $\neg$ logic nodes on the path from the root to an extraction point). This ensures that a pattern match will assign a node from the collection to each pattern node that is designated as an extraction point.

For notational convenience, for each node $x \in (N \cup L)$, parent(x) denotes the node $x' \in (N \cup L)$ such that $\langle x', x \rangle \in E$. Similarly, for each pattern node $n \in N$, child(n) denotes the node $x' \in (N \cup L)$ such that $\langle n, x' \rangle \in E$. For each node $x \in (N \cup L)$, children(x) denotes the set of nodes $\{x' \in (N \cup L)\}$ such that $\langle x, x' \rangle \in E$.

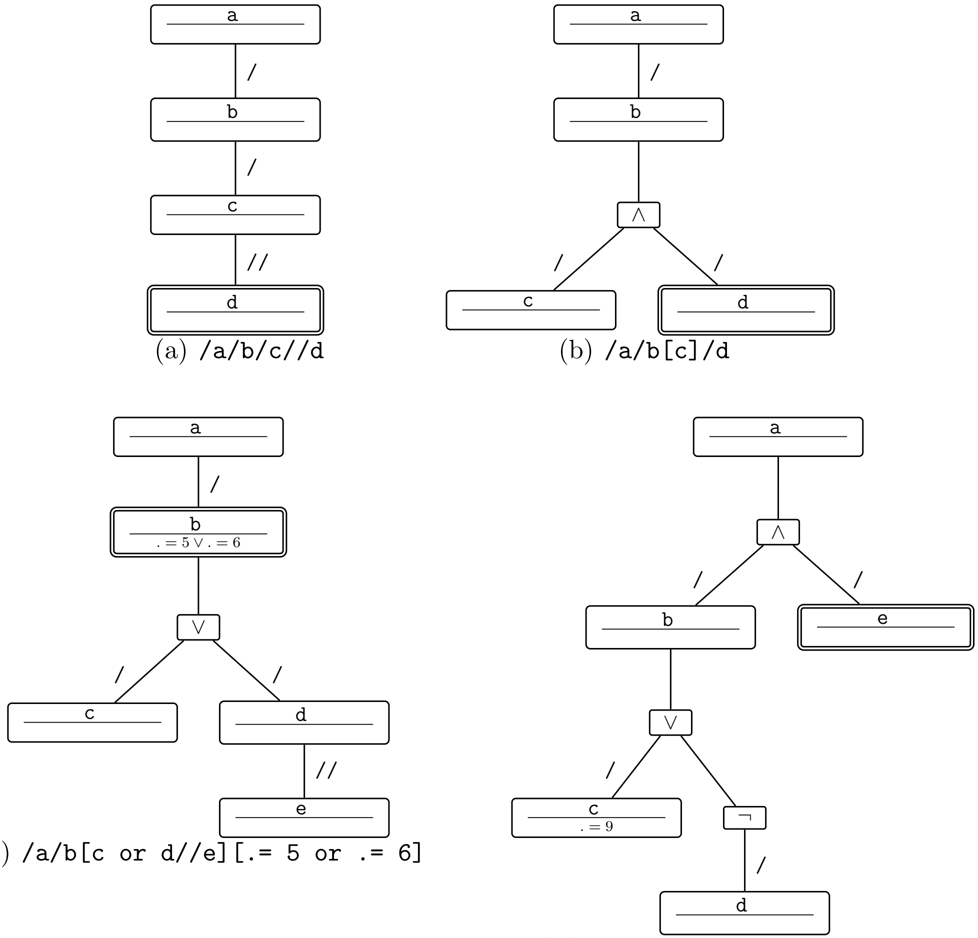
Since the intricacies of evaluating textual and numeric value constraints on a candidate node from the collection are beyond the scope of this thesis, the structure of $c(n)$ is not discussed further. It is important to note, however, that it consists of arbitrary conjunctions, disjunctions, or negations of textual and numeric value constraints.

It is interesting to note that, in addition to XQ queries, tree patterns can also be used to express queries with multiple extraction points. While this feature could be useful for supporting a larger class of queries, the focus of this work is on queries with a single extraction point. However, as will be shown in Chapter 5, tree patterns representing subqueries resulting from vertical fragmentation frequently contain multiple extraction points.

In this thesis, the tree pattern representation of a query is referred to as a *query tree pattern* (QTP). Figure 2.3 contains a few simple examples, which are used to illustrate the semantics of tree patterns.

- Figure 2.3(a) shows the QTP representation of a simple, linear path expression without predicates. As can be seen, each pattern node contains a node test (shown above the dividing line) and the edges are annotated with the axis of each step (`/` and `//`). The extraction point of the query (the pattern node with the node test `d`) is denoted by a double outline.

- Figure 2.3(b) introduces a path predicate. Note the implicit conjunction involving the path predicate (`c`) and the last step in the outermost path (`d`).



(a) `/a/b/c//d`

(b) `/a/b[c]/d`

(c) `/a/b[c or d//e][.= 5 or .= 6]`

(d) `/a[b[c[.= 9] or not(d)]]/e`

Figure 2.3: Tree pattern examples

- Figure 2.3(c) shows a query that contains disjunction both in a value constraint (shown below the dividing line in pattern node `b`) and in a path predicate (represented as a logic node labeled ∨).

- Figure 2.3(d) shows a more complex example with nested conjunction, disjunction, and negation.

The queries shown in Table 2.1 will be used as running examples throughout the remainder of this thesis. All of these queries correspond to the schema shown in Figure 2.2.

For the purpose of illustration, Figure 2.4 shows the QTP representations of queries q_1 through q_6.

2.2.3 Tree Pattern Matches

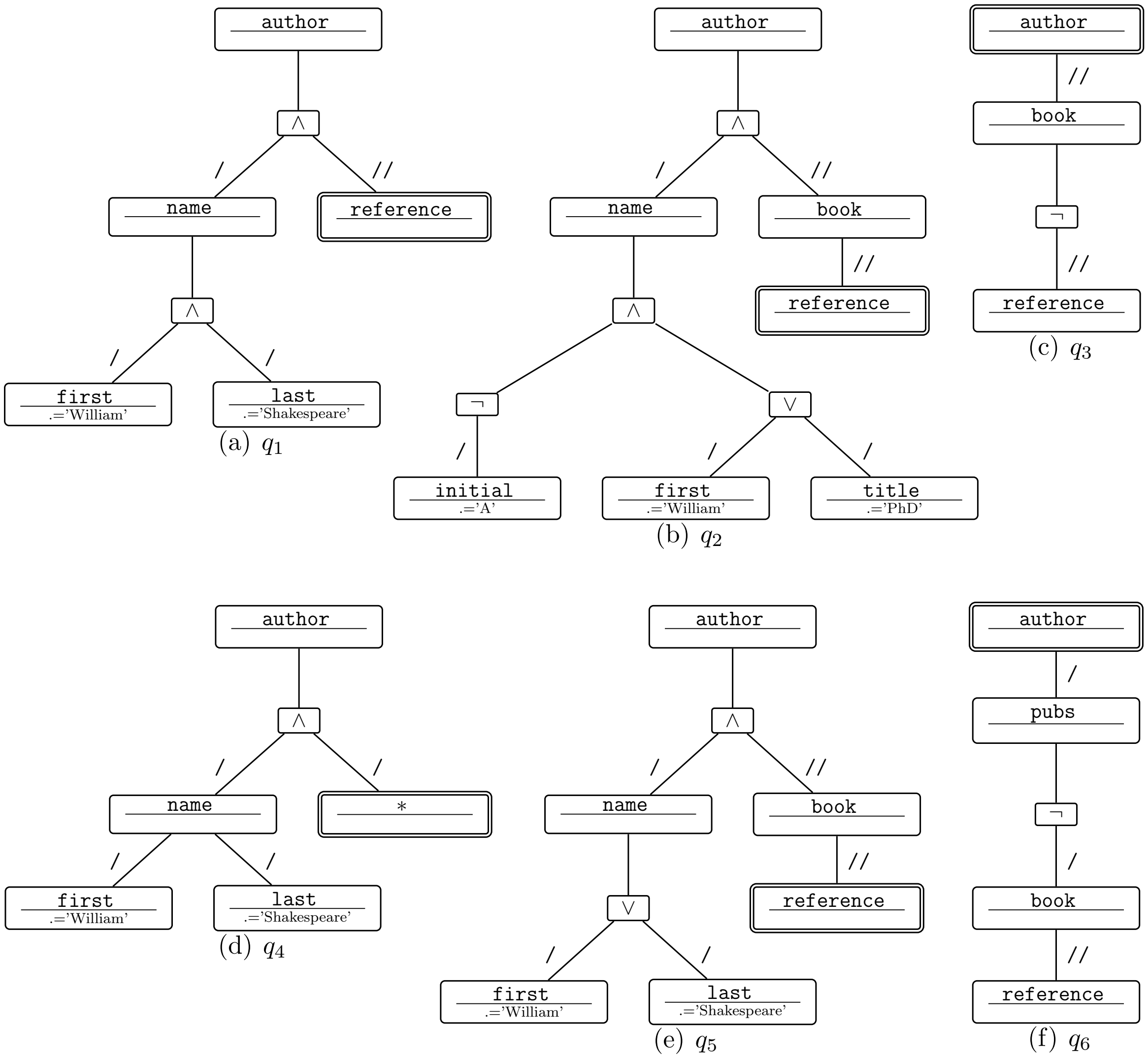
Tree pattern evaluation generally begins at the root of the pattern and assigns a node from the XML collection to each pattern node such that all node tests, value constraints, and structural constraints (expressed as axis relationships) are satisfied.

While this matching is straightforward for pattern nodes, logic nodes require special attention. When encountering a logic node labeled $\wedge$, all branches have to be matched. This is the same behaviour seen in traditional tree pattern matching techniques. For logic nodes labeled $\vee$, only one branch has to be matched and a pattern matching technique might use short-circuit evaluation to avoid processing the other branches. For $\neg$ nodes, the (single) branch is evaluated and the Boolean result is inverted, i.e., the branch is treated as satisfied if there is no match and not satisfied if there is a match.

Once a complete match for a tree pattern has been found, a result tuple is generated that consists of only those collection nodes that are associated with a pattern node that is

q_1	/author[name[first[.= ’William’] and last[.=’Shakespeare’]]]//reference
q_2	/author[name[first[.= ’William’] or title[.=’PhD’]][not(initial[.= ’A’])]]//book//reference
q_3	/author[.//book[not(.//reference)]]
q_4	/author[name[first[.= ’William’] and last[.=’Shakespeare’]]]/*
q_5	/author[name[first[.= ’William’] or last[.=’Shakespeare’]]]//book//reference
q_6	/author[pubs[not(book//reference)]]
q_7	/author[name[first[.= ’William’] and last[.=’Shakespeare’]]]//book//reference
q_8	/author[.//name[first[.= ’William’] and last[.=’Shakespeare’]]]//book//reference
q_9	/author[name[first[.= ’William’] and last[.=’Shakespeare’]]]//pubs//reference
q_{10}	/author[.//chapter]
q_{11}	/author[name[last[.=’Shakespeare’]]]//reference

Table 2.1: Example queries



(a) q_1

(b) q_2

(c) q_3

(d) q_4

(e) q_5

(f) q_6

Figure 2.4: QTP representation of queries q_1, q_2, q_3, q_4, q_5, and q_6

designated as an extraction point. Since $\vee$ and $\neg$ logic nodes are disallowed on the path from the root of the pattern to an extraction point, for any pattern match, there will be collection nodes assigned to all extraction points in the pattern.

When returning the collection nodes assigned to extraction points in the pattern, there are two general approaches one can take. First, it is possible to return references to these nodes, which can then be retrieved from the sites holding the corresponding fragments. This is a good approach when large sub-trees are selected, which are then processed further (for example, when using the techniques presented here to evaluate XPath expressions occurring within the context of a more complex XQuery FLWOR expression). The second option is to materialize the nodes along with their text content and then ship the result. This has the advantage that no additional accesses to the fragments are necessary. The distributed query evaluation techniques presented in this thesis are designed to work with either of these approaches. The primary focus, however, will be on the second approach.

2.2.3.1 Order of Tree Pattern Matches

Evaluating a tree pattern over a collection yields a sequence of tree pattern matches. To conform to the XPath semantics [24], tree pattern matches must be returned in document order. For QTPs with a single extraction point this means that the nodes matched to the extraction point will be returned in document order. For a QTP with multiple extraction points, however, it is not generally possible to return query results such that the collection nodes matched to all extraction points are returned in document order. In this case, one extraction point is designated as the *ordering extraction point* and result tuples are returned such that the collection nodes matched to the ordering extraction point are returned in document order.

Chapter 3

Related Work

This chapter presents a discussion of related work. The main focus of this discussion is on distributed query evaluation techniques for XML data. There is a significant amount of existing work on centralized XML query evaluation. For brevity, only a short overview of these techniques is given here, concerned primarily with techniques that are directly related to the distributed approach presented in this thesis, and with techniques that are employed by this distributed approach for the evaluation of sub-queries over individual fragments. For a more detailed discussion of centralized XML query processing, the reader is referred to Gou and Chirkova's survey [58].

In the area of distributed query evaluation over relational data, there exists a vast body of existing work. The discussion given here focuses on techniques that are directly related to the techniques for XML data presented in this thesis. See Kossmann's survey [86] and Özsu and Valduriez's book [115] for a more detailed discussion of this area of work.

The remainder of this chapter is structured as follows: In Section 3.1, techniques for fragmenting collections are discussed. Section 3.2 then describes query evaluation techniques that can be applied to fragmented and distributed collections. Finally, Section 3.3 discusses cost-based query optimization techniques and methods for cost estimation.

3.1 Fragmenting Collections

The key contribution of this thesis is a distributed query evaluation strategy that can be applied to collections that have been distributed across a cluster of machines. Before a collection can be distributed in this fashion, it is first necessary to fragment it. This section first gives a brief overview of relational fragmentation techniques (Section 3.1.1), by whose semantics the fragmentation approaches used in this work are inspired. Then, the various techniques for fragmenting XML data are discussed and compared to the fragmentation technique used in this work in Section 3.1.2.

3.1.1 Fragmenting Relational Data

While there exists a large body of work on fragmenting relational data (see [115] for an overview), the most common approach is based on two types of fragmentation: horizontal fragmentation based on selections that determine how rows are mapped to fragments, and vertical fragmentation where projection is used to determine how columns are mapped to fragments. These approaches are orthogonal and can be used together, yielding a hybrid fragmentation.

To illustrate how horizontal and vertical fragmentation work in the context of relational data, consider the table shown in Figure 3.1, which contains information about students. To horizontally fragment this table, rows are selected for each fragment. This can be done by specifying a selection predicate for each fragment. Figure 3.2 shows a possible horizontal

Student			
ID	Name	Phone	Email
1001	John Doe	519-555-1234	jdoe@example.com
1002	Jane Doe	519-555-1010	j2doe@example.com
1003	John Smith	416-555-0101	jsmith@example.org
1004	Jane Smith	905-555-0987	jsmith@example.com

Figure 3.1: Original table

$\sigma_{\text{ID}\leq 1002}$(Student)			
ID	Name	Phone	Email
1001	John Doe	519-555-1234	jdoe@example.com
1002	Jane Doe	519-555-1010	j2doe@example.com

$\sigma_{\text{ID}>1002}$(Student)			
ID	Name	Phone	Email
1003	John Smith	416-555-0101	jsmith@example.org
1004	Jane Smith	905-555-0987	jsmith@example.com

Figure 3.2: Horizontally fragmented table

fragmentation of this table. As can be seen, the first fragment contains students whose ID is less than or equal to 1002, whereas the second fragment contains students whose ID is greater than 1002. It is important to note that with horizontal fragmentation, the schema of each fragment (as expressed by the set of attributes) is identical to the schema of the unfragmented table.

When vertically fragmenting the same table (as shown in Figure 3.3), a set of columns is chosen for each fragment. Thus, a vertical fragmentation can be specified by a set of projection operations (one for each fragment). Note that this causes each fragment to hold only a subset of the attributes of the unfragmented table. The rows of the original table, however, are spread across all of the individual fragments. To ensure that the unfragmented table can be reconstructed and to ensure that no information is lost, it is necessary to replicate the key (corresponding to column ID in the example given here) in each fragment. Without replicating the key it would not be possible to infer how the rows in the individual fragments correspond to each other and, therefore, the decomposition would not be lossless.

While the significant differences in data models make it impossible to directly apply these fragmentation techniques to XML data, they inspire the techniques developed in this thesis. As will be discussed in more detail in Chapter 4, horizontal fragmentation of XML collections, similar to its relational counterpart, uses selection operations to define

$\pi_{\text{ID,Name}}(\text{Student})$	
ID	Name
1001	John Doe
1002	Jane Doe
1003	John Smith
1004	Jane Smith

$\pi_{\text{ID,Phone,Email}}(\text{Student})$		
ID	Phone	Email
1001	519-555-1234	jdoe@example.com
1002	519-555-1010	j2doe@example.com
1003	416-555-0101	jsmith@example.org
1004	905-555-0987	jsmith@example.com

Figure 3.3: Vertically fragmented table

individual fragments, each of which correspond to the same schema. Similarly, vertical fragmentation for XML partitions the schema, resulting in fragments that correspond to different portions of the overall schema.

3.1.2 Fragmenting XML Data

Due to the more flexible, tree-structured nature of XML data, there are many different possibilities for fragmenting XML data. These approaches can be classified into two main categories: *ad-hoc fragmentation*, which allows arbitrary cuts to be made in the XML document trees, and *structure-based fragmentation*, in which fragments are defined based on characteristics of the schema or the data. While the work presented in this thesis follows the latter approach, in this section, both approaches are discussed: Ad-hoc fragmentation is described in Section 3.1.2.1, and structure-based fragmentation is described in Section 3.1.2.2.

After discussing both approaches to fragmentation, Section 3.1.2.3 describes techniques that take query workload characteristics into account to find a suitable fragmentation. While most of the existing work on fragmenting XML collections focuses on static approaches, in which the fragmentation, once determined, never changes, there are several techniques that dynamically refragment a collection to adapt to changing query workloads.

3.1.2.1 Ad-hoc Fragmentation

Ad-hoc fragmentation is a flexible fragmentation model that does not rely on an explicit fragmentation specification. Instead, it allows an XML collection to be fragmented by arbitrarily cutting edges in the individual documents.

One approach that follows the ad-hoc fragmentation model is Active XML [2, 3, 4, 5], which represents cross-fragment edges as calls to remote functions. Figure 3.4 shows an example of an Active XML fragment that has a call to the remote function `getPubs()` embedded.

When the remote function call is activated, the data corresponding to the remote fragment is retrieved and is then available for local query processing. Figure 3.5 illustrates this. As can be seen, the function call has been replaced with a fragment containing information about this author's publications. Using this model, Active XML provides a flexible mechanism for describing fragmented collections of XML data. As with ad-hoc fragmentation approaches in general, Active XML is particularly well suited for describing how multiple sources of data can be integrated into a single XML view. This is in contrast

```
<author>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
  <call fun="getPubs(’J.\ Doe’)"/>
</author>
```

Figure 3.4: Active XML document

```
<author>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
  <pubs>
    <book>...</book>
    <article>...</article>
    <book>...</book>
  </pubs>
</author>
```

Figure 3.5: Active XML document after activating `getPubs()`

to the work presented in this thesis, which uses fragmentation as a means to spread query processing throughout a distributed system.

Based on this work, Abiteboul et al. [6] present a technique for ensuring that an Active XML document conforms to a specified type. This is achieved by reasoning about how the types of individual document fragments affect the overall type of a document, thereby combining Active XML with a more structure-based fragmentation approach.

Cong et al.'s [39, 33, 40] work on partial query evaluation is also based on ad-hoc fragmentation although their single-document data model allows the authors to infer certain structural relationships between fragments, which can then be used for distributed query optimization. Therefore, this work can be considered a hybrid case that has certain structure-based characteristics.

Deutsch and Tannen [43] describe a technique for publishing an XML view over existing relational and XML data. Their model uses XQuery expressions to map between the published view and the (possibly redundant) data sources. While the authors do not describe their work in a distributed context, they present a query rewriting technique that could be used to answer queries in a data integration scenario. When distributing to improve scalability, their technique seems less useful since the rewriting procedure is relatively complex and the complete freedom given by an XQuery-based fragmentation model with overlapping fragments would further increase the already large search space encountered when fragmenting for a given workload.

The representation of cross-fragment edges as pairs of proxy nodes is a technique that has been used successfully to fragment XML document trees onto pages in native XML database system such as Natix [30, 49, 78] and Timber [75, 116], albeit at a much smaller level of granularity than in the work presented here.

In summary, it can be observed that ad-hoc fragmentation offers great flexibility in how a collection can be distributed, which makes it a good candidate for a data integration system. This flexibility, however, comes at the cost of decreased opportunity for distributed query optimization, making this choice unsuitable for this work. Nevertheless, some of the techniques that have been proposed for ad-hoc fragmentation (such as a proxy-based representation of cross-fragment edges) are equally applicable to the structure-based scenario.

3.1.2.2 Structure-Based Fragmentation

Structure-based fragmentation is based on the concept of fragmenting a collection based on some property of the schema or the data themselves. As in the relational context, it is possible to distinguish between *horizontal fragmentation*, which defines fragments by *selecting* subsets of the collection, and *vertical fragmentation*, in which fragments are defined by *projecting* to different parts of the schema. In addition to these options, it is possible to define a *hybrid fragmentation* by concatenating selection and projection steps. A key advantage of structure-based fragmentation is that it yields a succinct specification of how a collection has been fragmented. As will be shown in this thesis, this specification is highly valuable for the efficient evaluation of queries over the fragmented collection. Because of this, structure-based fragmentation is particularly well suited when fragmenting for the purpose of improving query performance, which is why the work presented in this thesis follows a structure-based approach.

One of the first attempts to transfer the relational concepts of horizontal and vertical fragmentation to the realm of XML was made by Ma and Schewe [93]. However, their definition of vertical fragmentation is limited to elements whose content is a sequence of other elements. Under these constraints, it is straightforward to extend the relational definition of vertical fragmentation by treating the containing element type as a relation that contains attributes corresponding to the contained element types. Analogously to the relational case, it is then possible to simply project to subsets of the contained elements. The authors also assume a single-document collection, which means that a horizontal fragmentation step always has to be preceded by an implicit vertical fragmentation step. In addition, their approach is based on modifying the schema by renaming elements and rearranging their nesting. Therefore, unlike later techniques, it is not transparent, and it requires queries to be formulated explicitly for a particular fragmentation specification.

Bremer and Gertz [31, 57] present another mechanism for specifying a vertical fragmentation of XML data. They call such a specification a Repository Guide. In a Repository Guide, a fragment is defined by a selection path expression identifying the root nodes of the sub-trees contained (referred to as inclusion paths), as well as a set of exclusion paths representing nodes whose descendants are excluded from the fragment. The set of fragments is required to be both disjoint and complete. The authors argue that this approach can be expanded to horizontal fragmentation by allowing branching and value constraints in the defining path expressions. However, this would make it difficult to enforce completeness and disjointness.

Andrade et al. [12] expand Bremer's method for specifying vertical fragmentation by adding explicit support for horizontal and hybrid fragmentation. They define each horizontal fragment by giving a selection predicate in the form of a Boolean path expression with value constraints, in a way that is similar to how horizontal fragmentation is defined in this thesis. These predicates are then used to determine whether a given document is part of a given fragment. The predicates are required to cover all documents (completeness) and be mutually exclusive (disjointness). The authors also make the observation that by nesting horizontal and vertical fragmentation, both single-document and multiple-document scenarios can be accommodated.

In addition to predicate-based horizontal fragmentation, Kido et al. [79] introduce a novel definition of vertical fragmentation that is based on partitioning the schema graph, rather than on inclusion and exclusion paths. This definition closely resembles the way

Technique	Collection type		Fragmentation type	
	Single doc	Mult. doc	Vertical	Horizontal
Ma and Schewe	✓	×	renaming/renesting nodes	after vert. frag. only
Bremer and Gertz	✓	✓	incl./excl. paths	constraints in excl. paths
Andrade et al.	✓	✓	incl./excl. paths	predicates
Kido et al.	✓	✓	partitioning of schema	predicates
This thesis	✓	✓	partitioning of schema	predicates

Table 3.1: Comparison of structure-based XML fragmentation techniques

vertical fragmentation is defined in this thesis.

Rusu et al. [119] extend structure-based fragmentation approaches for XML data by adding explicit support for storing multiple versions of the same document. As the authors point out, this is particularly useful in a data warehouse, where historic versions of data need to be preserved.

While not directly related to fragmentation, Marian et al. [102] propose a technique that improves query performance by projecting away irrelevant portions of an XML collection. The goal of this technique is to reduce the size of the relevant portion of the collection and thus be able to process the query in main memory. Unlike the other fragmentation techniques discussed here (and unlike fragmentation as specified in this thesis), there is no attempt to preserve the entire collection. Thus, Marian et al.'s approach somewhat resembles techniques that generate materialized views of an XML collection and then answer queries based on these views (e.g., [17, 129]).

Table 3.1 shows an overview of the structure-based fragmentation techniques presented in this section. As can be seen, most techniques support both single-document and multiple-document collections, with the exception being the early technique by Ma and Schewe [93], which supports single-document collections only. Due to this limitation, this technique offers only limited support for horizontal fragmentation, requiring a prior, implicit vertical fragmentation step. At the same time, since this technique changes the names and nesting of elements during fragmentation, queries need to be reformulated before they can be evaluated over a fragmented collection. Later techniques offer full support for vertical fragmentation based on inclusion and exclusion paths (Bremer and Gertz [31, 57] and

Andrade et al. [12]) or based on a partitioning of the schema (Kido et al. [79]). Support for horizontal fragmentation is also present, either based on value constraints in exclusion paths (Bremer and Gertz [31, 57]) or based on a set of predicates (Andrade et al. [12] and Kido et al. [79]).

3.1.2.3 Fragmentation Based on Query Workloads

While structure-based fragmentation yields a fragmentation specification that can be exploited to improve the performance of query evaluation over fragmented collections, to obtain the best performance, it is helpful to tailor the fragmentation to the query workload. This section presents an overview of techniques for doing this. While in theory it is possible to determine the best possible fragmentation by exhaustively enumerating all possibilities, due to the large size of the search space, heuristics are usually favoured.

Based on their model of horizontal fragmentation, Ma and Schewe [94] propose an outline of a cost-based heuristic for determining the best horizontal fragmentation for a given query workload. Unlike the work presented in this thesis, which aims to consider all components of the cost of query evaluation, the cost model presented in Ma and Schewe's paper focuses on communication cost.

Based on their definition of horizontal and vertical fragmentation, Kido et al. [79] describe how a suitable fragmentation for a given workload can be obtained. Their strategy is based on first fragmenting the collection into a large number of small fragments, one corresponding to each simple path expression that can be extracted from the query workload. To obtain the desired number of fragments, some of these initial fragments are then combined. While the authors do not go into any detail on the cost model used to compare various ways of combining initial fragments, the authors point out that a complete enumeration of all possible fragmentations is not feasible. Instead, a greedy strategy and a strategy based on genetic algorithms are outlined.

Yu et al. [138, 128] describe two heuristic data allocation techniques that are designed to increase parallelism in query execution. The first technique, Path Schema based Path Instance Balancing (PSPIB), is based on the idea of a path schema, which represents label paths from the root of the document to one of its leaves. The nodes corresponding to a particular path schema are evenly distributed among all fragments in the system, with nodes occurring in more than one path schema being replicated in all fragments. The second technique, referred to as Node Schema based Node Round-Robin Balancing (NSNRR), on the other hand, places all nodes with the same node name in one fragment and allocates node names to fragments in a round-robin fashion. There is no replication with this technique.

Focusing on a specific query evaluation strategy using holistic twig joins (first proposed by Bruno et al. [32]), Machdi et al. [95, 96] describe a technique that can be used to cluster both documents and queries to enable parallel query evaluation. Unlike the fragmentation model used in this work, their model replicates data as necessary. Based on their fragmentation technique, the authors then discuss how this clustering can be refined dynamically to adapt to workload imbalances.

Kurita et al. [88] take this idea of dynamically adapting the fragmentation further. Initially, they create a vertical fragmentation consisting of fragments that are approximately equal in size by recursively inspecting the size of document sub-trees and splitting if necessary. During query processing, the processing load placed on each fragment is measured. Based on this information, fragments with high loads are then split and fragments with small loads are merged. This allows them to balance the loads placed on each site, which leads to improved query throughput. While this technique is based on a simple idea, its ability to cope with varying workloads is a considerable advantage. However, frequently rearranging large portions of the collection is likely to be expensive. In addition, the dynamic nature of the resulting fragments can have a negative impact on the locality of updates. Also, as with all ad-hoc fragmentation techniques, query optimization is more difficult since it is not generally possible to determine which part of the query corresponds to which of the fragments.

As can be seen, several approaches have been proposed to fragment and distribute XML collections based on a query workload. When fragmenting based on workload characteristics, it is particularly important that the characteristics of the query evaluation strategy are taken into account. Because of this, it is not possible to simply adapt an existing fragmentation strategy for this work. Instead, a technique needs to be tailored for the query evaluation strategy proposed in this thesis, which is presented in Chapter 8.

3.2 XML Query Evaluation

The problem of XML query evaluation has attracted a significant amount of attention in the research community and a large body of existing work exists in this area. This section focuses on the aspects of the existing work that are related to the problem addressed in this thesis. First, existing work related to the tree pattern query model used in this thesis is summarized (Section 3.2.1). Then, centralized query evaluation techniques are described (Section 3.2.2). Since the amount of work in this area is vast, only a brief overview of the most popular techniques is given and special focus is placed on techniques that share some commonality with the distributed techniques presented in this thesis (e.g., techniques that explicitly take fragmentation into account). Finally, the related work in the area of distributed XML query processing is discussed in detail (Section 3.2.3).

3.2.1 Tree Patterns as a Query Model

As discussed in Section 2.2, the query model used in this work (referred to as XQ) supports tree pattern queries with child and descendant axes, node tests and value constraints. Additionally, negation, conjunction, and disjunction are supported and pattern nodes may be designated as extraction points to allow for data selection.

Expressing queries as tree patterns is a well established technique, and much of the existing work in the areas of centralized (e.g., [32, 141]) and distributed XML query processing (e.g., [33, 39, 31]) employs this approach. While it is straightforward to transform simple, nested XPath queries into such tree patterns, the usefulness of this query model goes further than that. This is illustrated by Michiels et al. [106], who describe a formal procedure for extracting tree patterns from more complex XQuery expressions. Using this technique, it is possible to apply tree pattern-based query evaluation techniques to a wider range of queries.

3.2.2 Centralized Query Evaluation

The distributed query evaluation technique presented in this thesis works by decomposing a tree pattern query into multiple local sub-patterns, each of which is then evaluated independently over its corresponding fragment. Therefore, this work is independent of the centralized query evaluation techniques used at each site. The problem of evaluating tree pattern queries over centralized XML collections has been studied in great detail and many different solutions exist. While a complete discussion of all such techniques is beyond the scope of this thesis, for understanding the distributed techniques presented here, it is helpful to be familiar with the main approaches used in centralized query evaluation. This section presents an overview of these approaches, paying special attention to how logic nodes in a tree pattern can be accommodated.

Existing research on centralized tree pattern evaluation has yielded a large variety of techniques. Most of this work can be categorized into two main classes: *navigational* approaches (discussed in Section 3.2.2.1) and *structural join*-based approaches (discussed in Section 3.2.2.2). For a detailed discussion of the performance implications of choosing between these approaches, the reader to is referred to [103].

In addition to simple navigational and structural-join based query evaluation techniques, some techniques have been developed that take into account the storage layout of native XML database systems. In these systems, large documents are frequently stored in multiple portions (e.g., corresponding to disk pages). Thus documents stored in these systems are effectively fragmented. Section 3.2.2.3 describes these fragmentation-aware techniques and discusses the commonalities they share with the distributed query evaluation techniques proposed in this thesis.

3.2.2.1 Navigational Query Evaluation

Navigational approaches for tree pattern evaluation operate directly on the tree structure of the collection [20, 76, 84, 30, 66]. This is usually done using algebraic operators that, given a starting node in the collection and an XPath step, yield the nodes that are reachable via this step.

q_1	`/author[name[first[.= ’William’] and last[.=’Shakespeare’]]]//reference`
q_2	`/author[name[first[.= ’William’] or title[.=’PhD’]][not(initial[.= ’A’])]]//book//reference`

Table 3.2: Example queries

To illustrate this, consider Figure 3.6, which shows navigational query plans for queries q_1 and q_2 (given in Table 3.2). Both plans are based on the algebra presented in [30, 66].

The plan for q_1 starts by scanning the root nodes of the documents in the collection ($\text{scan}_{a_0:\text{root}}$). This scan yields a sequence of tuples, each of which contains one document root in attribute a_0. This sequence then reaches the operator $\Upsilon_{a_1:a_0/\texttt{author}}$ (unnest map), which computes the first step of the query. For each tuple received, this operator yields one tuple for each `author` node reachable via a child step from the node in attribute a_0. In the resulting tuple, the node that was reached through this step is returned in attribute a_1. The other steps in the query are similarly handled by Υ operators.

Predicates are handled by a selection operator (σ). Note that the selection predicate is expressed as a sub-plan, connected via a dotted line. The semantics of this are as follows. For each tuple t received from the left-hand side producer of the selection, the sub-plan on the right-hand side is evaluated. A sub-plan begins with a singleton scan operator ($\Box$), which yields a single tuple identical to t. Following this are the Υ and σ operators that evaluate the predicate. Finally the aggregate operator $\mathbb{A}_{\text{exists}}$ is reached, which yields true if its producer yields at least one tuple and false otherwise. Therefore, the result of $\mathbb{A}_{\text{exists}}$ is true if and only if the predicate is satisfied. This result is then used as the selection predicate for tuple t. This approach also works for nested predicates. As shown in Figure 3.6(a), this makes it possible to support conjunction.

To handle queries with negation and disjunction, additional logic operators can be inserted. These operate on Boolean values rather than tuples, which is indicated by dotted lines in Figure 3.6(b). As before, sub-plans for each predicate are evaluated. Their result tuples are then aggregated ($\mathbb{A}_{\text{exists}}$), which yields a Boolean result for each sub-plan. The Boolean results of multiple sub-plans are then combined by logic operators ($\vee$ and $\neg$) and finally used as the predicate for a selection (σ).

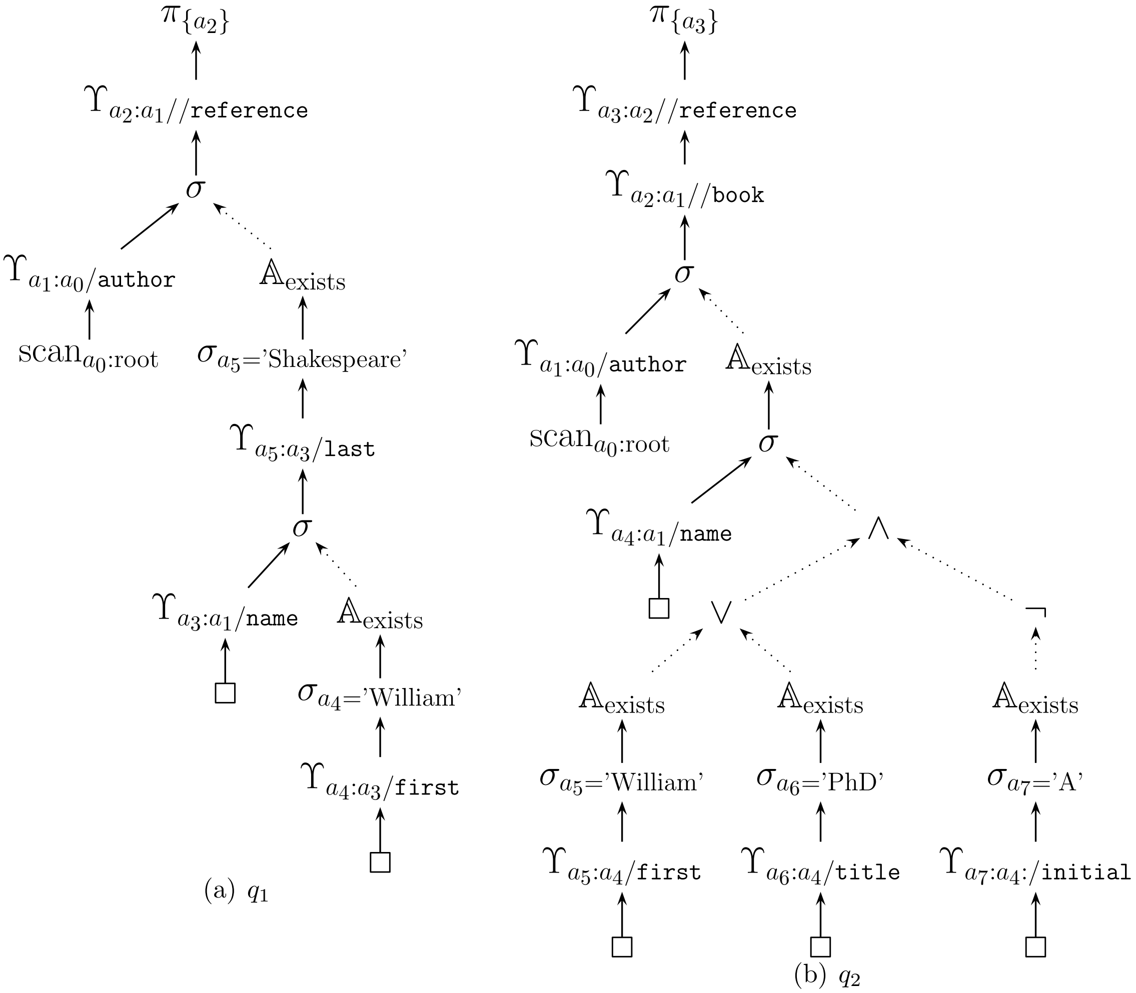

$\pi_{\{a_2\}}$
$\Upsilon_{a_2:a_1}$//reference
σ
$\Upsilon_{a_1:a_0}$/author
$\mathbb{A}_{\text{exists}}$
$\text{scan}_{a_0:\text{root}}$
$\sigma_{a_5=\text{'Shakespeare'}}$
$\Upsilon_{a_5:a_3}$/last
σ
$\Upsilon_{a_3:a_1}$/name
$\mathbb{A}_{\text{exists}}$
$\sigma_{a_4=\text{'William'}}$
$\Upsilon_{a_4:a_3}$/first
(a) q_1
$\pi_{\{a_3\}}$
$\Upsilon_{a_3:a_2}$//reference
$\Upsilon_{a_2:a_1}$//book
σ
$\Upsilon_{a_1:a_0}$/author
$\mathbb{A}_{\text{exists}}$
$\text{scan}_{a_0:\text{root}}$
σ
$\Upsilon_{a_4:a_1}$/name
$\wedge$
$\vee$
$\neg$
$\mathbb{A}_{\text{exists}}$
$\mathbb{A}_{\text{exists}}$
$\mathbb{A}_{\text{exists}}$
$\sigma_{a_5=\text{'William'}}$
$\sigma_{a_6=\text{'PhD'}}$
$\sigma_{a_7=\text{'A'}}$
$\Upsilon_{a_5:a_4}$/first
$\Upsilon_{a_6:a_4}$/title
$\Upsilon_{a_7:a_4:}$/initial
(b) q_2

Figure 3.6: Navigational plans for queries q_1 and q_2

At the root of the plan, the tuples are projected to contain only those attributes that correspond to extraction points in the query pattern. In both examples shown in Figure 3.6, only the attribute corresponding to `reference` nodes that match the pattern are returned (a_2 for query q_1 and a_3 for query q_2).

3.2.2.2 Structural Join-Based Query Evaluation

Structural join-based query evaluation techniques follow a markedly different approach [137, 139, 11, 62, 41, 32]. Instead of navigating the document tree, they perform linear scans of all nodes in a document (possibly filtered for a single node type) and then combine the results of such scans using joins that determine whether two nodes are in a particular structural relationship to each other. Usually, these joins rely on some kind of encoding of the document structure in the ID of each node or in a separate index. In addition, structural joins frequently exploit the order in which nodes are processed, thus reducing the number of comparisons that need to be made.

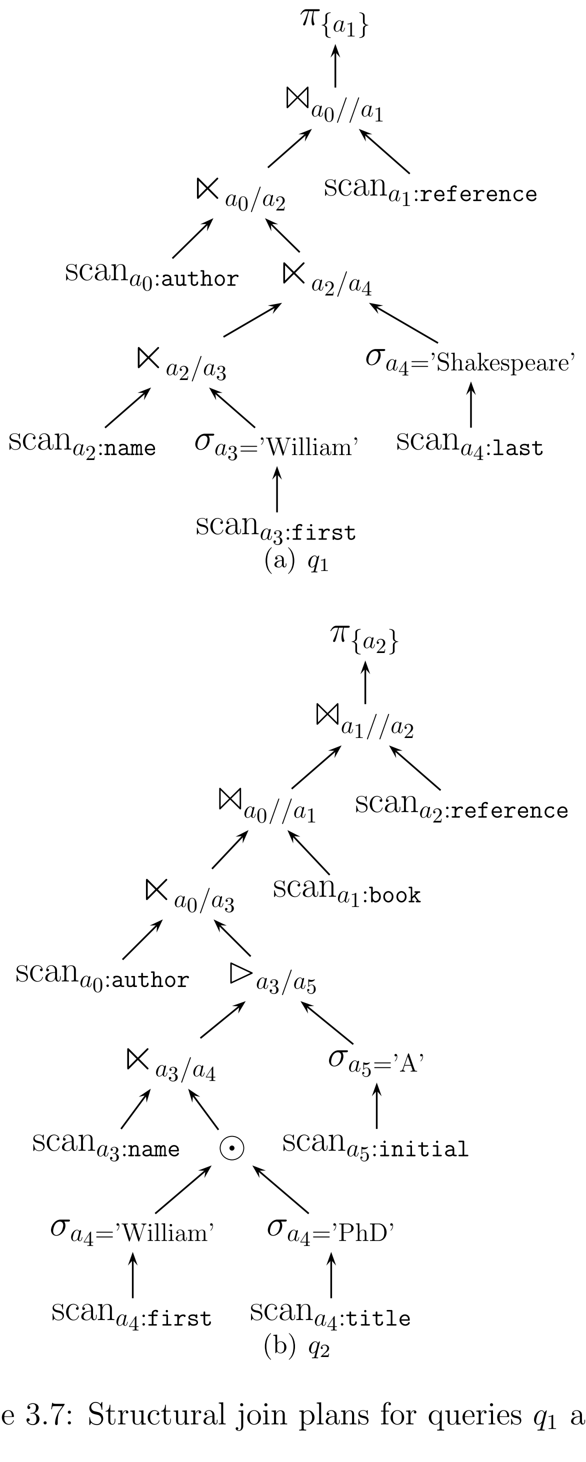
Figure 3.7 contains two examples for this approach, corresponding to queries q_1 and q_2, respectively. Both plans consist of scans and joins. $\text{scan}_{a_0:\texttt{author}}$, for example, scans all nodes of the type `author` and emits one tuple for each of them, with the `author` node in attribute a_0. The join operator $\bowtie_{a_0//a_1}$ proceeds as follows. Given a tuple $t_1 = [\ldots, a_0, \ldots]$ from the left-hand side producer and a tuple $t_2 = [\ldots, a_1, \ldots]$ from the right-hand side producer, the join emits a tuple $t = t_1 \cup t_2$ if and only if the node in a_1 is a descendant (denoted by `//`) of the node in a_0.

To evaluate predicates, semi-joins can be used instead of full joins. $\ltimes_{a_0/a_3}$ in the plan for q_1, for example, works as follows. Each tuple $t_1 = [\ldots, a_0, \ldots]$ from the left-hand side is passed on unchanged if and only if there is a tuple $t_2 = [\ldots, a_3, \ldots]$ on the right-hand side such that the node in a_3 is a child (denoted by `/`) of the node in a_0.

As with navigational approaches, supporting nested predicates connected by conjunction is straightforward. One can simply insert multiple semi-joins into the plan.

To accommodate disjunction, it is possible to use the merge operator (denoted as $\odot$), which emits tuples received from either side unchanged. The plan for query q_2 in Figure 3.7(b) shows an instance of this. Note how the producer sub-plans of $\odot$ are set up so that they emit tuples that contain nodes in the same attribute a_4. After merging both streams of tuples, this attribute is then used in a semi-join predicate.

To support negated predicates, an anti-join ($\triangleright$) can be used in place of the semi-join. The operator $\triangleright_{a_3/a_5}$, for example, proceeds as follows: Each tuple $t_1 = [\ldots, a_3, \ldots]$ received


$\pi_{\{a_1\}}$
$\bowtie_{a_0//a_1}$
$\ltimes_{a_0/a_2}$
$\text{scan}_{a_1:\texttt{reference}}$
$\text{scan}_{a_0:\texttt{author}}$
$\ltimes_{a_2/a_4}$
$\ltimes_{a_2/a_3}$
$\sigma_{a_4=\text{'Shakespeare'}}$
$\text{scan}_{a_2:\texttt{name}}$
$\sigma_{a_3=\text{'William'}}$
$\text{scan}_{a_4:\texttt{last}}$
$\text{scan}_{a_3:\texttt{first}}$
$\pi_{\{a_2\}}$
$\bowtie_{a_1//a_2}$
$\bowtie_{a_0//a_1}$
$\text{scan}_{a_2:\texttt{reference}}$
$\ltimes_{a_0/a_3}$
$\text{scan}_{a_1:\texttt{book}}$
$\text{scan}_{a_0:\texttt{author}}$
$\triangleright_{a_3/a_5}$
$\ltimes_{a_3/a_4}$
$\sigma_{a_5=\text{'A'}}$
$\text{scan}_{a_3:\texttt{name}}$
$\odot$
$\text{scan}_{a_5:\texttt{initial}}$
$\sigma_{a_4=\text{'William'}}$
$\sigma_{a_4=\text{'PhD'}}$
$\text{scan}_{a_4:\texttt{first}}$
$\text{scan}_{a_4:\texttt{title}}$

(a) q_1

(b) q_2

Figure 3.7: Structural join plans for queries q_1 and q_2

from the left-hand side is returned if and only if there is no tuple $t_2 = [\ldots, a_5, \ldots]$ from the right-hand side such that the node in a_5 is a child of the node in a_3.

3.2.2.3 Exploiting Fragmentation in Centralized Query Evaluation

Initially, research on XML database systems did not pay much attention to page boundaries during query evaluation. Instead, these systems operated entirely on logical document trees, without considering how these trees are stored. Assuming that document trees are stored on disk pages, as is commonly the case in the context of a native XML database system, this approach has the disadvantage that pages may be accessed out of order or even multiple times, leading to increased disk seek cost.

Several techniques have been proposed that take into account how documents are fragmented across multiple pages. While the fragments encountered by these techniques are generally much smaller than those seen when an XML collection is fragmented for the purpose of distribution across multiple nodes in a distributed system, it is nevertheless interesting to compare these query evaluation techniques to the techniques that can be applied in a distributed system.

Kanne et al. [77] present a navigational query evaluation technique that can process location path queries while avoiding the random I/O penalty associated with approaches that operate purely on the logical document tree. This is achieved by delegating all page accesses to a single scan operator. All other operators are only allowed to navigate within the current page. Since this does not, in general, yield enough information to compute matches for the entire path, partial matches are generated for each page. To obtain the overall query result, partial matches are then stitched together.

Kanne et al.'s work supports a variety of disk access policies. One option is to use a scheduler to access pages when they are needed, guaranteeing that each page is only accessed once. This ensures that no page is retrieved unnecessarily, but it makes no guarantees about access order. Alternatively, all pages can be accessed in a sequential scan, completely eliminating random I/O.

The process of combining partial matches somewhat resembles the way sub-query results derived from multiple vertical fragments are combined in this thesis. However, since

Kanne et al.'s technique does not have the benefit of a fragmentation schema (i.e., a succinct specification of how different portions of the schema are mapped to individual fragments) it cannot benefit from decomposing the query into multiple sub-queries and instead has to evaluate the full query over each page. This complicates the way partial matches are combined since it is not possible to use a join with two well-defined inputs. Instead, a self-join has to be used since any partial match might need to be combined with any other partial match.

Chan and Ni [36] use a similar fragmentation-aware approach to implement a publish/subscribe systems that models subscriptions using Boolean XPath queries. Queries are decomposed into sub-queries using a synopsis of how the data are fragmented. Special attention is paid to the order in which fragments are accessed and to eliminating partial results from consideration as early as possible.

Zhang et al. [141] present a technique that is based on dividing a query pattern into multiple next-of-kin sub-patterns, which consist only of parent/child and following-sibling/preceding-sibling relationships. Using a novel, page-based storage scheme, individual next-of-kin sub-patterns can be evaluated using a single, linear scan of the data. The resulting sub-pattern matches can then be assembled using structural join techniques to evaluate the axes connecting the corresponding sub-patterns. This approach combines some of the advantages of navigational query answering with those of index-based techniques.

Cong et al. [40] start with a distributed query evaluation strategy and describe how it can be used to evaluate queries over large collections in a centralized environment. Random disk I/O is avoided by partitioning the collection into fragments that fit into memory and then evaluating the query over one such fragment at a time. Finally, the partial results obtained from each of these fragments are combined to the overall query result.

The problem of centralized query processing on fragmented collections of XML data has also been studied within the context of streamed XML data on devices with limited resources [29]. In this work, the data are assumed to be fragmented based on a partitioning of the schema, resembling vertical fragmentation as defined in this thesis. Fragments are assumed to be streamed through a centralized query evaluation plan consisting of multiple operators, each of which is responsible for a single fragment. Document sub-trees are buffered at their corresponding operators until it can be decided whether they form part of the query result. To evaluate predicates across fragment boundaries, information is exchanged between operators using a table that encodes how sub-trees are related to each other.

Most of the fragmentation-aware query processing techniques that have been proposed for centralized XML query evaluation assume a single-threaded model with no parallelism. Bordawekar et al. [28] go beyond this and focus on how XPath queries can be evaluated efficiently in a shared memory system with multiple processor cores. To take advantage of the processing power of these cores, two approaches are used to partition the query processing work: With *data partitioning*, the node set resulting form a previous query step (which is assumed to be evaluated on a single core) is partitioned and the remainder of the query is then evaluated in parallel over each resulting partition. With *query partitioning*, in contrast, the query is rewritten into multiple sub-queries (for example by devising a set of mutually exclusive range predicates and inserting one of these predicates into each sub-query). Each sub-query is then evaluated independently at a separate core. Both partitioning strategies can be combined, resulting in a hybrid partitioning. By ensuring that all cores are utilized, the performance of CPU-bound single query workloads can be improved significantly.

3.2.3 Distributed Query Evaluation

In this section, existing work on distributed XML query evaluation is examined and compared to the technique presented in this thesis. Some of the techniques proposed in this area consist of extensions to XML query languages that allow for the explicit querying of remote documents. These techniques are described in Section 3.2.3.1. Other techniques aim to be transparent, i.e., they accept a fragmentation-unaware query and automatically determine how to distribute execution of this query over a fragmented collection. With this approach, it is usually necessary to decompose the query into sub-queries that can be evaluated over individual fragments. Techniques for performing this decomposition are discussed in Section 3.2.3.2. Next, Section 3.2.3.3 describes techniques that aim to eliminate irrelevant sub-queries and thereby avoid accessing some of the fragments of a collection. Section 3.2.3.4 describes index structure that can be used in distributed query evaluation. Section 3.2.3.5 then discusses techniques for the distributed execution of queries, with special focus on how sub-query results are assembled to the overall query result. There are several approaches for representing the partial results obtained from individual sub-queries. Section 3.2.3.6 discusses some of the alternatives proposed in the literature. Finally, Section 3.2.3.7 describes a set of implementation frameworks that have been proposed for building distributed XML query evaluation systems.

3.2.3.1 Distributed Query Language Extensions

A simple way to query distributed collections is to make the distribution explicit in the query language. Zhang and Boncz have developed the query language XRPC [144, 145], which is a superset of XQuery that has been enriched with facilities for shipping queries to remote sites. When XRPC queries are evaluated, these requests are forwarded and the results are used during local query processing. If a remote site does not support XRPC but supports plain XQuery, an adapter can be used to translate. This allows queries to make use of remote data sources without requiring any changes to those sources, which is desirable since a user might not have administrative control over them in a data integration scenario. While Zhang and Boncz do not describe any optimizations that go beyond what is explicitly specified in the query, XRPC may be well suited to serve as a target language for a distributed optimizer.

A similar approach is taken by Ré et al. [118]. They present the distributed query language XQueryD, which makes it possible to query multiple XML repositories within a single query. As with XRPC, a query shipping paradigm is followed. Additionally, the authors describe a suite of rewrites that can be used to improve distributed query performance.

Fernández et al. [47] present another language extension for XQuery named DXQ. Rather than focusing primarily on query evaluation, this technique is concerned with building distributed applications.

While these language-based approaches allow for the easy integration of multiple data sources, they require queries to be formulated to explicitly access individual fragments. In a scenario where fragmentation is used to improve performance and the schema is fixed, this is a significant disadvantage when compared to techniques that automatically infer how to distribute query processing.

3.2.3.2 Query Decomposition

Many existing distributed query evaluation techniques work by evaluating the entire query over each fragment (e.g., [33, 39]). If information about the way a collection is fragmented is available, it is possible to obtain better performance by decomposing the query into multiple sub-queries and only evaluating the relevant sub-queries over each fragment.

Le et al. [91] present a schema-based technique for decomposing a global query into local queries within the context of a data integration system. They identify which of the local schemas contain information that can be mapped to the global schema types used in the query. While their technique is not directly applicable to the distributed database scenario, one might employ a similar method to identify which fragments in a vertically fragmented collection are relevant for a given query.

Based on the XRPC extension of XQuery, Zhang et al. [146] describe a technique that transforms a centralized, data shipping-oriented query into a distributed, query shipping equivalent. This is achieved by decomposing the query and pushing part of the query execution to remote sites. This work supports all of XQuery, although certain query primitives make it impossible to perform effective query decomposition while maintaining result correctness. In these cases, the technique falls back to a data shipping approach.

Andrade et al. [12, 13] also mention that a query can be decomposed into sub-queries corresponding to individual fragments, however their papers do not describe how exactly this could be done.

3.2.3.3 Pruning Irrelevant Fragments

Pruning is an important step in distributed query optimization. The idea behind pruning is to identify which fragments are irrelevant for a given query and to refrain from accessing these fragments altogether. This can help improve the query throughput of a distributed system and can also reduce latency by eliminating the need to wait for processing of irrelevant fragments to finish.

Based on their partial evaluation strategy, Cong et al. [39] present a simple technique for pruning fragments. They identify fragments that can be pruned by examining the structural relationship between fragments. Unlike the pruning technique presented in this thesis, only structural constraints are taken into account. This results in a technique that can prune vertical fragments if it can be shown that they are not reached by the query.

Within the context of Active XML, Abiteboul et al. [2] present a technique that avoids calling certain remote functions and thereby limits the number of fragments that have to be retrieved in order to answer a given query. Due to the ad-hoc fragmentation of Active XML documents, it is not possible to identify in advance the set of irrelevant fragments. Instead, a lazy approach to retrieving fragments is employed, and fragments are only shipped to the central query processing site when the corresponding function call is reached during query evaluation. This is consistent with Active XML's focus on querying over integrated XML data services.

Within the context of structure-based fragmentation, Andrade et al. [12, 13] mention that it is possible to eliminate fragments from consideration in certain cases. While they provide a sketch of how this works in the case of horizontal fragmentation (by extracting predicates from the query workload and comparing them to the predicates used in the definition of the horizontal fragmentation), they do not address pruning of vertically partitioned data.

Hammerschmidt et al. [64] have developed a technique that uses automata to determine whether two XPath expressions intersect. This technique could be used to prune horizontal fragments by detecting whether a query and a fragmentation predicate are compatible. However, unlike the pruning technique presented in this thesis, only queries with a single extraction point are supported. Queries with multiple extraction points, as are frequently encountered in sub-queries within a hybrid fragmentation consisting of both vertical and horizontal fragmentation steps are, not supported, which limits the applicability of this technique in the context of hybrid fragmentation. Furthermore, the automaton-based technique is likely to deliver worse performance than the pruning technique presented in this thesis. This is because (potentially large) product automata have to be constructed, whereas the pruning technique given in this thesis aggressively prunes branches that are not shared between the query and the fragmentation predicate.

3.2.3.4 Index Structures

One option for enabling distributed query processing is the use of index structures, which can provide a compact summary of the data stored in other fragments and thereby enable some amount of local query processing over remote data.

Bremer and Gertz [31, 57] employ this approach to evaluate queries over a collection that is fragmented based on structure. One of their indexes stores label path information for all the nodes in the collection. The query evaluation technique presented in this thesis, on the other hand, only stores this information for proxy nodes and doesn't require this information to be part of a centralized index. By replicating the indexes across the system, the bulk of the query processing work can be performed efficiently and at a single site. Remote fragments only need to be accessed in order to evaluate value constraints in the query. While replicated indexes allow the authors to achieve good query performance, this comes at the potential cost of decreased scalability and more complicated update management (since replicated indexes have to be updated when changes are made to the collection). The centralized nature of index-based query processing might also lead to reduced intra-query parallelism and can potentially cause bottlenecks in the system when queries are not evenly distributed across all sites.

Koloniari and Pitoura [85] present a Bloom filter-based index structure that can be used to derive top-k results for an approximate structural query on a distributed XML collection. This index is used to prune fragments that will not yield top-k results. It can also serve to determine the order in which fragments are accessed, with the most promising fragments being accessed first.

Index structures are also widely used for the centralized querying of XML collections. For an overview of these techniques, refer to [89]. One technique of particular interest is Dewey IDs [44]. These IDs, which have been used in centralized XML query evaluation

[65], are employed in this thesis to express structural relationships between proxy nodes (i.e., the nodes that are inserted into the fragmented collection to represent edges that cross fragment boundaries).

3.2.3.5 Distributed Query Execution

An important consideration when evaluating queries in a distributed system is the trade-off between shipping data and shipping queries. On the one hand, it is possible to ship all relevant data to a central location where all query processing is performed. On the other hand, it is possible to ship the query or parts of the query to the sites storing the individual fragments and perform as much as possible of the query processing work distributed throughout the system, thereby taking advantage of parallelism and reducing communication cost; finally, only the (partial) results derived from each fragment are shipped back to the originating site. While not specific to XML, Franklin et al. [54] present an overview of the trade-offs between data shipping and query shipping.

While most of the literature on Active XML employs a data shipping approach [2, 3] there has been some work on distributing query processing [5]. Distributing query processing is complicated by the ad-hoc fragmentation of Active XML, which makes it difficult to determine which part of the query has to be executed over which fragments. Thus, query shipping is only applied in certain circumstances, while falling back to data shipping in other cases.

Based on a hybrid of ad-hoc and structure-based fragmentation, Cong et al. [39, 33] present a distributed query evaluation strategy that computes partial matches at each fragment and then combines them at a central location. The authors start with a technique that is designed to answer Boolean queries and then expand the scope of their work to include data-selecting queries with a single extraction point while maintaining impressive performance guarantees. This approach is developed further and implemented within a map-reduce framework [40]. Additionally, it is shown how this technique can be applied in the context of centralized query evaluation over large collections.

The main goal of Cong et al.'s strategy is to limit the number of times that each fragment has to be accessed and to provide a bound on the amount of network traffic incurred. The technique presented in this thesis, in contrast, considers the overall cost of evaluating a query, including the computation cost at each site. As the performance evaluation in Chapter 9 shows, optimizing for overall cost leads to lower overall query response time (cf. Section 9.1.2). Cong et al.'s partial evaluation approach requires that a specific technique be used for local sub-query evaluation at each fragment, limiting the potential for local query optimization.

Suciu presents a technique for evaluating queries over an ad-hoc distributed collection of semistructured data [124]. As in Cong et al.'s work, the main focus is on bounding the number of communication steps and the amount of data transferred, rather than on overall query performance, which explains why the technique presented in this thesis leads to better performance when considering overall query cost (as shown experimentally in Section 9.1.2).

3.2.3.6 Representing Partial Results

A common problem encountered when using a query shipping approach to distributed query evaluation is how to represent the partial results that need to be shipped from one site to another. If more than one of these results contain the same node, it may be advantageous not to send multiple copies of this redundant node.

Tajima and Fukui [127] present a technique that can be used to solve this problem by sending a minimal view that contains all results rather than sending each result separately. While their work is primarily intended for querying a single XML database instance over a network, it could also be used to ship partial results within a distributed system.

3.2.3.7 Distributed Query Evaluation Frameworks

Andrade et al. [12, 13] present the PartiX framework, which facilitates the distributed evaluation of XQuery over data sources that are fragmented horizontally, vertically, or in a hybrid fashion. They outline the different phases of localizing the distributed data, transforming global query plans to local plans for the suitable fragments, performing local optimization and reconstructing the final result.

Figueiredo et al. [50, 51] further develop the ideas presented in the PartiX paper and present a software architecture that implements the phases of query processing in distributed database systems. The authors present a clean architecture with well-defined hooks for optimization techniques although they describe no optimizations of their own.

3.2.3.8 Summary

While much of the existing work in the area of distributed XML query processing focuses on data integration, some of the existing work follows a performance motivation, as does the work presented in this thesis. These techniques include Cong et al.'s work on partial query evaluation [39, 33, 40], Bremer and Gertz's index-based technique [31, 57], and Suciu's work on querying semistructured data [124]. Table 3.3 shows an overview of these techniques and compares them to the techniques presented in this thesis.

As can be seen, both Bremer and Gertz's technique and the technique presented here follow a structure-based fragmentation approach. Suciu's technique on the other hand allows for ad-hoc fragmentation and Cong et al.'s work is based on a hybrid of both fragmentation approaches.

Comparing the features of the individual techniques, it can be seen that both Suciu's technique and the technique presented here offer some kind of query decomposition. However, in the case of Suciu's technique support is only partial: in general the same query is evaluated over each fragment, however this query can be optimized using local schema information resulting in a different plan for each fragment.

Both Cong et al.'s technique and the technique proposed here aim to prune irrelevant

Technique	Fragmentation	Feature		Needs index	Perf. focus
		Query decomp.	Pruning		
Cong et al.	hybrid	×	vert. only	no	communication
Bremer and Gertz	struc.-based	×	×	yes	communication
Suciu	ad hoc	partial	×	no	communication
This thesis	struc.-based	✓	✓	no	overall cost

Table 3.3: Comparison of distributed query evaluation techniques

fragments from distributed query execution. However, in the case of Cong's technique pruning is somewhat more limited and only applicable to a vertically fragmented scenario.

As can be seen, most techniques operate without the help of a replicated index structure and thereby avoid the potential complications for update management. The exception to this is Bremer and Gertz's technique, which relies on a fully replicated index structure.

When comparing the primary performance motivation, it can be seen that all three existing techniques primarily focus on communication cost. Cong et al. also evaluate the impact on query response time in their experimental sections, whereas the other two papers give no performance figures. This thesis, in contrast, focuses on the end-to-end cost of query evaluation.

3.3 Cost-Based Optimization

In addition to individual query evaluation techniques, a main contribution of this thesis is a cost-based optimization procedure that can determine the best distributed execution plan for a given query and distributed collection. This technique leverages centralized cost estimation techniques to estimate the cost of individual sub-queries. Based on these local cost estimates, the cost of various candidate plans is estimated and the candidate plan with the lowest estimated cost is chosen.

This section presents related work in the area of cost-based optimization. First, techniques for estimating the cost of centralized query evaluation techniques are presented (Section 3.3.1). Then, the focus is on distributed cost estimation techniques (Section 3.3.2), including techniques for XML and related techniques from a relational context. Finally, the approaches for enumerating plan alternatives are summarized (Section 3.3.3).

3.3.1 Centralized Cost Estimation

In the literature, several techniques have been developed to estimate the properties of centralized XML query evaluation. This section presents an overview of these techniques.

In addition to techniques that estimate the cost of evaluating a given query over a given collection, which are discussed in Section 3.3.1.1, some techniques focus on estimating cardinality (Section 3.3.1.2) or order properties (Section 3.3.1.3). Together, these techniques can be used to determine the local sub-query properties necessary for distributed query optimization.

3.3.1.1 Cost

Zhang et al. [140] point out that due to the more complex operators employed for the evaluation of queries over XML collections (e.g., holistic twig joins [32]), cost estimation is more complicated than in relational systems, where most operators have performance characteristics that can more easily be captured in a simple model. Thus, the authors propose a model for inferring cost properties through a statistical learning approach. Instead of aiming to model each aspect of the operators used in a query plan, features are extracted from query plans and these features are then used to predict the cost of these plans, leading to accurate cost estimates.

Hidaka et al. [68, 67] follow a relative approach and define formulas for estimating the cost of various XQuery language constructs based on the costs of the atoms used in these constructs. For example, the cost of a `for` expression that applies a function to each node in a set is estimated as the cost of obtaining the set plus the cost of applying the function to each element in the set (which depends on the size of the set). Using this approach, cost estimates for a wide range of XQuery features can be obtained.

Based on the tree algebra used in the database system Timber, Jagadish et al. [75] propose a cost estimation technique that focuses on estimating the cardinality of each intermediate result. Based on these cardinalities, the cost of each operator can be estimated and the overall cost of a query plan can be determined.

Systems such as MonetDB/XQuery [27] that process XML queries using a relational engine can obtain cost estimates using existing, relational cost estimation techniques. These can then be fed into an existing cost-based optimizer to determine the best relational plan for a query.

3.3.1.2 Cardinality

There is a substantial body of work on cardinality estimation for XML query processing. As pointed out by Jagadish et al. [75], cardinality estimation not only yields a useful property that can be used during distributed query optimization, by estimating the cardinality of intermediate results during query evaluation it is also possible to obtain cost estimates.

The Timber algebra [75] uses this approach. As described by Wu et al. [135, 136], cardinality estimates are obtained based on a specialized class of histograms that takes into account the position where elements matching certain predicates occur in the collection. Additionally, schema information is considered to detect cases in which the number of occurrences of a type of node can be inferred from cardinality constraints in the schema.

Aboulnaga et al. [7, 8] present a technique that estimates the selectivity of individual steps within a path expression based on a compact structural synopsis of the data. While this technique is primarily aimed at finding the best centralized query plan for evaluating a given query (making it useful for optimizing the sub-queries encountered by distributed query evaluation), it can also be used to obtain cardinality estimates for the overall query result.

Zhang et al. [143] follow a similar approach and use a synopsis of the collection to determine cardinality estimates. A key aspect of this technique is its adaptability to varying memory budgets, which is achieved by an incremental construction procedure. This not only makes it feasible to support changes to the underlying collection without having to fully recompute the synopsis, it also makes it possible to focus the synopsis on the portion of the collection that has the highest impact on query performance.

Freire et al. [55] present another technique for obtaining a histogram-based summary of a collection that can be used for cardinality estimation. The focus is on the information that can be extracted from the schema. Their algorithm for constructing the collection summary is designed to be run while documents are validated for compliance with a given schema, which allows the authors to reduce the performance overhead of histogram construction.

Chen et al. [38] focus on twig queries and propose a technique for determining the number of matches for a given twig query in a given document. This is done by storing cardinality information for small portions of twig queries, referred to as twiglets. While the space of possible twiglets is very large, this problem is addressed by the authors by storing twiglet properties efficiently in a compact tree structure.

Teubner et al. [131] present a technique that can be used to obtain cardinality estimates for wider range of XQuery expressions. This technique is based on query evaluation within a relational system and leverages both cardinality estimation techniques for XPath expression and relational cardinality estimation approaches.

Balmin et al. [16, 18] describe how cardinality estimation has been implemented within a production system consisting of a relational database combined with a native XML store. These cardinality estimation techniques are specialized for the operators used within the system's algebra to evaluate queries over XML collections.

3.3.1.3 Order Properties

Other important inputs to distributed query optimization are the order properties of local query plans. The idea to take order properties into account during query optimization was first proposed by Selinger et al. [121] in a relational context. In this paper, the concept of *interesting orders* was defined, which corresponds to orders that may enable the use of efficient evaluation strategies (e.g., operators that require their input tuples to be ordered in a particular way) or that can avoid the use of explicit sorting steps (e.g., if the result of a query is required to be ordered). The use of order properties in this thesis differs significantly from the way these properties are used in the relational context, most notably by not considering a hierarchy of order properties and instead dealing with sets of attributes such that a sequence of tuples is ordered independently by each attribute in the set.

In general, centralized XML query evaluation strategies ensure that their result is returned in document order, as is required by the XPath standard [24]. For queries with multiple extraction points, it is usually necessary to choose a single extraction point by which the result will be ordered (the ordering extraction point). Using the technique presented in Section 7.3.2.2, it may be possible to infer additional order properties, and all of these order properties can then be exploited to obtain an efficient distributed execution plan.

Within the context of the centralized evaluation of path expressions, Fernández et al. [69, 46] use a similar approach to avoid explicit sorting and duplicate elimination steps that would otherwise be required. Unlike the work presented in this thesis, which infers order properties based on Dewey IDs [44], Fernández et al. follow an automaton-based approach. Using this approach, the authors obtain a significant improvement in (centralized) query performance by avoiding unnecessary sort and duplicate elimination operators.

May et al. [104] describe how cost-based query optimization works within the centralized XML database system Natix. While the main focus of this paper is on finding the best join order, order properties are taken into account and aid in the selection of an appropriate physical join implementation.

3.3.2 Distributed Cost Estimation

Based on the properties of local sub-queries and their corresponding query plans, the distributed optimizer, as proposed in this thesis, aims to construct the most efficient distributed execution plan. To do this, multiple candidate plans are enumerated and the performance of each of these plans is estimated. This section describes related work in the area of cost estimation for distributed plans. While there is little work in the context of XML, there is a significant body of relational cost estimation techniques, many of which share characteristics with the technique presented in this thesis and thus warrant comparison.

3.3.2.1 Distributed Cost Estimation for Relational Collections

There is a sizable body of work in the area of distributed cost estimation for relational data and only the technique that are related to the approach taken in this thesis are presented here. For a more detailed discussion of this area of research, the reader is referred to Kossmann's survey [86] and Özsu and Valduriez's book [115].

Traditionally, distributed cost estimation has focused primarily on communication cost. In the work presented in this thesis, instead, the focus is on the end-to-end response time cost of query evaluation (taking into account parallelism).

In relational systems, a similar cost model was first considered by Ganguly et al. [56]. For a simple query model consisting of selection, projection, and join, the authors describe how response time estimates can be obtained in the presence of parallel execution, both in the shape of multiple independent inputs to a single operator and in the shape of pipelining (where parallelism is more limited).

Hong and Stonebraker [71] similarly use response time-based cost estimation within the context of a parallel database system in which the individual nodes share a common pool of memory. In addition to response time, this technique also takes resource usage into account and can be tuned to trade off between the two performance factors.

Ziane et al. [147, 148, 90] give response time-based cost estimation formulas for hash joins in a parallel database. However, unlike this work, the main focus of the authors is on the plan shapes considered, rather than on cost estimation itself.

In more recent work, Florescu and Kossmann [52] propose a multi-dimensional distributed cost model that takes both response time and resource usage into account. However, both of these dimensions are treated as constraints for which a minimum level has to be achieved but that do not represent the main goal of optimization. Instead, the focus is placed on the monetary cost of query evaluation. Response time and performance targets should be met as cheaply as possible, for example by minimizing the number of machines used to achieve a given level of performance.

3.3.2.2 Distributed Cost Estimation for XML Collections

Work on cost estimation for distributed XML processing is much more limited than the corresponding work in relational systems.

One of the few works in this area that explicitly mentions the use of a cost model is that of Gertz and Bremer [57, 31]. Unlike the cost model used in this thesis, Gertz and Bremer use a cost model that focuses on communication cost. In addition, they state that given the query evaluation method used in their work, full cost-based optimization is infeasible. Instead, their technique relies on a heuristic that aims to reduce the size of intermediate results.

3.3.3 Plan Enumeration

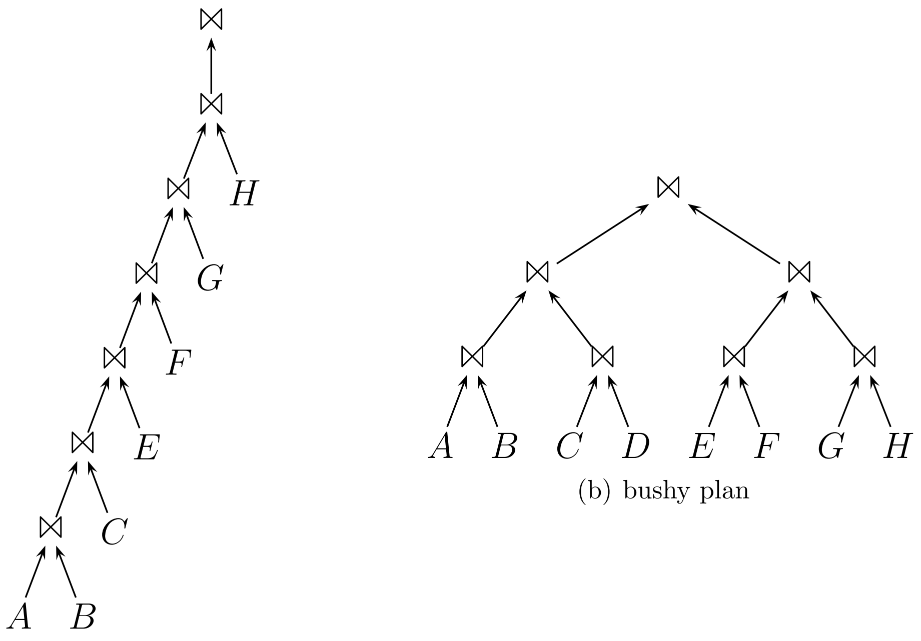
To determine the best distributed execution plan for a given query and distributed collection, plan alternatives are enumerated and compared based on their cost. There exists a significant body of work on the problem of efficiently enumerating plans, and while much of this work does not explicitly take distribution into account, many of these techniques can nevertheless be used to solve the problem of enumerating distributed plan alternatives.

In the following, a few notable plan enumeration techniques are discussed, particularly focusing on the applicability to distributed query optimization. For a more in-depth overview of plan enumeration, the reader is referred to Steinbrunn et al.'s survey of plan enumeration techniques in general [123], Kossmann and Stocker's overview of plan enumeration techniques that can be applied in a distributed system [87], and Özsu and Valduriez's book [115].

In general, plan enumeration techniques can be categorized into three groups. First are the techniques that are guaranteed to find the plan with the lowest estimated cost (either by exhaustively enumerating the entire search space or by using dynamic programming to enumerate plans consisting of optimal sub-plans). While these work well when the search space is relatively small, for more complex search spaces (in particular when bushy plans are considered), they might not be feasible. In this case, it is possible to use a randomized technique that samples the search space and is not guaranteed to find the global optimum. As the third alternative, it is possible to use heuristics rather than attempting to find the optimal plan. This has the advantage of significantly reducing the cost of plan enumeration. However, this advantage may come at the cost of reduced performance of the resulting plan.

3.3.3.1 Optimizing Techniques

In one of the earliest works in this area, Selinger et al. [121] propose a bottom-up plan enumeration strategy based on dynamic programming. Starting with individual relations, optimal sub-plans are constructed for increasingly large subsets of the relations referenced in the query. Sub-optimal sub-plans are discarded immediately and not considered further. The remaining sub-plans then serve as the building blocks of larger sub-plans until a



(a) left-deep plan

(b) bushy plan

Figure 3.8: Left-deep vs. bushy plans

complete plan has been obtained. Special attention is paid to order properties, which are exploited to find the most efficient implementation for each join. To limit the search space, only left-deep plans are considered. Thus, for each join, the inner operand consists of a single relation whereas the outer operand may consist of a plan containing further joins. See Figure 3.8(a) for an example of a left-deep plan. While in principle this technique could be applied to the distributed query optimization problem addressed in this work, the focus on left-deep plans may lead to decreased levels of parallelism in a distributed system. Thus, in general, it is preferable to use plan enumeration techniques that consider bushy plans (an example is shown in Figure 3.8(b)). However, in this case, the search space is significantly larger, and it may not be feasible to fully enumerate this space.

Ganguly et al. [56] extend the dynamic programming approach to bushy query plans as are encountered in a system with parallelism. A notable aspect of this work is its focus on a multi-dimensional cost model. To handle the large size of the search space encountered in this case, pruning techniques are proposed, which eliminate sub-optimal sub-plans from consideration in the later stages of optimization.

In a case study of DB2 Parallel Edition, Baru et al. [21, 22] describe how the optimizer of this system takes parallelism into account by considering bushy plans. The authors point out that considering these plans significantly increases the size of the search space. However, rather than resorting to heuristics to cope with this large space, they present a suite of techniques for pruning the search space. For example, they consider the placement of relations on the sites in the system and only consider plans in which there is some locality between operators and their inputs.

Haas et al. [63] present the distributed optimizer Garlic, which applies dynamic programming to the problem of plan enumeration in a data integration system. The optimizer assumes that the individual systems that are part of the integrated system may have varying query capabilities. This is then considered during plan enumeration.

Kossmann and Stocker [87] also extend the dynamic programming approach for use in a distributed database system and propose a novel plan enumeration technique called *iterative dynamic programming*. This technique combines dynamic programming with a greedy heuristic. Using this approach, optimal plans can be obtained when enough resources are available for plan enumeration. In cases where the search space gets too large, the technique automatically adapts and still produces plans that outperform those obtained using randomized strategies.

In more recent work, Moerkotte and Neumann [108, 109] present additional dynamic programming techniques for plan enumeration with join queries. In addition to a detailed study of the performance characteristics of several such approaches under different circumstances, this line of works expands the applicability of dynamic programming to queries that contain non-inner joins.

DeHaan and Tompa [42] follow a different approach. Rather than performing plan enumeration in a bottom-up fashion, as is usually done by dynamic programming approaches, their work follows a top-down approach that uses memoization. In this scenario, branch-and-bound pruning can be applied to significantly reduce the size of the search space, whereas this optimization is unavailable with bottom-up approaches. Additionally, DeHaan and Tompa's technique is flexible with regard to the available memory; in cases where memory is scarce additional computation can be substituted by reducing memoization. Fender and Moerkotte [45] expand on this work on top-down enumeration and present a technique that is easier to implement (by avoiding the use of specialized data structures) as well as being more efficient for certain classes of queries.

3.3.3.2 Randomized Techniques

An early approach to solve the problem of optimizing bushy query plans is proposed by Ioannidis and Wong [73] for a class of recursive queries. They present a randomized strategy based on simulated annealing to optimize a bushy query plan. This method works by choosing a naïve initial plan and then attempting to improve it. A key feature of simulated annealing is the fact that rather than greedily focusing on improving the performance of the plan at each step, with a certain probability (which decreases over time), slight decreases in plan performance are accepted. Using this strategy, the technique avoids getting stuck in a local optimum.

Swami and Gupta [126] follow a similar approach based on iteratively improving an initial query plan. Unlike Ioannidis and Wong's work, here, the focus is on queries with many joins, which more closely corresponds to the scenario encountered during distributed query optimization as described in this thesis. In later work [125], this technique is then combined with heuristics, and it is shown that good results can be obtained using a combination of these two approaches.

Ioannidis and Kang [72] present a further randomized plan enumeration technique, referred to as *two phase optimization* (commonly abbreviated as 2PO). This technique combines the iterative improvement strategy proposed by Swami and Gupta with a simulated annealing strategy and obtains better results than each technique alone.

Grošelj and Malluhi [61] point out that in the context of distributed query processing exhaustive enumeration techniques are generally not feasible. To address this problem, the authors propose a technique that is based on sampling the search space and then comparing the costs of the plans contained in this sample. To obtain a good sample of the plan options, a randomized plan generation procedure is used that yields plans that are uniformly distributed throughout the search space.

3.3.3.3 Heuristic Techniques

Lu et al. [92] present a greedy plan enumeration technique that considers bushy query plans and takes parallelism into account. In addition to considering the parallelism between multiple join operators (which is applicable to distributed query optimization as considered in this thesis), intra-operator parallelism is considered, where a single operator is simultaneously assigned to multiple processors.

Chen et al. [37] develop this idea further and propose additional heuristics for plan enumeration in the presence of parallelism. A particular focus of this work is on allocating operators to the various processors in a parallel system.

Transformation-based optimizers (such as the Exodus optimizer [60]) also fall into the category of heuristic optimizers. With these optimizers, a set of legal transformation rules for query plans are defined and a heuristic measure of the expected performance benefit of each rule is used when choosing which rules to apply and in which order to apply them.

3.3.3.4 Summary

As can be seen, there is a large variety of techniques that have been designed to enumerate plan alternatives. Traditionally, optimizing techniques have been considered unsuitable for application in a distributed system. This is because, in this scenario, search spaces tend to be large: not only do join order and physical join implementations (of which there are more alternatives in a distributed system) need to be determined, it is also necessary to determine where (i.e., at which site) each operator is to be evaluated.

In the distributed optimization problem considered in this thesis, the size of the search space is comparatively limited. This is because distributed query plans merely combine the results of local query plans (each of which is optimized independently at the site holding the corresponding fragment). Thus, distributed query plans consist of relatively few, large atoms that are combined by a limited number of operators. Pruning irrelevant sub-queries further reduces the size of the search space, making optimizing plan enumeration techniques a feasible alternative in many cases.

Chapter 4

Fragmenting XML Collections

The focus of this thesis is on improving the scalability of XML query execution by parallelizing the process across the sites of a distributed system. As a first step, the collection is decomposed into multiple fragments, each of which can then be placed at a different site in the distributed system. This chapter describes a suite of techniques for decomposing XML data.

The fragmentation model described in this chapter partitions an XML collection based on characteristics of the content and the structure of the data. Two methods of fragmentation are supported, whose semantics are inspired by relational distribution techniques but whose mechanisms are notably different (see Section 3.1 for a discussion of this).

The two methods are

- *horizontal fragmentation*, which is based on predicates and results in a collection that is partitioned into fragments that all follow the same schema, and
- *vertical fragmentation*, which is based on partitioning the schema, with each fragment covering a different portion of the schema.

While each of these mechanisms can be used on their own, it is also possible to combine multiple fragmentation steps of either type to form a hybrid fragmentation. Together, these techniques make it possible to fragment a collection in a wide variety of ways. As the later chapters will show, this is important for tailoring a fragmentation such that it maximizes the performance of a given query workload. At the same time, the fragmentation model remains simple, which makes it tractable for distributed query evaluation techniques to take advantage of the characteristics of a particular fragmentation.

One of the key design goals for the fragmentation model presented here is the ability to obtain a succinct specification of any fragmentation within this model. In the case of horizontal fragmentation, this specification comes in the shape of a set of *fragmentation predicates*. For vertical fragmentation, the specification is represented by a *fragmentation schema*, in which each node type in the schema is assigned to exactly one fragment. In either case, as will be shown in the later chapters of this thesis, the fragmentation specification is an invaluable asset for optimizing distributed query evaluation.

Fragmentation, as defined here, focuses on partitioning a collection into non-overlapping fragments. Other approaches, which replicate all or part of the collection, can be used in conjunction with the techniques presented in this thesis for further performance improvement. However, these approaches are outside the scope of this thesis, as are the ad-hoc fragmentation approaches described in Section 3.1.2.1.

The remainder of this chapter gives a formal definition of horizontal and vertical fragmentation and describes how a fragmentation of either type can be specified. The chapter also discusses how fragmentation steps of both types can be combined to form a hybrid fragmentation.

4.1 Horizontal Fragmentation

Horizontal fragmentation as modeled here assumes a collection that consists of multiple document trees. These document trees can either be entire XML documents or they can be the result of a previous vertical fragmentation step. In either case, all document trees are required to correspond to the same schema. Multiple-document collections where all documents follow the same schema are a common use case for XML. Popular examples include collections of MathML [34] and CML [112] documents.

A horizontal fragmentation of an XML collection is defined by a set of fragmentation predicates. In distributed relational systems, fragmentation predicates are commonly expressed as algebraic expressions. In the case of XML data, tree patterns represent a convenient abstraction for expressing fragmentation predicates. Therefore, in this work, the fragmentation predicates that specify a horizontal fragmentation are expressed as tree patterns without an extraction point, which are referred to in the following as *fragmentation tree patterns* (FTPs).

Definition 4.1. A tree pattern $fp = \langle N, E, r, \nu, \epsilon, T, c\rangle$ is a *fragmentation tree pattern* if $T = \emptyset$. A document tree d *matches* the fragmentation tree pattern fp if evaluating fp over d yields at least one result tuple. ■

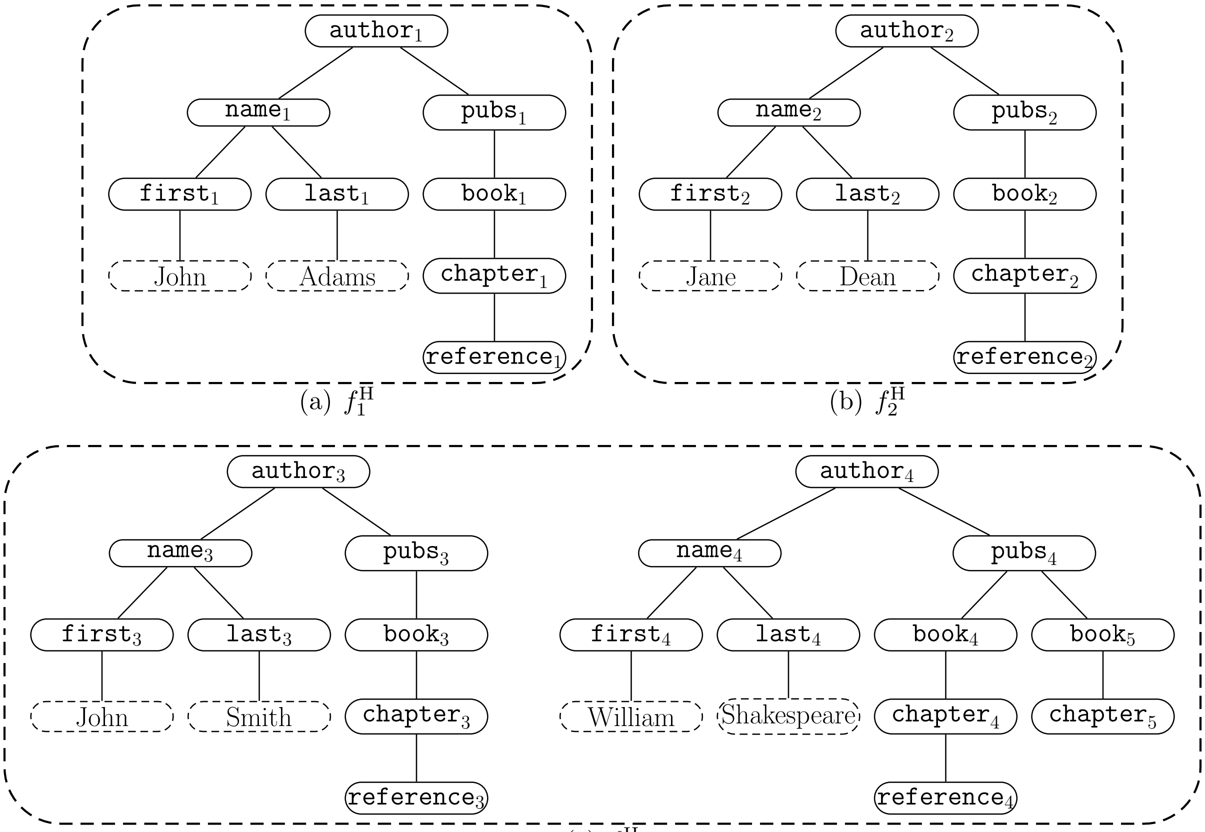
For notational convenience, $fp(d)$ denotes that document d matches FTP fp.

Definition 4.2. Let $D = \{d_1, d_2, \ldots, d_n\}$ be a collection of document trees such that each $d_i \in D$ corresponds to the same schema. Further, let $FP = \{fp_1, fp_2, \ldots fp_m\}$ be a set of FTPs. Then $F = \{\{d_i \in D \mid fp_j(d_i)\} \mid fp_j \in FP\}$ is the set of *horizontal fragments* of D corresponding to the FTPs in FP. ■

Each fragment consists of the document trees that match the FTP corresponding to that fragment. To ensure that the fragmentation is lossless and that the fragments are disjoint, for each document that conforms to the schema of the collection, there must be exactly one matching FTP.

Definition 4.3. Let $F = \{f_1, f_2, \ldots f_m\}$ be a set of horizontal fragments of the documents D corresponding to the FTPs in $FP = \{fp_1, fp_2, \ldots fp_m\}$. Then F is a *horizontal fragmentation* of D if $\forall d_i \in D : \exists$ unique $fp_j \in FP$ where $fp_j(d_i)$ (i.e., if FP induces a partitioning of D). ■

The losslessness of a horizontal fragmentation can be enforced by carefully crafting the value constraints in the FTPs so that they cover the entire domain of the values to which they refer. Since this is generally difficult to verify, in this thesis, the algorithms that are used to produce a horizontal fragmentation are required to ensure that the losslessness requirement holds.



(a) f_1^{H}

(b) f_2^{H}

(c) f_3^{H}

Figure 4.1: A horizontally fragmented collection

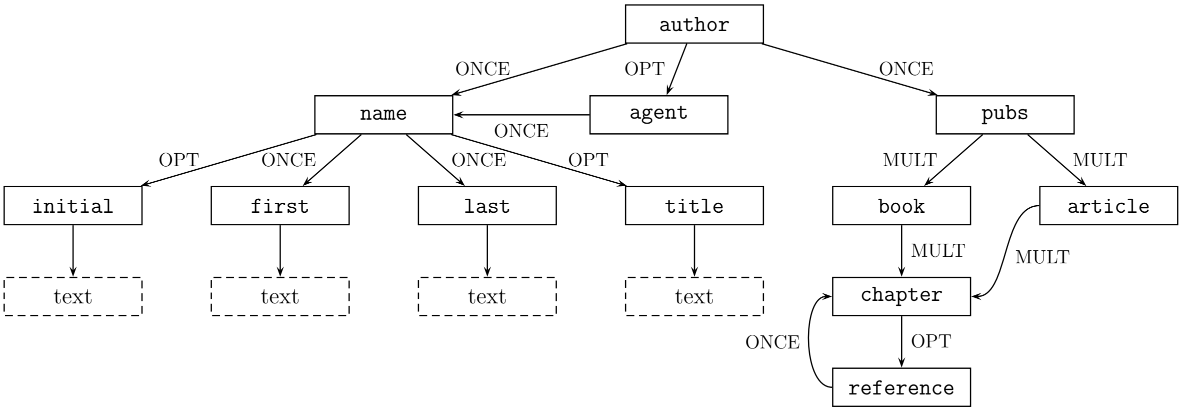


Figure 4.2: An XML schema graph

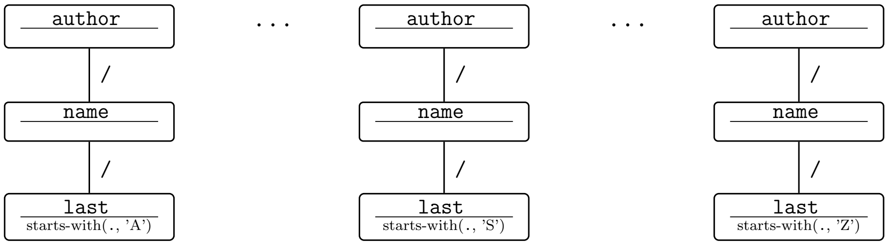


Figure 4.3: Set of fragmentation tree patterns (FTPs)

Assuming that the document trees in the fragmented collection shown in Figure 4.1 conform to the schema in Figure 4.2 and that $m(\texttt{last})$ (i.e., the domain of the text content of `last` nodes in the collection) is the set of strings that start with upper-case letters of the English alphabet, then the fragmentation of this collection can be described by the set of FTPs shown in Figure 4.3.

In general, horizontal fragments consist of multiple document trees. In the following, the set of document trees in a horizontal fragment f_i^{H} is denoted as $\mathrm{subtset}(f_i^{\mathrm{H}})$ and the number of such document trees is denoted as $\mathrm{nsubt}(f_i^{\mathrm{H}}) = |\mathrm{subtset}(f_i^{\mathrm{H}})|$.

4.2 Vertical Fragmentation

Unlike horizontal fragmentation, which defines fragments based on the content of the collection, vertical fragmentation defines them based on its structure. As will be shown later, this distinction has a large impact on how efficiently certain types of queries can be answered. In addition, vertical fragmentation enables a set of optimization techniques that complement the techniques that can be applied with horizontal fragmentation.

A vertical fragmentation is defined by partitioning the schema graph into connected subgraphs. This yields a vertical fragmentation schema.

Definition 4.4. Let $\langle \Sigma, \Psi, s, m, \rho \rangle$ be a schema graph. A *vertical fragmentation schema* is defined by a partitioning $F_\Sigma = \{f_0, f_1, \ldots\}$ of the set of node types Σ such that for each node type $\sigma \in \Sigma$ there exists a unique fragment $f \in F_\Sigma$, such that $\sigma \in f$. For each $f \in F_\Sigma$

$\langle f, (\Psi \cap (f \times f))\rangle$ is required to be weakly connected. That is, for any two node types σ_1 and $\sigma_2 \in f$, there must be a path from σ_1 to σ_2 or from σ_2 to σ_1 and this path must only traverse node types in f. ■

As can be seen in the definition, each type $\sigma \in \Sigma$ is assigned to exactly one fragment $f \in F_\Sigma$. For notational convenience, the fragment corresponding to σ will be referred to as $f_\Sigma(\sigma)$ in the following.

This model of vertical fragmentation can handle collections that consist of a single or of multiple document trees. As in the horizontal case, it is possible that these document trees are the result of a previous fragmentation step, which makes it possible to combine horizontal and vertical fragmentation.

For an example of a vertical fragmentation schema consider Figure 4.4. The dashed outlines show how the node types in this schema have been fragmented into six disjoint,

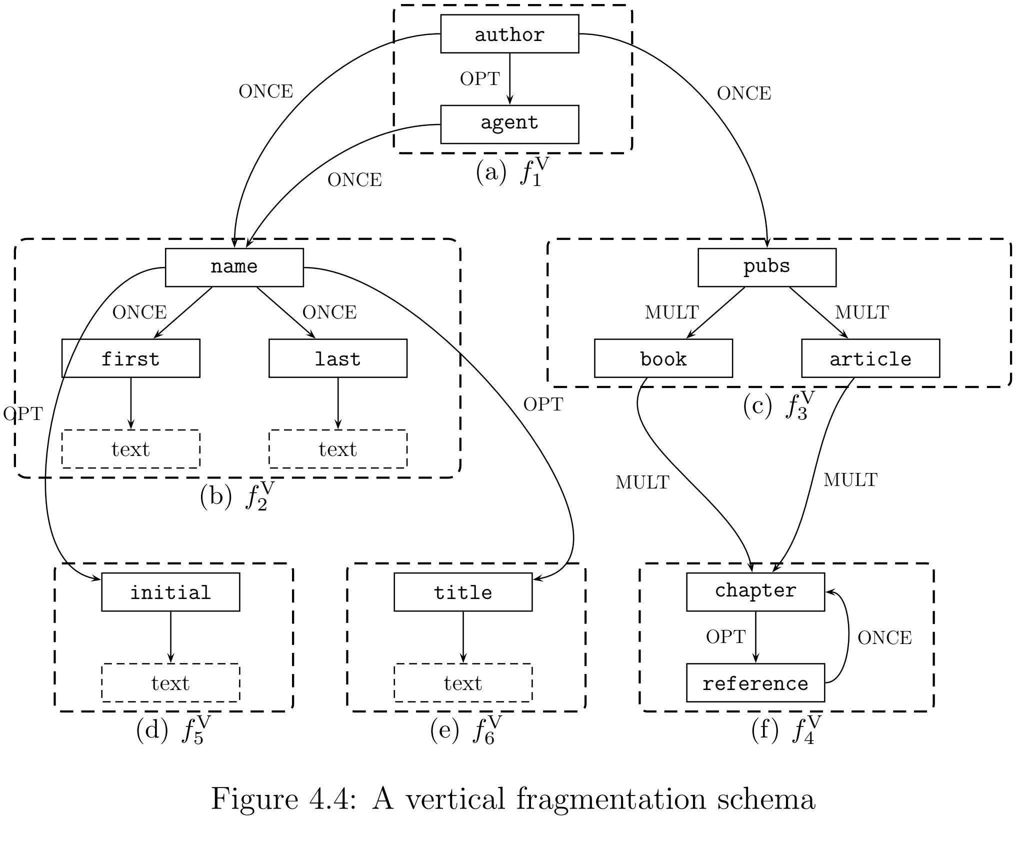


Figure 4.4: A vertical fragmentation schema

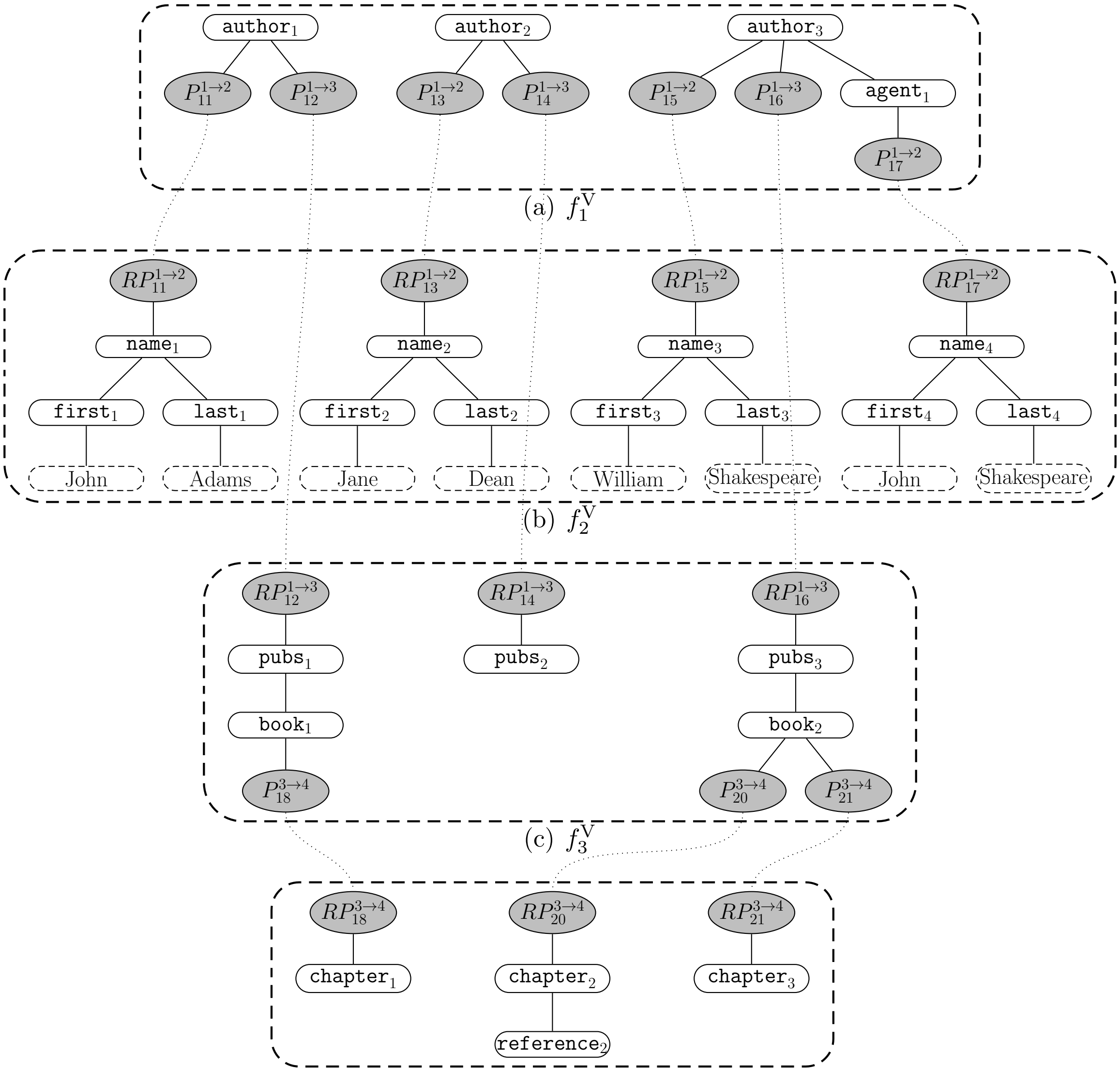
connected subgraphs. Fragment f_1^{V} consists of the node types `author` and `agent`; fragment f_2^{V} consists of the node types `name`, `first` and `last` along with their text content; fragment f_3^{V} consists of `pubs`, `book` and `article`; fragment f_4^{V} includes the node types `chapter` and `reference`; fragments f_5^{V} and f_6^{V} contain the node types `initial` and `title`, respectively.

Since the schema graph is required to be connected, after fragmentation, there will be graph edges that cross fragment boundaries. Whenever the schema contains an edge from a fragment f_i^{V} to another fragment f_j^{V}, f_j^{V} is referred to as a *child fragment* of f_i^{V} and f_i^{V} is referred to as a *parent fragment* of f_j^{V}. There is exactly one fragment $f_\rho^{\mathrm{V}} \in F_\Sigma$ that contains the root node type ρ. This fragment is referred to as as the *root fragment*. While the schema graph may contain cycles, for performance reasons, the fragmentation schema is required to be a DAG (i.e., cycles must be contained within a single fragment).

When fragmenting a collection according to a vertical fragmentation schema, there are generally some document edges that cross fragment boundaries. Since the individual fragments are to be stored at different sites in a distributed system, these edges cannot simply be left in place, as it would be unclear which fragment they belong to and how they can be represented. To solve this, the concept of a pair of *proxy nodes* is introduced. These special nodes are inserted into the collection on either side of a cross-fragment edge.

More precisely, a document edge from fragment f_i^{V} (the originating fragment) to fragment f_j^{V} (the target fragment) is represented by inserting a pair of proxy nodes $P_b^{i \to j}$ and $RP_b^{i \to j}$ into fragments f_i^{V} and f_j^{V}, respectively. $P_b^{i \to j}$ in the originating fragment f_i^{V} is referred to as a *proxy node* and $RP_b^{i \to j}$ in the target fragment f_j^{V} is referred to as a *root proxy node*. The latter is called a root proxy node because it forms the root of a sub-tree in the target fragment. Since $P_b^{i \to j}$ and $RP_b^{i \to j}$ share the same ID (denoted by the subscript b) and reference the same originating and target fragment ($i \to j$), they correspond to each other and together represent a cross-fragment edge in the collection.

For an example of a vertically fragmented collection, consider Figure 4.5. This collection has been fragmented according to the vertical fragmentation schema shown in Figure 4.4. The fragments f_5^{V} and f_6^{V} are empty and therefore not shown. The proxy pair consisting of $P_{11}^{1 \to 2}$ in fragment f_1^{V} and $RP_{11}^{1 \to 2}$ in fragment f_2^{V}, for example, represents an edge from an `author` node in f_1^{V} to a `name` node in f_2^{V}. For greater clarity, corresponding proxy and



(a) f_1^{V}

(b) f_2^{V}

(c) f_3^{V}

(d) f_4^{V}

Figure 4.5: A vertically fragmented collection

root proxy nodes are connected by a dotted line.

As can be seen in Figure 4.5, vertical fragments generally consist of multiple unconnected pieces of XML data, referred to as *document sub-trees*. Fragment f_1^{V}, for example, contains three sub-trees, each of which consists of the `author` and `agent` nodes of one of the documents in the collection. In the following, the set of sub-trees in a vertical fragment f_i^{V} is denoted as $\mathrm{subtset}(f_i^{\mathrm{V}})$ and the number of such sub-trees is denoted as $\mathrm{nsubt}(f_i^{\mathrm{V}}) = |\mathrm{subtset}(f_i^{\mathrm{V}})|$.

4.3 Hybrid Fragmentation

Horizontal and vertical fragmentation (as defined above) are completely orthogonal. This makes it possible to compose multiple fragmentation steps of both types, resulting in a hybrid fragmentation.

For example, consider the collection shown in Figure 4.7. This collection has first been fragmented vertically according to the fragmentation schema shown in Figure 4.4. After that, fragment f_2^{V} has been further fragmented horizontally according to fragmentation predicates shown in Figure 4.6, yielding the fragments f_{2a}^{VH} and f_{2b}^{VH}.

The ability to concatenate multiple fragmentation steps significantly increases the flexibility of the fragmentation model. As will be shown later, this can help to significantly improve the performance of distributed query evaluation by increasing the number of options for finding a fragmentation that is suitable for a given query workload. In addition,

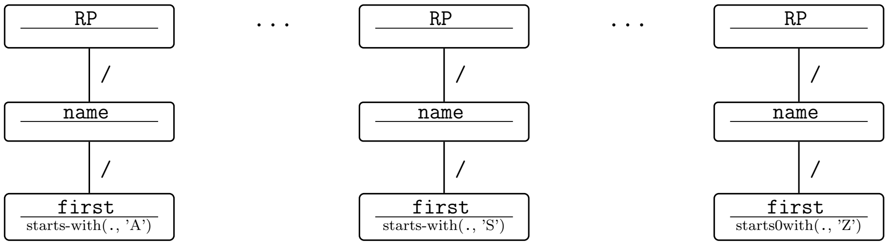
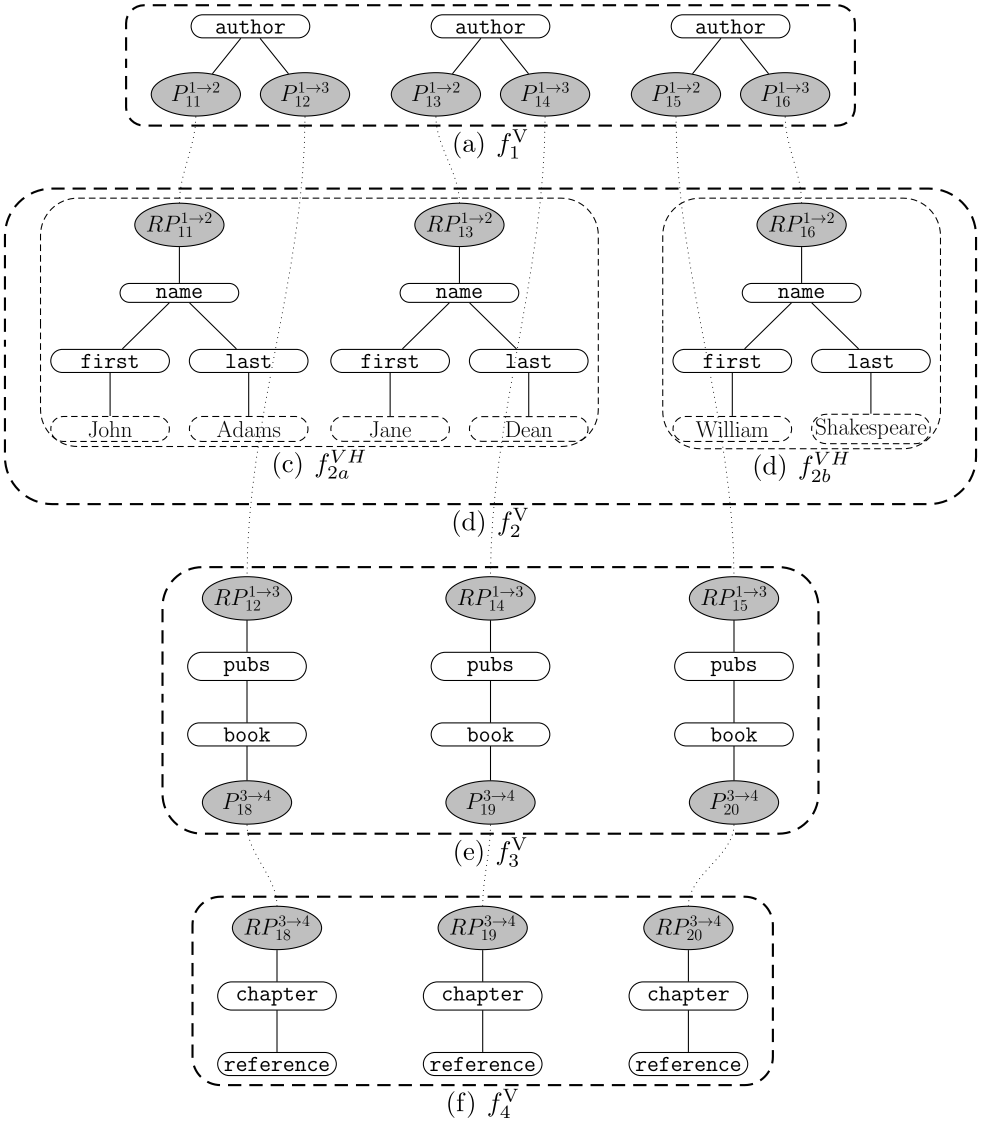


Figure 4.6: FTPs used in hybrid fragmentation

hybrid fragmentation also makes it possible to apply horizontal fragmentation (which by itself requires a multiple-document collection) to a single-document collection by first fragmenting vertically.

4.4 Summary

This chapter has described a model for fragmenting XML collections. Based on characteristics of the data and the schema, a collection is partitioned into multiple fragments and a fragmentation specification is obtained. While the fragmentation model described here provides an inventory of possible fragmentation steps, it does not specify how a collection should be fragmented to optimize performance for a given query workload. To define such a workload-aware fragmentation strategy, the characteristics of the distributed query evaluation techniques need to be taken into account. Thus, the presentation of a workload-aware fragmentation algorithm is deferred to Chapter 8, after the query evaluation techniques proposed in this thesis have been described.



(a) f_1^V

(c) f_{2a}^{VH}

(d) f_{2b}^{VH}

(d) f_2^V

(e) f_3^V

(f) f_4^V

Figure 4.7: A hybrid fragmented collection

Chapter 5

Distributed Query Evaluation Over Fragmented Collections

A simple approach to evaluating a query over a fragmented and distributed collection is based on shipping the data relevant to the query to a central location and then applying a centralized query evaluation strategy. While this *data shipping* approach works, it has the drawback of having to transfer potentially large volumes of data over the network. More importantly, due to the fact that queries are evaluated centrally, parallelism is severely limited, which makes this strategy unsuitable for improving the scalability of query evaluation. Therefore, the techniques in this thesis are based on the *query shipping* paradigm, in which the query (or parts of the query) are shipped to the sites holding the individual fragments and then evaluated locally at these sites. The results from each site are then shipped back to the query dispatcher and combined to the overall query result. The query shipping approach has the advantage that the query can be evaluated in parallel over each fragment, thus distributing the processing cost across the sites in the system, leading to increased scalability. For further discussion of the trade-off between data shipping and query shipping, see Section 3.2.3.5.

Distributed query evaluation based on query shipping consists of three main steps:

Query localization Localization is the process of determining which fragments are relevant to a given query and decomposing the query into sub-queries that can be evaluated over individual fragments.

Local query execution Using existing, centralized query evaluation techniques, such as those presented in Section 3.2.2, the sub-queries resulting from localization can then be evaluated at the sites holding the individual fragments. Each site is free to choose the most appropriate centralized execution technique for a given fragment and sub-query.

Distributed execution plans Localization alone is not sufficient to answer a query over a fragmented and distributed collection. While evaluating the sub-queries resulting from localization yields partial results corresponding to individual fragments, these partial results still need to be assembled to the overall query result. How this is done is specified in a distributed execution plan (DEP).

The focus of this chapter is mainly on the distribution aspects. First, a localization technique is proposed for horizontally fragmented collections, and, based on this, a technique for defining distributed execution plans is described. Then, localization and execution plan generation in the vertically fragmented scenario are discussed. For both cases, this chapter presents initial, unoptimized techniques, which will form the foundation for the performance improvements presented in Chapter 6.

5.1 Horizontal Fragmentation

This section discusses how a query can be evaluated over a horizontally fragmented and distributed collection. First, a simple data-shipping strategy is introduced to give context and to motivate the need for a query-shipping approach. Then, the query-shipping approach is explained and distributed execution plans for horizontal fragmentation are introduced.

5.1.1 Data Shipping

A horizontal fragmentation is defined as a partitioning of the set of document trees in the collection (cf. Definition 4.3 on page 65). Based on this definition, query q can be evaluated over collection D, which has been horizontally fragmented into a set of fragments F, by evaluating a centralized query plan for query q (denoted as p) over the union of all fragments in F (after gathering them at a central location):

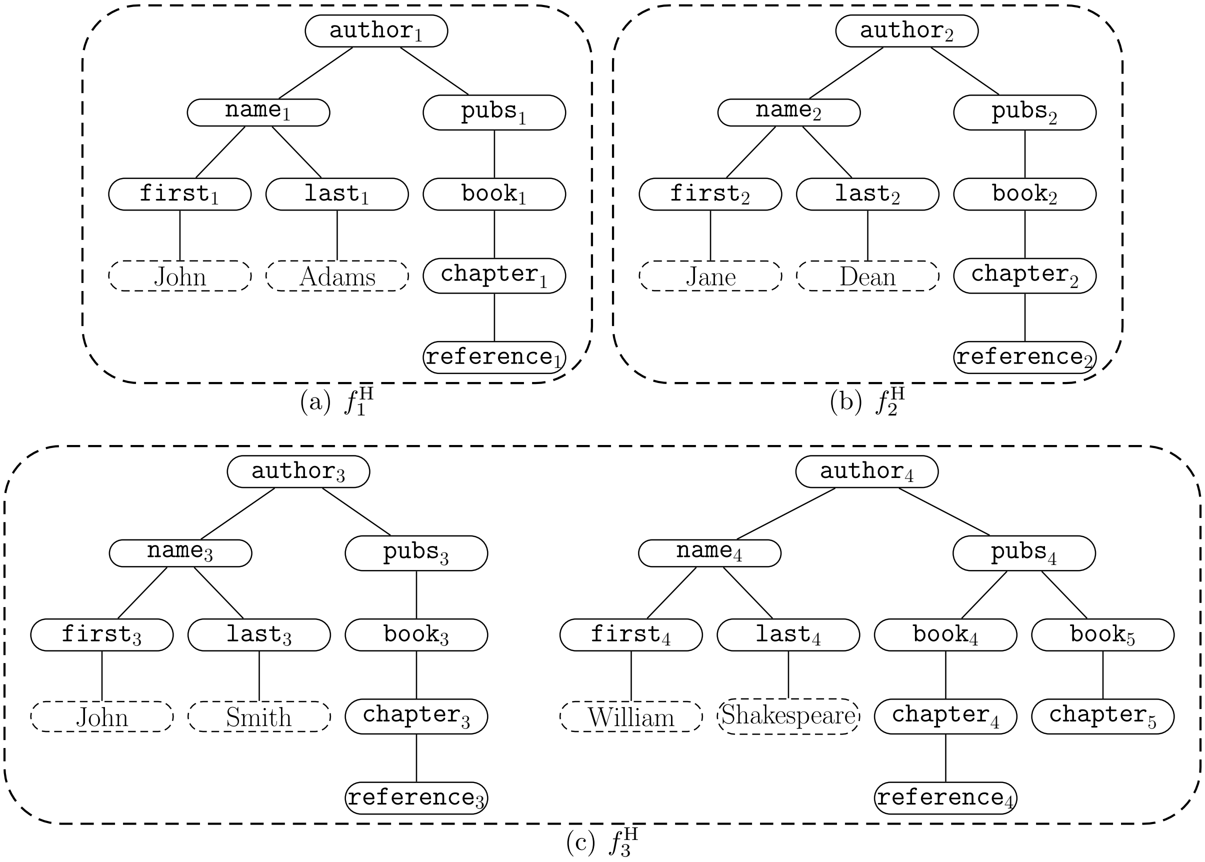
$$p(D) = p\left(\bigcup_{f \in F} f\right)$$

It is easy to see that this always leads to the correct result, since $\bigcup\limits_{f \in F} f$ is identical to D. However, as mentioned before, this data shipping technique is inefficient because the entire collection has to be transmitted for each query. While caching might alleviate this problem to some degree, it introduces additional overhead when processing updates. Furthermore, with a data shipping approach, query processing is performed at a centralized location, which limits parallelism and makes this technique unsuitable for the scalability goals of this thesis.

5.1.2 Distributed Execution Plans

Tree patterns (as defined in Definition 2.3 on page 19) can express structural constraints only between nodes in the same document. Therefore, a match for a tree pattern is always derived from exactly one document tree in the collection. This insight can be exploited to define a distributed execution plan that parallelizes query execution by pushing pattern matching to the sites holding the individual fragments.

As shown in Figure 5.1, horizontal fragmentation never splits a single document tree across multiple fragments. Instead, each fragment consists of a subset of the set of document trees that make up the collection. In addition, horizontal fragmentation guarantees that all fragments correspond to the same schema as the overall collection. Therefore, it is possible to evaluate a query over a horizontally fragmented collection by evaluating



(a) f_1^{H}

(b) f_2^{H}

(c) f_3^{H}

Figure 5.1: A horizontally fragmented collection

the same query over each fragment. This yields a sequence of pattern matches for each fragment, which can then be merged to obtain the overall query result.

Definition 5.1. If p is a plan that evaluates query q over an un-fragmented collection of document trees D and F is a horizontal fragmentation of D, then

$$p_f(F) := \mathbb{S}_{a_1^e}\left(\bigodot_{f \in F} p(f)\right)$$

is a *distributed execution plan*(DEP) that evaluates the same query on F, where $\odot$ denotes merging the result sequences, and $p_f(F) = p(D)$. ■

As shown in the definition, it may be necessary to sort ($\mathbb{S}$) the results received from the individual fragments in order to return them in a stable global order as required by the XQuery data model [48]. Section 6.1.2 further discusses this and presents a set of techniques that make it possible to avoid the overhead associated with sorting.

It is easy to observe that the distributed query execution technique presented in this section accesses all fragments of a horizontally fragmented collection. Depending on the query, this may not always be necessary. Based on this insight, Section 6.1.1 introduces a technique for pruning certain horizontal fragments from a DEP if it can be shown that these fragments do not contribute to the query result.

5.2 Vertical Fragmentation

This section describes an initial strategy for the distributed evaluation of a query over a vertically fragmented XML collection. This strategy works by decomposing the query into sub-queries corresponding to individual vertical fragments. QTPs provide a convenient abstraction for performing this decomposition. Decomposing the QTP yields a set of local QTPs corresponding to individual fragments. These are then converted to local query plans and each local query plan (abbreviated as LQP) is evaluated at the site holding the corresponding fragment. Finally, the results of all LQPs are combined to the overall query result.

Before decomposing a QTP, it is necessary to determine how the pattern nodes in the QTP relate to the vertical fragments of the collection. Section 5.2.1 describes how this is done by traversing the QTP and annotating each node with its corresponding fragment. After the QTP is fully annotated, it is decomposed into multiple local QTPs, each corresponding to a single fragment. Next, each local QTP is converted to an LQP and evaluated at the site holding the corresponding fragment. Each site is free to choose the most suitable local query evaluation strategy. Finally, the results derived from each fragment are combined using joins. How this is done is specified in a distributed execution plan. The remainder of this section explains each of these steps in detail.

Algorithm 1: annotate(x, f_{parent}) *annotates nodes in a QTP with fragments*

```
   input   : QTP node $x$, vertical fragment of parent node $f_{\text{parent}}$
 1 if $x \in N$ then
 2     // $x$ is a pattern node
 3     if $\nu(x) \in \Sigma$ then
 4         // $x$ has explicit node test
 5         $a(x) \leftarrow f_{\Sigma}(\nu(x))$
 6         if $\exists$ child $y$ of $x$ then
 7             annotate($y, a(x)$)
 8     else
 9         // $x$ has wildcard node test
10         $X \leftarrow$ split-wildcard($x$)
11         for $\langle x_{\text{split}}, f_{\text{split}} \rangle \in X$ do
12             $a(x_{\text{split}}) = f_{\text{split}}$
13             if $\exists$ child $y$ of $x_{\text{split}}$ then
14                 annotate($y, a(x_{\text{split}})$)
15 else
16     // $x$ is a logic node
17     $a(x) \leftarrow f_{\text{parent}}$
18     for child $y$ of $x$ do
19         annotate($y, a(x)$)
```

5.2.1 Annotating QTPs

Conceptually, a QTP is decomposed by splitting it into multiple portions, each of which consists of pattern nodes corresponding to a single vertical fragment. Therefore, before decomposing a QTP, each node in the QTP is annotated with the fragment to which it belongs.

Algorithm 1 describes how this is done. The function annotate() is first called with the root node r and the root fragment f_{ρ}^{V} passed as parameters. It then recursively traverses the entire pattern in a depth-first manner (lines 7, 14, and 19). For each node x encountered, $a(x)$ is set to the fragment to which this node corresponds. For a pattern node with an explicit node test for a type $\sigma \in \Sigma$, this is straightforward: the pattern node is assigned to the fragment that contains nodes of type σ (line 5). Logic nodes are assigned to the fragment corresponding to their nearest pattern node ancestor (line 17), this information is passed as parameter f_{parent} with each recursive call. To illustrate this, Figure 5.2 shows

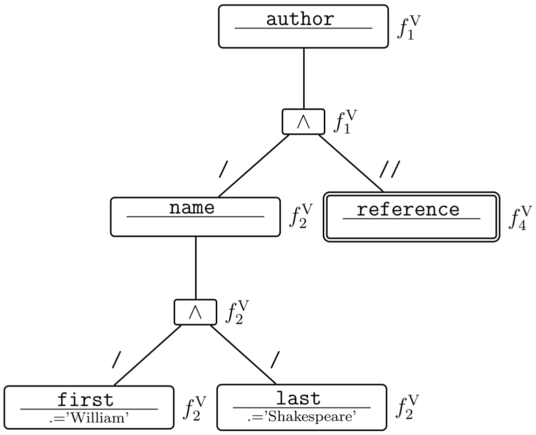


Figure 5.2: Annotated QTP representation of query q_1

an annotated version of query q_1.

For pattern nodes with wildcard node tests, the situation is more complicated since the algorithm needs to take into account the possibility that these pattern nodes may match collection nodes from more than one vertical fragment. Consider, for example, the QTP representation of query q_4 shown in Figure 5.3. In this QTP, all nodes have been annotated, except for the pattern node with a wildcard node test ($*$). Inspecting the schema shows that the child step leading to this pattern node could be satisfied by nodes of the types `name`, `agent`, or `pubs` (highlighted in Figure 5.4), corresponding to fragments f_2^{V}, f_1^{V}, and f_3^{V}, respectively.

To resolve this ambiguity, pattern nodes with wildcard node tests (and the branches below these pattern nodes) are duplicated for each fragment that they may match as shown in Algorithm 2. Assuming that x is a pattern node with a wildcard node test and parent(x) is assigned to fragment f_i, then the candidate fragments for x can be determined as follows:

- If $\varepsilon(\langle \text{parent}(x), x\rangle)$ = `/self::` then there is only a single candidate fragment f_i (Algorithm 2, line 9).
- If $\varepsilon(\langle \text{parent}(x), x\rangle)$ = `/`, then the algorithm determines all types $\sigma \in \Sigma$ that are directly reachable in the schema from the type corresponding to the node test in parent(x) (i.e., the types σ to which there is an edge from $\nu(\text{parent}(x))$ in the schema).

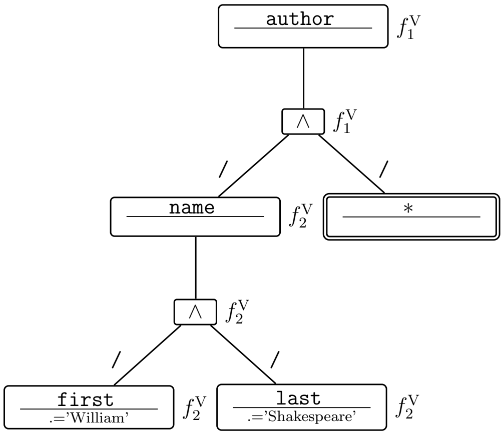


Figure 5.3: Partially annotated QTP representation of query q_4

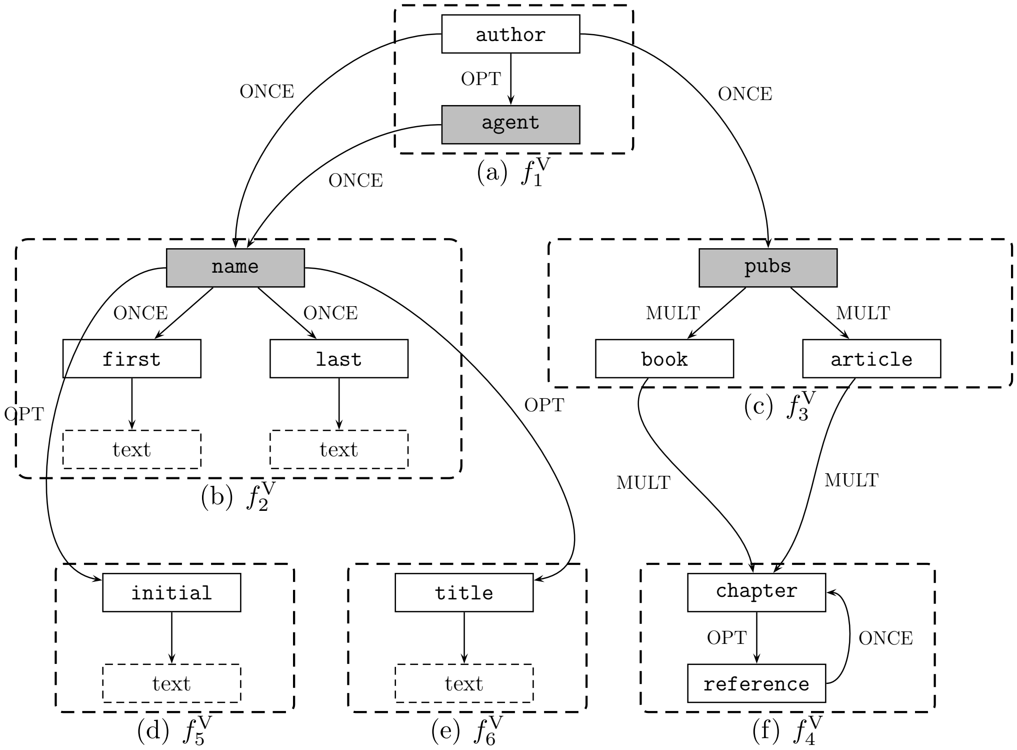


Figure 5.4: Vertical fragmentation schema with reachable nodes highlighted

Algorithm 2: split-wildcard(x) *splits wildcard node in QTP*

```
    input  : QTP node with wildcard node test $x$
    output : Set $X$ consisting of pairs of pattern nodes and fragments into which $x$ was split
1   if $x = r$ then
2       // $x$ is the pattern root
3       return $\{x, f_\rho^{\mathrm{V}}\}$
4   else
5       // $x$ is not the pattern root
6       $x_{\text{parent}} \leftarrow \text{parent}(x)$
7       $\varepsilon_{\text{in}} \leftarrow \varepsilon(\langle x_{\text{parent}}, x\rangle)$
8       if $\varepsilon_{\text{in}} = $ /self:: then
9           return $\{\langle x, a(x_{\text{parent}})\rangle\}$
10      else
11          if $\nu(x_{\text{parent}}) \in \Sigma$ then
12              // parent node is non-wildcard node
13              $\Sigma_{\text{reachable}} \leftarrow$ set of types reachable from $\nu(x_{\text{parent}})$ via a $\varepsilon_{\text{in}}$ step
14              $F_{\text{reachable}} \leftarrow$ set of fragments corresponding to types in $\Sigma_{\text{reachable}}$
15          else
16              // parent node is wildcard node
17              $F_{\text{reachable}} \leftarrow$ set of fragments reachable from $a(x_{\text{parent}})$ in schema graph
18          $E \leftarrow E \setminus \{\langle x_{\text{parent}}, x\rangle\}$
19          $v \leftarrow$ new logic node($\vee$)
20          $E \leftarrow E \cup \{\langle x_{\text{parent}}, v\rangle\}$
21          $X \leftarrow \emptyset$
22          for $f_{\text{reachable}} \in F_{\text{reachable}}$ do
23              $x_\sigma \leftarrow$ copy sub-pattern($x$)
24              $E \leftarrow E \cup \{\langle v, x_\sigma\rangle\}$
25              $\varepsilon(\langle v, x_\sigma\rangle) \leftarrow \varepsilon_{\text{in}}$
26              $X \leftarrow X \cup \{\langle x_\sigma, f_{\text{reachable}}\rangle\}$
27          return $X$
```

This is the case encountered in the example in Figure 5.3. As can be seen, the highlighted types in the schema in Figure 5.4 correspond to the types that are directly reachable from `author`. Thus, the algorithm determines the set of fragments that correspond to at least one reachable type and introduces a duplicate pattern node for each such fragment (Algorithm 2, line 14).

- Similarly, if $\varepsilon(\langle \text{parent}(x), x\rangle) =$ `//`, then the algorithm determines all types $\sigma \in \Sigma$ that are directly or indirectly reachable in the schema from the type corresponding to

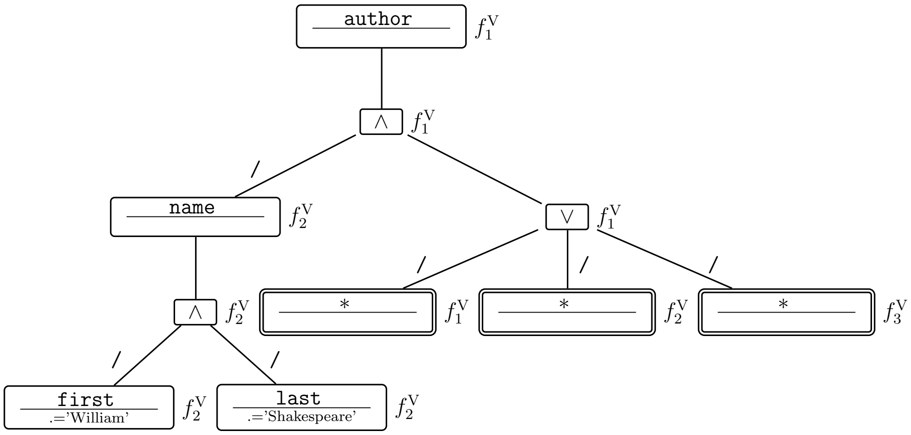


Figure 5.5: Annotated QTP representation of query q_4

the node test in parent(x) (i.e., the types σ to which there is a path from ν(parent(x)) in the schema, Algorithm 2, lines 13). As in the previous case, it then determines the set of fragments corresponding to these types (Algorithm 2, line 14).

- If parent(x) also has wildcard node test, then the algorithm falls back to returning all fragments reachable in the schema graph from the fragment assigned to parent(x) (Algorithm 2, line 17).

The duplicated pattern nodes are connected by a disjunction and inserted into the pattern (Algorithm 2, lines 18–25). Finally, each copy of the pattern node is assigned to one candidate fragment (Algorithm 1, lines 11–12).

When evaluating query q_4, the pattern node with the wildcard node test needs to be matched to a node in fragment f_1^V, f_2^V, or f_3^V. Thus, after applying Algorithm 2, the QTP (shown in Figure 5.5) contains three copies of this pattern node (one for each of the three fragments) that are connected by a disjunction logic node. Since the pattern node with the wildcard node test is designated as an extraction point in query q_4, this violates the constraint that the path from the pattern root to an extraction point node may consist of only pattern nodes and ∧ logic nodes. To address this problem, special attention is needed when decomposing this QTP, as is described in Section 5.2.5.

5.2.2 Decomposing QTPs

Once the QTP is fully annotated, it is divided into maximal contiguous sub-patterns consisting of nodes assigned to a single fragment. As shown in Figure 5.6, for query q_1, there are three such sub-patterns, corresponding to fragments f_1^{V}, f_2^{V}, and f_4^{V}, respectively. To enable distributed processing, each of these sub-patterns is converted into a separate local QTP, which can then be evaluated over a single fragment.

In order for this decomposition not to alter the semantics of the query, it is necessary to represent pattern edges that cross fragment boundaries (denoted by dotted lines in Figure 5.6 and referred to as *cross-fragment steps*). How this is done depends on the XPath axis associated with the edge. Since self edges never cross fragment boundaries, it is necessary to consider two cases, corresponding to child and descendant axes, respectively.

A child edge from a node in a sub-pattern corresponding to fragment f_i^{V} to a node in a sub-pattern corresponding to fragment f_j^{V} is converted to a pattern node matching a proxy in the local QTP corresponding to fragment f_i^{V} (referred to as a *proxy pattern node*) and a pattern node matching a root proxy in the local QTP corresponding to fragment f_j^{V} (referred to as a *root proxy pattern node*). These new pattern nodes are marked as extraction points because they are needed to join the results of local QTPs to generate the final result.

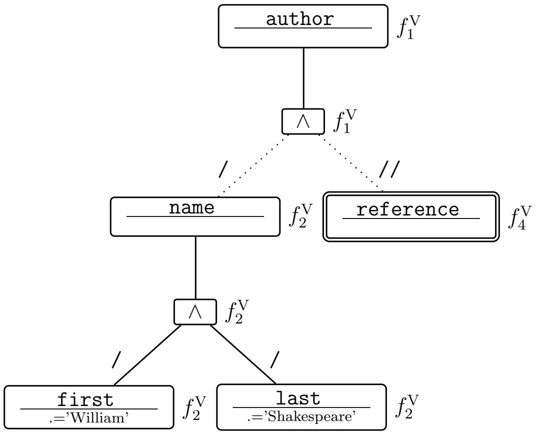
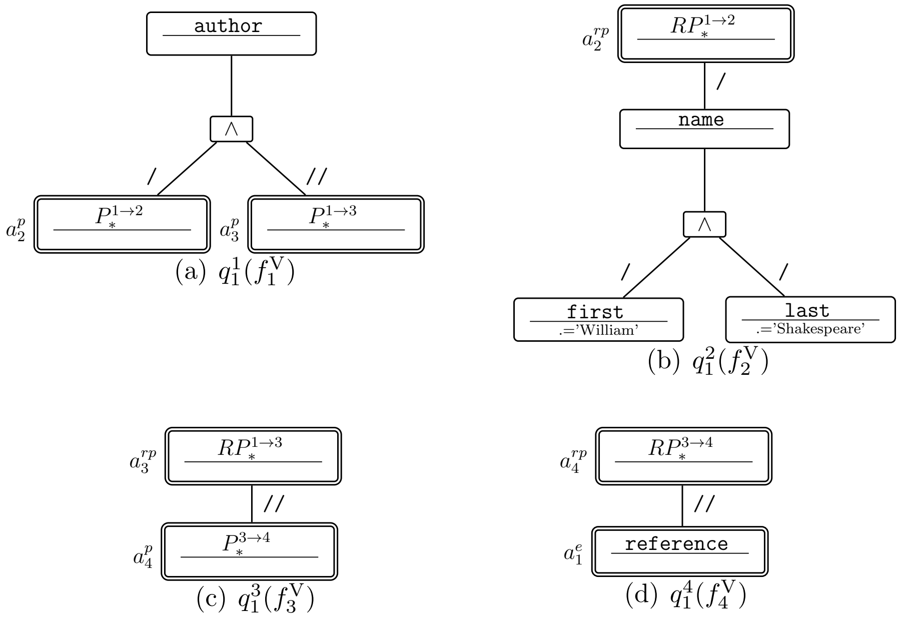


Figure 5.6: Decomposed QTP representation of query q_1



(a) $q_1^1(f_1^{\mathrm{V}})$

(b) $q_1^2(f_2^{\mathrm{V}})$

(c) $q_1^3(f_3^{\mathrm{V}})$

(d) $q_1^4(f_4^{\mathrm{V}})$

Figure 5.7: Local QTPs corresponding to query q_1

Figure 5.7 shows the local QTPs for query q_1. Each local QTP is labeled with the query from which it is derived, a unique identifier for each QTP, and the fragment to which the QTP corresponds. $q_1^2(f_2^{\mathrm{V}})$, for example, refers to local QTP number 2, derived from query q_1 and corresponding to fragment f_2^{V}.

As can be seen, the child edge from the logic node labeled $\wedge$ to the pattern node with the node test `name` is represented as follows:

- In the local QTP corresponding to fragment f_1^{V} (denoted as $q_1^1(f_1^{\mathrm{V}})$), a proxy pattern node is inserted (matching $P_*^{1\to2}$). Since the original cross-fragment edge is a child edge, the edge leading to the proxy pattern node is also a child edge.
- In local QTP $q_1^2(f_2^{\mathrm{V}})$, a root proxy pattern node is inserted (matching $RP_*^{1\to2}$). Again, since the cross-fragment edge is a child edge, the edge leading from the newly inserted root proxy pattern node is also a child edge.

Descendant edges across fragment boundaries are similarly represented by pairs of proxy pattern nodes and root proxy pattern nodes, except that the edges to and from these

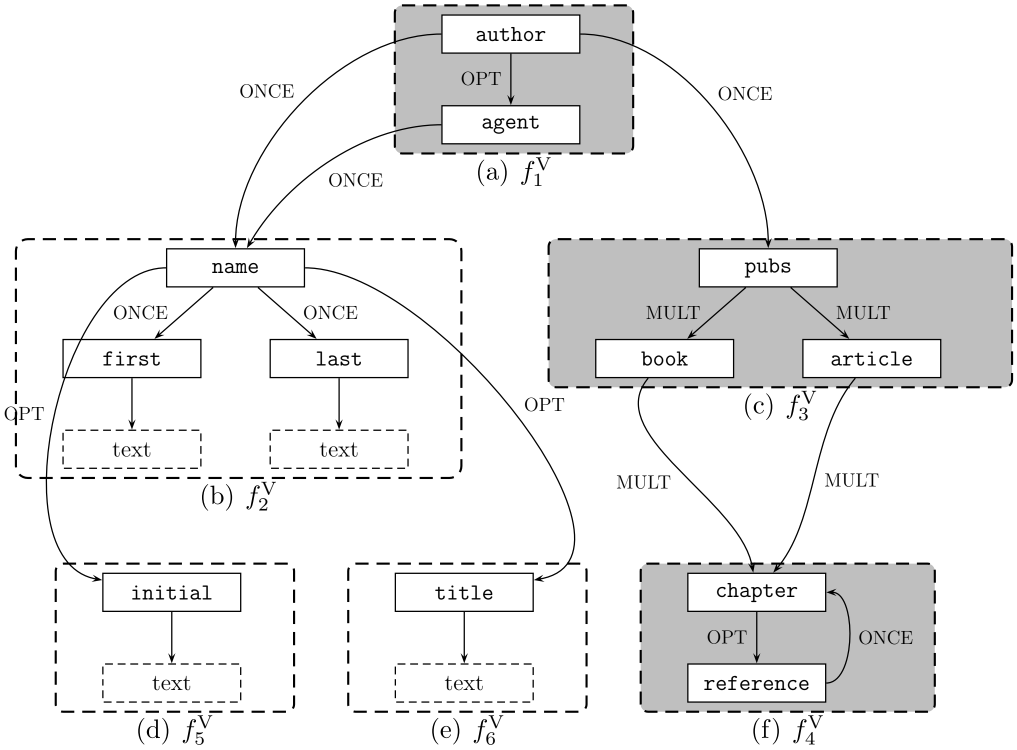


Figure 5.8: Fragments on path between f_1^{V} and f_4^{V}

pattern nodes are descendant edges. However, one complication arises from the far-reaching nature of XPath descendant steps: it is possible that a descendant step may cross multiple fragments. Therefore, it is necessary to identify all paths in the fragmentation schema that satisfy the descendant edge. Consider, for example, the descendant edge from the pattern node with the node test `author` in fragment f_1^{V} to the pattern node with the node test `reference` in fragment f_4^{V} (shown in Figure 5.6). To determine the fragments traversed by this edge, the fragmentation schema needs to be inspected for paths from f_1^{V} to f_4^{V}. As shown in Figure 5.8, there is exactly one path from f_1^{V} to f_4^{V}, which traverses fragment f_3^{V}. Thus, an additional local QTP corresponding to fragment f_3^{V} is introduced ($q_1^3(f_3^{\mathrm{V}})$ in Figure 5.7), despite the fact that there is no pattern node in the global QTP that refers to node types in this fragment. In cases where there is more than one path in the fragmentation schema, $\vee$ logic nodes are inserted and each path is decomposed into local QTPs separately.

To avoid ambiguity between proxy pattern nodes, root proxy pattern nodes, and other extraction points and to distinguish between multiple proxy pattern nodes that match proxy nodes corresponding to document edges between the same pair of fragments, all extraction points in the local QTPs corresponding to a query are given labels that are unique within that query. The labels are assigned based on the following rules.

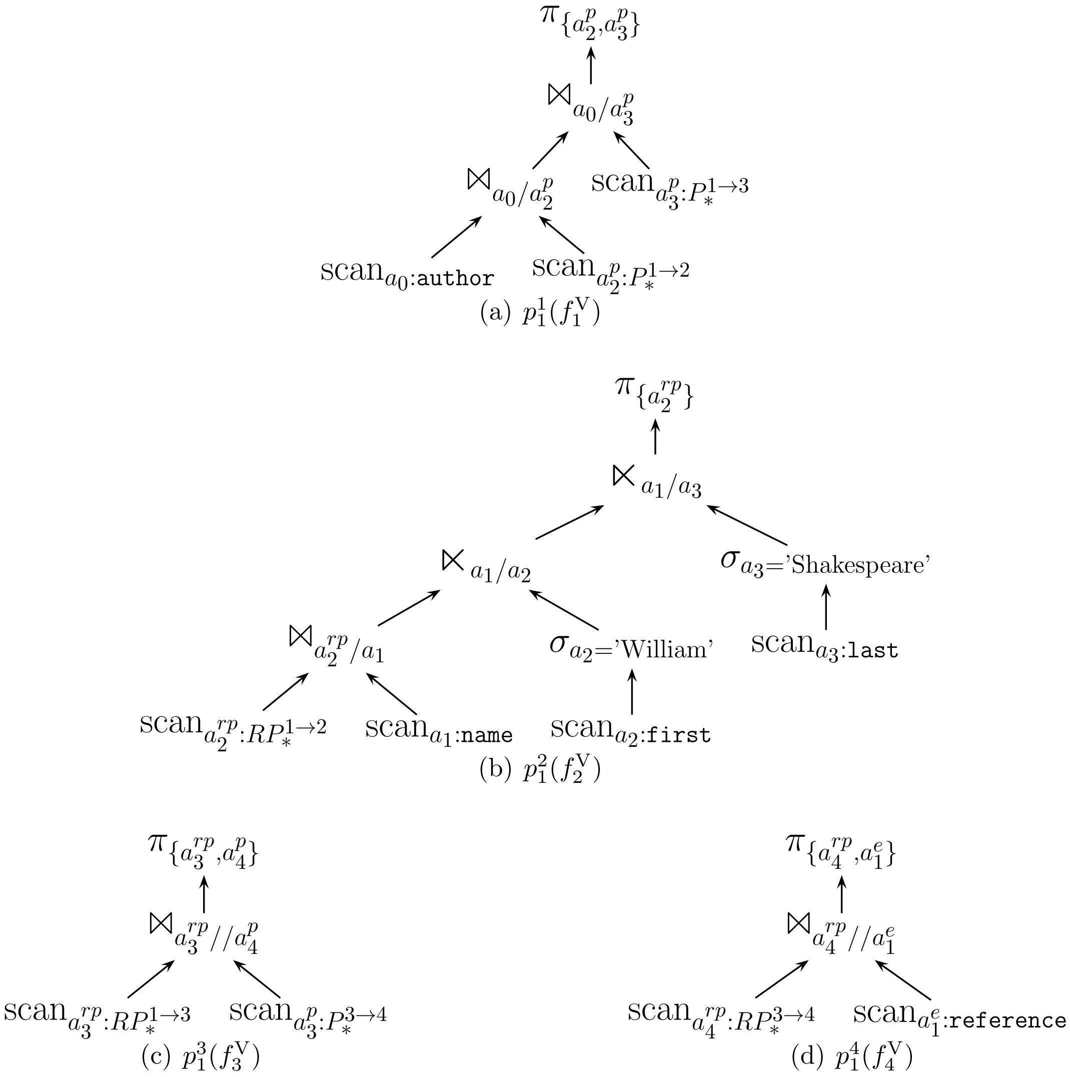
- A root proxy pattern node in local QTP $q_k^u(f_i^{\mathrm{V}})$ is labeled a_u^{rp}.
- Similarly, the proxy pattern node in local QTP $q_k^v(f_j^{\mathrm{V}})$ that corresponds to the same cross-fragment edge as the root proxy pattern node a_u^{rp} in $q_k^u(f_i^{\mathrm{V}})$ is labeled a_u^p.
- The ith extraction point in the global QTP representation of the query is labeled a_i^e.

Whenever a QTP is decomposed, all but one of the resulting local QTPs are rooted at a root proxy pattern node. One local QTP, however, is always rooted at a non-root proxy pattern node. In the following this unique local QTP is referred to as the *root QTP*. In the example shown in Figure 5.7, $q_1^1(f_1)$ is the root QTP. If local QTP $q_k^v(f_j^{\mathrm{V}})$ contains a root proxy pattern node matching $RP_*^{i \to j}$ and local QTP $q_k^u(f_i^{\mathrm{V}})$ contains the corresponding proxy pattern node matching $P_*^{i \to j}$, then $q_k^u(f_i^{\mathrm{V}})$ is referred to as a *parent QTP* of $q_k^v(f_j^{\mathrm{V}})$ and $q_k^v(f_j^{\mathrm{V}})$ is referred to as a *child QTP* of $q_k^u(f_i^{\mathrm{V}})$.

If the query does not reach a certain fragment (because no pattern nodes are annotated with this fragment) and if no intermediate QTP is generated for this fragment because of cross-fragment descendant steps, then distributed query evaluation will not access this fragment. Therefore, unlike in the horizontal case, even this initial strategy for querying vertically fragmented collections avoids accessing some vertical fragments. The pruning techniques presented in Section 6.2.1 expand upon this idea and identify additional cases where it is possible to avoid accessing certain fragments.

5.2.3 Converting Local QTPs to LQPs

In the next step, each local QTP $q_k^u(f_i^{\mathrm{V}})$ is transformed into an LQP $p_k^u(f_i^{\mathrm{V}})$. This is done at the site holding the corresponding fragment f_i^{V}, using techniques for the centralized


$\pi_{\{a_2^p, a_3^p\}}$
$\bowtie_{a_0/a_3^p}$
$\bowtie_{a_0/a_2^p}$
$\text{scan}_{a_3^p:P_*^{1\to3}}$
$\text{scan}_{a_0:\texttt{author}}$
$\text{scan}_{a_2^p:P_*^{1\to2}}$
$\pi_{\{a_2^{rp}\}}$
$\ltimes_{a_1/a_3}$
$\ltimes_{a_1/a_2}$
$\sigma_{a_3=\text{'Shakespeare'}}$
$\bowtie_{a_2^{rp}/a_1}$
$\sigma_{a_2=\text{'William'}}$
$\text{scan}_{a_3:\texttt{last}}$
$\text{scan}_{a_2^{rp}:RP_*^{1\to2}}$
$\text{scan}_{a_1:\texttt{name}}$
$\text{scan}_{a_2:\texttt{first}}$
$\pi_{\{a_3^{rp}, a_4^p\}}$
$\bowtie_{a_3^{rp}//a_4^p}$
$\text{scan}_{a_3^{rp}:RP_*^{1\to3}}$
$\text{scan}_{a_3^p:P_*^{3\to4}}$
$\pi_{\{a_4^{rp}, a_1^e\}}$
$\bowtie_{a_4^{rp}//a_1^e}$
$\text{scan}_{a_4^{rp}:RP_*^{3\to4}}$
$\text{scan}_{a_1^e:\texttt{reference}}$

(a) $p_1^1(f_1^{\mathrm{V}})$

(b) $p_1^2(f_2^{\mathrm{V}})$

(c) $p_1^3(f_3^{\mathrm{V}})$

(d) $p_1^4(f_4^{\mathrm{V}})$

Figure 5.9: LQPs for query q_1

evaluation of tree pattern queries such as those described in Section 3.2.2. If $q_k^u(f_i^{\mathrm{V}})$ is the root QTP then $p_k^u(f_i^{\mathrm{V}})$ is the *root LQP*. Similarly, if $q_k^v(f_j^{\mathrm{V}})$ is a child QTP of $q_k^u(f_i^{\mathrm{V}})$ then $p_k^v(f_j^{\mathrm{V}})$ is a *child LQP* of $p_k^u(f_i^{\mathrm{V}})$ and $p_k^u(f_i^{\mathrm{V}})$ is a *parent LQP* of $p_k^v(f_j^{\mathrm{V}})$.

For the purpose of illustration, Figure 5.9 shows a set of LQPs that correspond to the local QTPs depicted in Figure 5.7. The plans shown are based on structural joins but, as mentioned before, this is not a requirement.

For greater clarity, the tuple attributes in LQPs are unique within the context of a query (i.e., an attribute from one LQP is not re-used in another LQP corresponding to the same query). Attributes corresponding to extraction points in the local QTPs receive the same labels as their corresponding pattern nodes, other attributes are assigned sequential identifiers. This results in the following labeling system.

- a_i^e refers to the attribute that holds the ith extraction point in the QTP representation of the query (before decomposition). In Figure 5.9, there is one instance of this in LQP $p_1^4(f_4^{\mathrm{V}})$.
- a_v^{rp} refers to the attribute that holds the root proxy nodes matched to the root proxy pattern node in LQP $p_k^v(f_j^{\mathrm{V}})$.
- Similarly, a_v^p refers to the attribute that holds the proxy nodes matched by LQP $p_k^u(f_i^{\mathrm{V}})$ that correspond to the root proxy nodes matched by $p_k^v(f_j^{\mathrm{V}})$.
- All other attributes (i.e, the attributes holding non-extraction point and non-proxy nodes) are labeled a_n. As can be seen in Figure 5.9, these attributes are only used internally within a single LQP and then projected away at the root of this plan.

5.2.4 Distributed Execution Plans

To obtain the overall query result, the results derived from each LQP need to be combined to the overall query result. The results of two LQPs can be combined directly if one is the parent LQP of the other. To combine the results of LQP $p_k^u(f_i^{\mathrm{V}})$ with that of its child LQP $p_k^v(f_j^{\mathrm{V}})$, a join is performed between the results of both LQPs (represented as sequences of tuples). Since attribute a_v^p in the result tuples of $p_k^u(f_i^{\mathrm{V}})$ contains the proxy nodes matching $P_*^{i \to j}$ and attribute a_v^{rp} in the result tuples of $p_k^v(f_j^{\mathrm{V}})$ contains the root proxy nodes matching $RP_*^{i \to j}$, $p_k^u(f_i^{\mathrm{V}})$ and $p_k^v(f_j^{\mathrm{V}})$ can be combined by the join $p_k^u(f_i^{\mathrm{V}}) \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} p_k^v(f_j^{\mathrm{V}})$ where id is a function that extracts the ID from a proxy or root proxy node. We refer to this join as a *cross-fragment join*.

A *distributed execution plan* (DEP) specifies how this strategy is applied to all LQPs corresponding to a query. Formally, a distributed execution plan is defined as follows.

Definition 5.2. Let $P = \{p_k^1(f_i^{\mathrm{V}}), \ldots, p_k^n(f_j^{\mathrm{V}})\}$ be the set of LQPs corresponding to a query q_k. Further let $P' \subseteq P$. Then $G_{P'}$ is a *distributed execution plan* (abbreviated as DEP) for P' iff

1. $P' = \{p_k^u(f_i^{\mathrm{V}})\}$ and $G_{P'} = p_k^u(f_i^{\mathrm{V}})$ (i.e., $G_{P'}$ consists of a single LQP), or
2. $P' = P_u \cup P_v$, $P_u \cap P_v = \emptyset$; $p_k^u(f_i^{\mathrm{V}}) \in P_u, p_k^v(f_j^{\mathrm{V}}) \in P_v$, $p_k^u(f_i^{\mathrm{V}}) = \mathrm{parent}(p_k^v(f_j^{\mathrm{V}}))$; G_{P_u} and G_{P_v} are DEPs for P_u and P_v returning the sets of attributes $A\left(G_{P_u}\right)$ and $A\left(G_{P_v}\right)$, respectively; and $G_{P'} = \pi_{(A(G_{P_u}) \cup A(G_{P_v}))\setminus\{a_v^p, a_v^{rp}\}}(G_{P_u} \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v})$ (i.e., $G_{P'}$ is composed of two distributed execution plans G_{P_u} and G_{P_v} connected by a join between a parent LQP $p_k^u(f_i^{\mathrm{V}})$ in G_{P_u} and a child LQP $p_k^v(f_j^{\mathrm{V}})$ in G_{P_v}, followed by a projection that removes the proxy/root proxy nodes used in the join).

If $G_{P'}$ consists of a single LQP $p_k^u(f_i^{\mathrm{V}})$, then the set of attributes returned by $G_{P'}$ (denoted as $A\left(G_{P'}\right)$) is identical to the set of attributes returned by $p_k^u(f_i^{\mathrm{V}})$. If $G_{P'} = \pi_{(A(G_{P_u}) \cup A(G_{P_v}))\setminus\{a_v^p, a_v^{rp}\}}(G_{P_u} \bowtie_{\mathrm{id}(a_n^p)=\mathrm{id}(a_v^{rp})} G_{P_v})$, then $A\left(G_{P'}\right) = \left(A\left(G_{P_u}\right) \cup A\left(G_{P_v}\right)\right) \setminus \{a_v^p, a_v^{rp}\}$.

If G_P is a DEP for P (the entire set of LQPs), then $G_{q_k} = \pi^D(G_P)$ is a DEP for the query q_k. ■

A DEP must contain all the LQPs corresponding to the query. As shown in the recursive definition above, a DEP for a single LQP is simply the LQP itself (condition 1). For a set of multiple LQPs P', P_u and P_v are assumed to be two non-overlapping subsets of P' such that $P_u \cup P_v = P'$. The definition requires that P_u contains the parent LQP $p_k^u(f_i^{\mathrm{V}})$ for some LQP $p_k^v(f_j^{\mathrm{V}})$ in P_v. A DEP for P' is then defined by combining DEPs for P_u and P_v using a join whose predicate compares the IDs of root proxy nodes derived from $p_k^v(f_j^{\mathrm{V}})$ to the IDs of corresponding proxy nodes derived from $p_k^u(f_i^{\mathrm{V}})$ (condition 2).

After the results of all LQPs have been combined by joins, it may be necessary to eliminate duplicates from the query result. While local query execution avoids the creation of duplicates within the result derived from a single fragment, additional duplicates may be inserted when these results are joined together. Thus, in general, DEPs require an additional duplicate elimination operator (π^D) at their root.

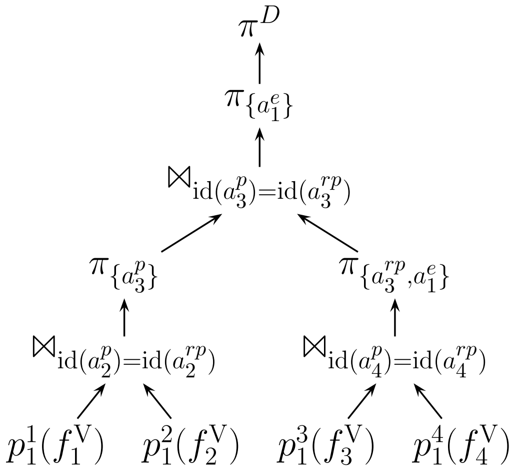

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_3^p)=\mathrm{id}(a_3^{rp})}$
$\pi_{\{a_3^p\}}$
$\pi_{\{a_3^{rp},a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$\bowtie_{\mathrm{id}(a_4^p)=\mathrm{id}(a_4^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$
$p_1^3(f_3^{\mathrm{V}})$
$p_1^4(f_4^{\mathrm{V}})$

Figure 5.10: DEP for query q_1

Figure 5.10 shows an example of a DEP for query q_1, which combines the results of the LQPs shown in Figure 5.9. There are usually many different DEPs that all yield the correct result but that may differ significantly in query performance. Methods for improving the performance of DEPs are discussed in Chapter 6.

5.2.5 Handling Disjunction

The techniques for decomposing QTPs and generating DEPs mentioned above work well for QTPs that consist solely of pattern nodes and $\wedge$ logic nodes (i.e., conjunction). To handle $\vee$ logic nodes (disjunction), extra steps are necessary.

To decompose a query with $\vee$ logic nodes, the QTP is first annotated as described in Section 5.2.1. Note that this may introduce additional $\vee$ logic nodes if wildcard nodes are split. After annotation, there are two scenarios to consider.

Case 1 If all descendants of a $\vee$ logic node are annotated with the same fragment as the $\vee$ logic node itself, then the QTP is decomposed normally. This is the case for

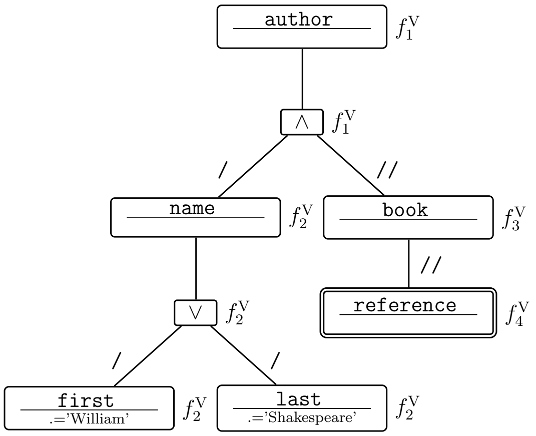
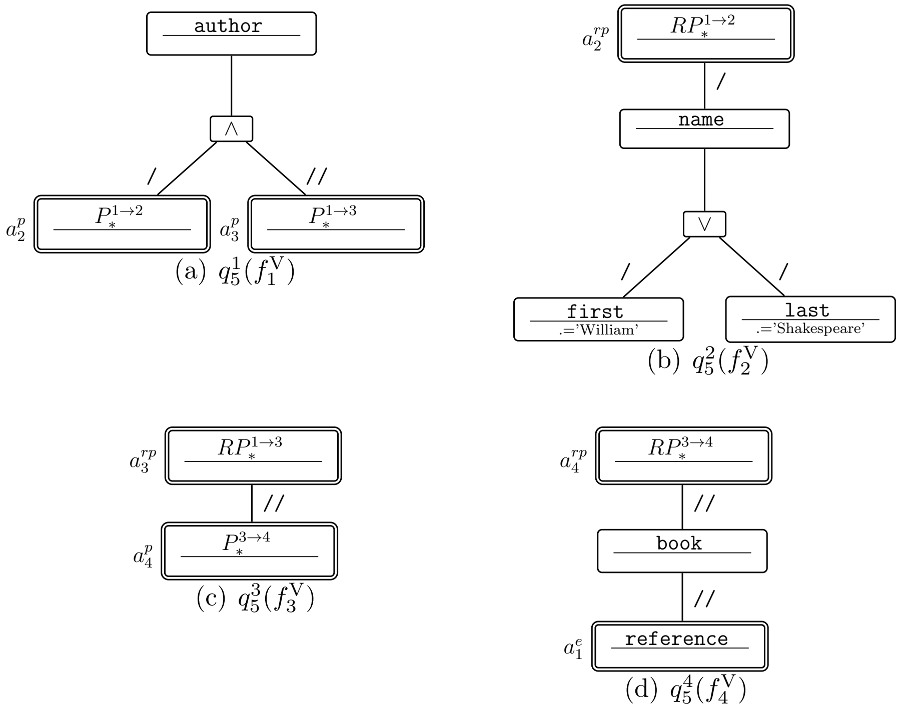


Figure 5.11: Annotated QTP representation of query q_5



(a) $q_5^1(f_1^{\vee})$

(b) $q_5^2(f_2^{\vee})$

(c) $q_5^3(f_3^{\vee})$

(d) $q_5^4(f_4^{\vee})$

Figure 5.12: Local QTPs corresponding to query q_5

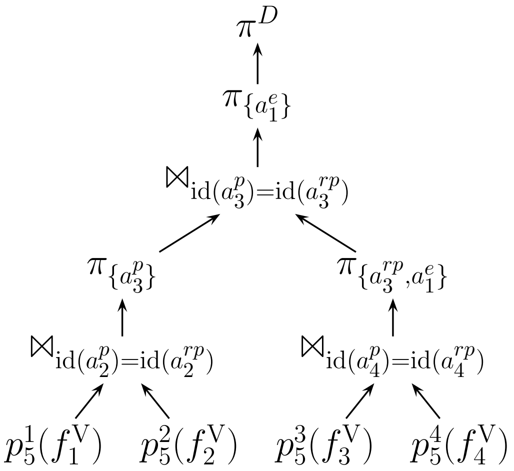
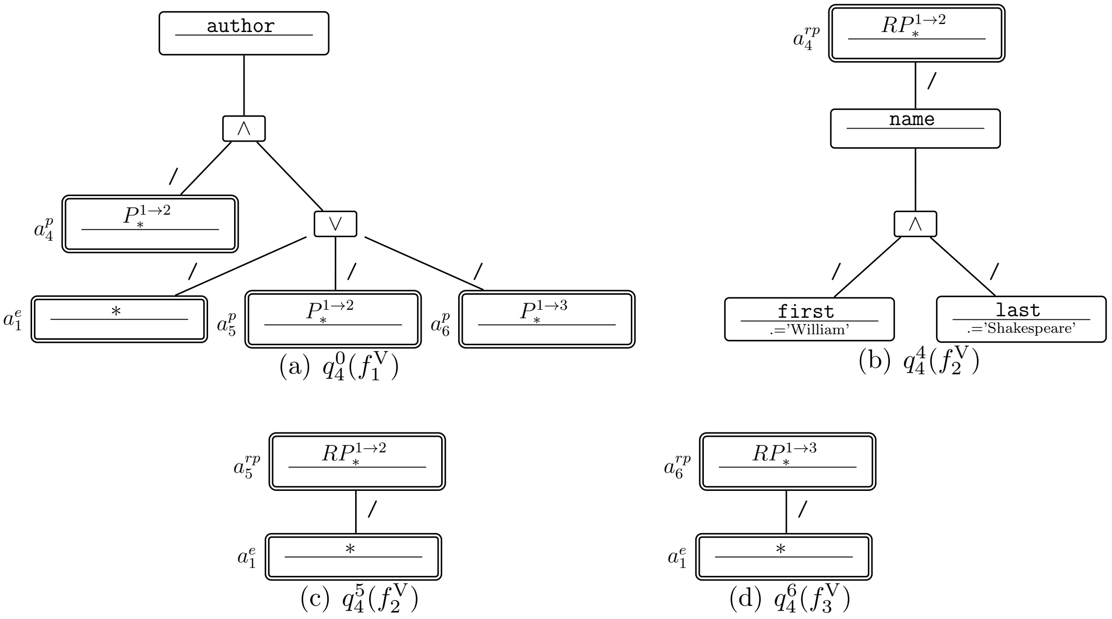

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_3^p)=\mathrm{id}(a_3^{rp})}$
$\pi_{\{a_3^p\}}$
$\pi_{\{a_3^{rp},a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$\bowtie_{\mathrm{id}(a_4^p)=\mathrm{id}(a_4^{rp})}$
$p_5^1(f_1^{\mathrm{V}})$
$p_5^2(f_2^{\mathrm{V}})$
$p_5^3(f_3^{\mathrm{V}})$
$p_5^4(f_4^{\mathrm{V}})$

Figure 5.13: DEP for query q_5

query q_5, whose annotated QTP is shown in Figure 5.11. As can be seen the $\vee$ logic node in this query is annotated with fragment f_2^{V} and all of its descendants are also annotated with this fragment. Decomposing this QTP yields the local QTPs shown in Figure 5.12. Note that, after decomposition and the insertion of pattern nodes matching proxy nodes, all local QTPs are valid and there are no extraction point nodes below the $\vee$ logic node. After converting each local QTP to an LQP, query q_5 can be evaluated by the DEP shown in Figure 5.13.

Case 2 If a $\vee$ logic node has at least one descendant node annotated with a different fragment, applying the decomposition strategy described above introduces additional proxy pattern nodes. Since proxy pattern nodes are always designated as extraction points, there are now extraction points below the $\vee$ logic node, violating the definition of QTPs. Consider, for example, query q_4, whose annotated QTP is shown in Figure 5.5 on page 84. Applying the decomposition strategy described above yields the local QTPs shown in Figure 5.14. As can be seen, this results in three extraction point nodes below the $\vee$ logic node in $q_4^0(f_1^{\mathrm{V}})$, thus rendering this local QTP invalid.

To solve the problem in the second scenario, the local QTP is split into one local QTP for each branch of the offending disjunction. This results in a set of local QTPs that are copies of the original (invalid) local QTP. However, in place of the disjunction, each



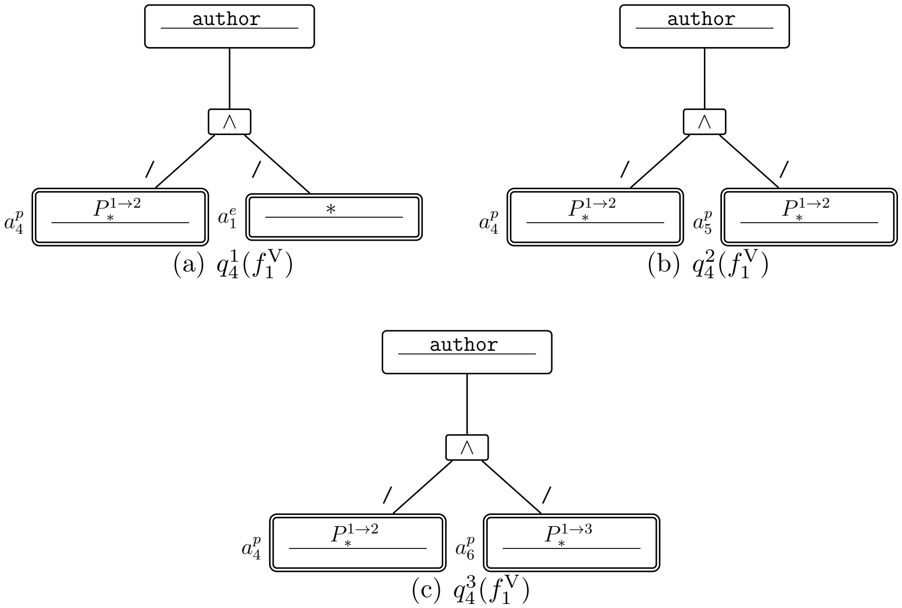
(a) $q_4^0(f_1^{\mathrm{V}})$

(b) $q_4^4(f_2^{\mathrm{V}})$

(c) $q_4^5(f_2^{\mathrm{V}})$

(d) $q_4^6(f_3^{\mathrm{V}})$

Figure 5.14: Local QTPs corresponding to query q_4, with invalid local QTP $q_4^0(f_1^{\mathrm{V}})$



(a) $q_4^1(f_1^{\mathrm{V}})$

(b) $q_4^2(f_1^{\mathrm{V}})$

(c) $q_4^3(f_1^{\mathrm{V}})$

Figure 5.15: Local QTPs resulting from splitting $q_4^0(f_1^{\mathrm{V}})$

resulting local QTP instead contains just one of the branches of the disjunction. Applying this to $q_4^0(f_1^{\mathrm{V}})$, yields the three local QTPs shown in Figure 5.15. Note that splitting local QTPs also duplicates extraction points with the same label. Consider, for example, the three extraction points labeled a_1^e, which all correspond to the single extraction point of the original query.

To preserve the semantics of the query, the DEP combines the LQPs resulting from the split by merging the sequences of tuples resulting from each of them. More formally, this can be expressed as follows.

Definition 5.3. Let P' be a subset of the LQPs corresponding to a query q_k such that $P' = P_u \cup \ldots \cup P_v$ and $P_u, \ldots, P_v$ are pairwise disjoint. If $\exists p_k^u(f_i^{\mathrm{V}}) \in P_u, \ldots, p_k^v(f_j^{\mathrm{V}}) \in P_v$ that all result from the same QTP split and if $G_{P_u}, \ldots, G_{P_v}$ are DEPs for $P_u, \ldots, P_v$, respectively, such that $A(G_{P_u}) = \ldots = A(G_{P_v})$ then $G_{P'} = G_{P_u} \odot \ldots \odot G_{P_v}$ is a DEP for P' and $A(G_{P'}) = A(G_{P_u}) = \ldots = A(G_{P_v})$. ■

Note that the definition requires that in order for a set of DEPs to be combined by a merge operator, they all have to yield result tuples consisting of the same set of attributes.

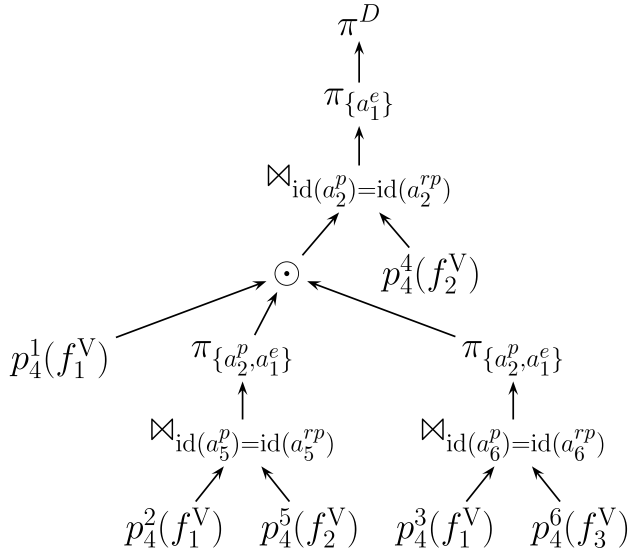

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$\odot$
$p_4^4(f_2^{\mathrm{V}})$
$p_4^1(f_1^{\mathrm{V}})$
$\pi_{\{a_2^p,a_1^e\}}$
$\pi_{\{a_2^p,a_1^e\}}$
$\bowtie_{\mathrm{id}(a_5^p)=\mathrm{id}(a_5^{rp})}$
$\bowtie_{\mathrm{id}(a_6^p)=\mathrm{id}(a_6^{rp})}$
$p_4^2(f_1^{\mathrm{V}})$
$p_4^5(f_2^{\mathrm{V}})$
$p_4^3(f_1^{\mathrm{V}})$
$p_4^6(f_3^{\mathrm{V}})$

Figure 5.16: DEP for query q_4

In the DEP for query q_4 shown in Figure 5.16, this means that the joins between $p_4^2(f_1^{\mathrm{V}})$ and $p_4^5(f_2^{\mathrm{V}})$ and between $p_4^3(f_1^{\mathrm{V}})$ and $p_4^6(f_3^{\mathrm{V}})$ have to be performed before the merge.

While splitting fragments with disjunction makes it possible to distribute the processing of queries even where the disjunction would otherwise prevent this, in some cases, the duplication of local QTPs may increase the processing cost at the site corresponding to the split local QTP. Section 6.2.4 presents an optimization technique that addresses this by evaluating the shared portions of split local QTPs simultaneously.

5.2.6 Handling Negation

Negation also requires special attention during QTP decomposition and during the generation of DEPs. This is because, as in the case of disjunction, the insertion of proxy pattern nodes potentially introduces extraction points below the negation, resulting in an invalid local QTP. For negation, there are three cases to be considered, and for each case there is a different solution.

Case 1 If all descendants of a $\neg$ logic node annotated with fragment f_i^{V} are annotated with the same fragment f_i^{V}, then no special treatment is necessary since the negation only affects pattern nodes within a single local QTP. Due to the fact that the overall query is required to be a valid QTP, all of the pattern nodes in the branch below the $\neg$ logic node are guaranteed not to be extraction points and thus the resulting local QTPs are also valid.

Case 2 If a $\neg$ logic node annotated with fragment f_i^{V} has as its child a node annotated with a different fragment f_j^{V}, then it is possible to combine the negation with the cross-fragment join between the fragments f_i^{V} and f_j^{V}. Section 5.2.6.1 describes how this is done.

Case 3 If a $\neg$ logic node annotated with fragment f_i^{V} has a child node that is also annotated with f_i^{V} and some descendant node that is annotated with a different fragment f_j^{V}, then rewrites are applied to the query until either case 1 or case 2 are satisfied. The rewrite rules are described in Section 5.2.6.2.

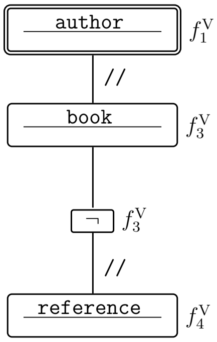


Figure 5.17: Annotated QTP corresponding to query q_3

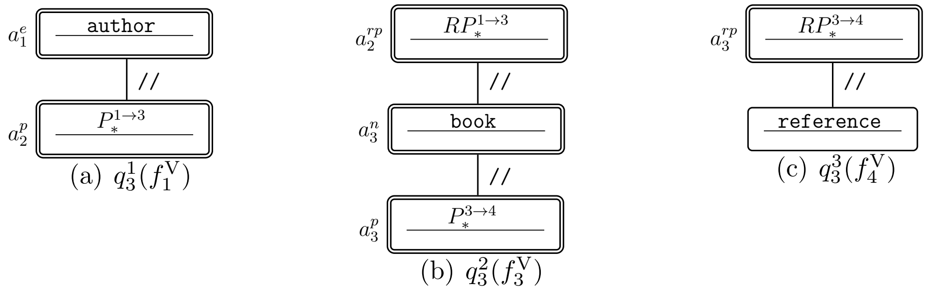
To show that these are the only three cases that may occur, consider a negation logic node l assigned to fragment f_i^{V} and let N be the set of nodes that occur as descendants of l in the pattern. If all nodes in N are also assigned to fragment f_i^{V}, then case 1 applies. Otherwise, there must be at least one node $n \in N$ such that n is assigned to some fragment f_j^{V} with $f_i^{\mathrm{V}} \neq f_j^{\mathrm{V}}$.

If there is a descendant node assigned to a different fragment, it is necessary to inspect the child of l in the pattern. Since each negation logic node has exactly one child node (as required by Definition 2.3 in Section 2.2), there is a unique child node n' of l. If n' is assigned to some fragment f_j^{V} with $f_i^{\mathrm{V}} \neq f_j^{\mathrm{V}}$, then case 2 applies.

If, however, n' is assigned to f_i^{V} (i.e., to the same fragment as l), then there must be some other node $n \in N$ that is assigned to fragment f_j^{V} with $f_i^{\mathrm{V}} \neq f_j^{\mathrm{V}}$. This corresponds directly to case 3.

5.2.6.1 Folding Negation Into Cross-Fragment Joins

If a ¬ logic node annotated with fragment f_i^{V} has as its child a node annotated with a different fragment f_j^{V}, then the negation can be rolled into the cross fragment join between the local QTPs $q_k^u(f_i^{\mathrm{V}})$ and $q_k^v(f_j^{\mathrm{V}})$. To do this, let a_v^n be the nearest pattern node ancestor of the ¬ logic node. If it is not marked as an extraction point already, a_v^n is marked as an extraction point. Next, the ¬ logic node is removed from the QTP. Then, the QTP is decomposed as described in Section 5.2.2.



(a) $q_3^1(f_1^{\mathrm{V}})$

(b) $q_3^2(f_3^{\mathrm{V}})$

(c) $q_3^3(f_4^{\mathrm{V}})$

Figure 5.18: Local QTPs corresponding to query q_3

Consider, for example, query q_3. As can be seen in the annotated QTP shown in Figure 5.17, the $\neg$ logic node is annotated with fragment f_3^{V}, whereas its child node, the pattern node with the node test `reference` is annotated with fragment f_4^{V}. Since the negation immediately precedes the step across the fragment boundary, it can be folded into the cross-fragment join. Thus, the $\neg$ logic node is removed from the QTP and the QTP is then decomposed, yielding the local QTPs shown in Figure 5.18. Note how the pattern node with the node test `book` in local QTP $q_3^2(f_3^{\mathrm{V}})$ is designated as an extraction point and labeled a_3^n.

When generating the DEP for the query, the negation is integrated into the join between the LQPs $p_k^u(f_i^{\mathrm{V}})$ and $p_k^v(f_j^{\mathrm{V}})$. This is done as follows.

Definition 5.4. Let P' be a subset of the LQPs corresponding to query q_k such that $P' = P_u \cup P_v$ and $P_u \cap P_v = \emptyset$. Further let $p_k^u(f_i^{\mathrm{V}}) \in P_u$ and $p_k^v \in P_v$ such that $a_v^n, a_v^p \in A\left(p_k^u\right)$, $a_v^{rp} \in A\left(p_k^v\right)$, and $A\left(G_{P_v}\right) = \{a_v^{rp}\}$. Then

$$G_{P'} := \pi_{A(G_{P_u})\setminus\{a_v^n,a_v^p,a_v^{rp}\}}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\mathrm{count}(a_v^{rp})}\left(G_{P_u} ⟕_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$$

is a DEP for P' and $A\left(G_{P'}\right) = A\left(G_{P_u}\right) \setminus \{a_v^n, a_v^p\}$. ■

Conceptually, the combination of the left outer join (⟕), the grouping/aggregation ($\mathbb{G} \ldots \mathbb{A}$), and the selection ($\sigma$) act in a way that is similar to an anti-join ($\triangleright$). Each tuple from the left-hand side is passed on if there is no matching tuple from the right-hand side. However, the DEP has to capture the scenario where a node matched to a_v^n has more than one proxy node as its descendant and a DEP based on anti-joins fails to do this.

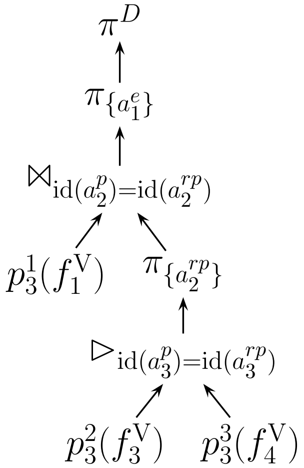
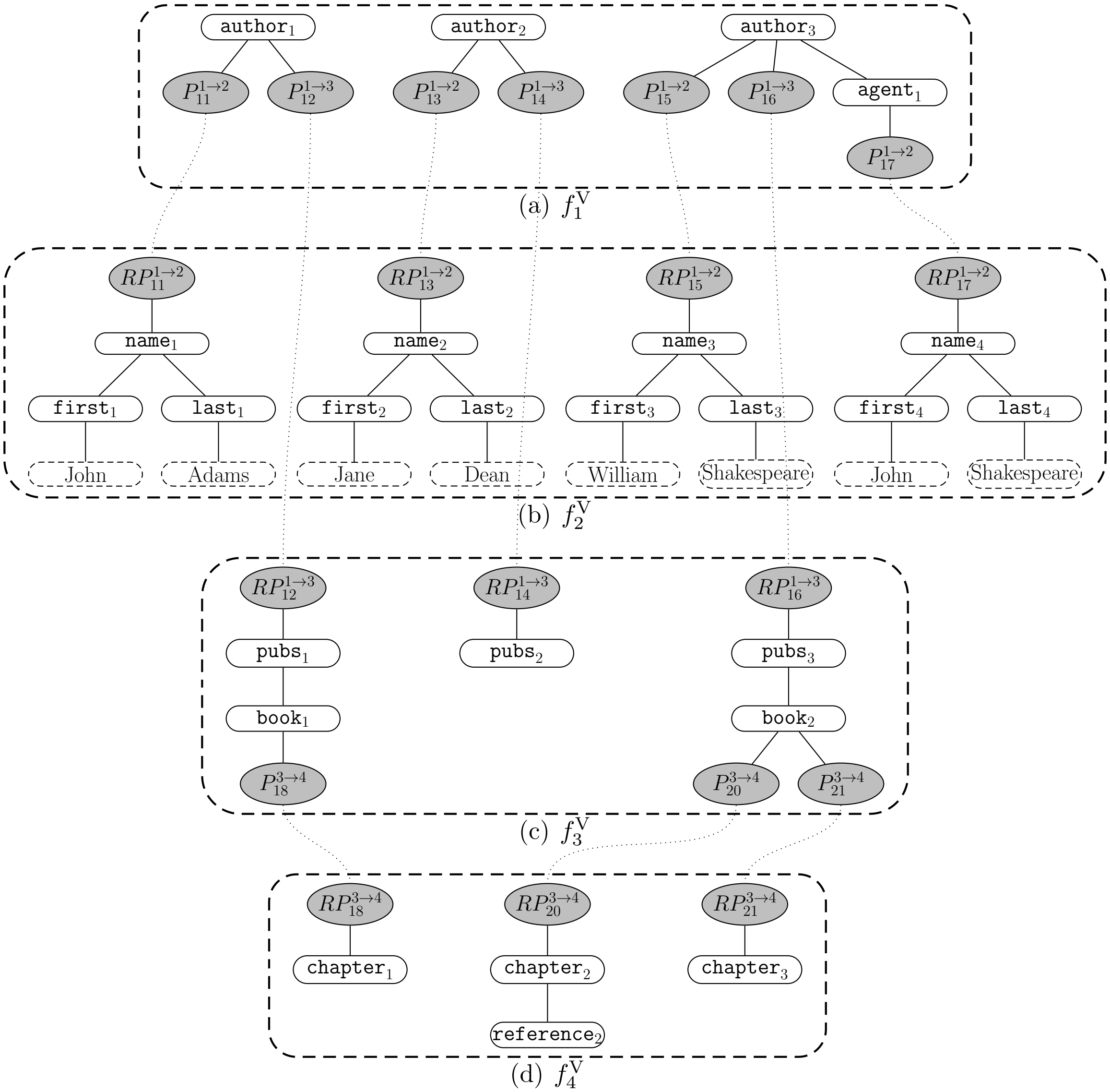

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_3^1(f_1^{\mathrm{V}})$
$\pi_{\{a_2^{rp}\}}$
$\triangleright_{\mathrm{id}(a_3^p)=\mathrm{id}(a_3^{rp})}$
$p_3^2(f_3^{\mathrm{V}})$
$p_3^3(f_4^{\mathrm{V}})$

Figure 5.19: Incorrect DEP for query q_3



(a) f_1^{V}

(b) f_2^{V}

(c) f_3^{V}

(d) f_4^{V}

Figure 5.20: A vertically fragmented collection

$$[a_1^e = \texttt{author}_1, a_2^p = P_{12}^{1\to 3}]$$
$$[a_1^e = \texttt{author}_2, a_2^p = P_{14}^{1\to 3}]$$
$$[a_1^e = \texttt{author}_3, a_2^p = P_{16}^{1\to 3}]$$

(a) $R(p_3^1(f_1^{\mathrm{V}}))$

$$[a_2^{rp} = RP_{12}^{1\to 3}, a_3^n = \texttt{book}_1, a_3^p = P_{18}^{3\to 4}]$$
$$[a_2^{rp} = RP_{16}^{1\to 3}, a_3^n = \texttt{book}_2, a_3^p = P_{20}^{3\to 4}]$$
$$[a_2^{rp} = RP_{16}^{1\to 3}, a_3^n = \texttt{book}_2, a_3^p = P_{21}^{3\to 4}]$$

(b) $R(p_3^2(f_3^{\mathrm{V}}))$

$$[a_3^{rp} = RP_{20}^{3\to 4}]$$

(c) $R(p_3^3(f_4^{\mathrm{V}}))$

Figure 5.21: LQP results for query q_3

To illustrate this point, consider query q_3. It is easy to see that the correct result for this query, when evaluated over the collection shown in Figure 5.20, consists only of $\texttt{author}_1$, since this is the only author node in the collection that has a book without references. In contrast to this, consider what happens when evaluating this query using the DEP shown in Figure 5.19, which contains a simple anti-join ($\rhd$) in place of the combination of operators mentioned above. Evaluating the LQPs $p_3^1(f_1^{\mathrm{V}})$, $p_3^2(f_3^{\mathrm{V}})$, and $p_3^3(f_4^{\mathrm{V}})$ yields the tuples shown in Figure 5.21 (denoted as $R(p_3^1(f_1^{\mathrm{V}}))$, $R(p_3^2(f_3^{\mathrm{V}}))$, and $R(p_3^3(f_4^{\mathrm{V}}))$, respectively). Performing the anti-join between $p_3^2(f_3^{\mathrm{V}})$ and $p_3^3(f_4^{\mathrm{V}})$ yields two tuples (shown after projection):

$$[a_2^{rp} = RP_{12}^{1\to 3}]$$
$$[a_2^{rp} = RP_{16}^{1\to 3}]$$

Finally, joining these tuples with the result of $p_3^1(f_1^{\mathrm{V}})$ yields two results:

$$[a_1^e = \texttt{author}_1]$$
$$[a_1^e = \texttt{author}_3]$$

Therefore, $\texttt{author}_3$ is incorrectly reported as part of the query result. The reason why this happens is that the anti-join matches all proxy nodes that lead to a chapter without a reference. Thus, all authors are reported that have books with at least one chapter without a reference (rather than authors with entire books without a reference, as specified in the query).

A correct DEP, as specified in the definition above, addresses this problem by making sure that the implicit quantifier is related to the pattern node that precedes the negation,

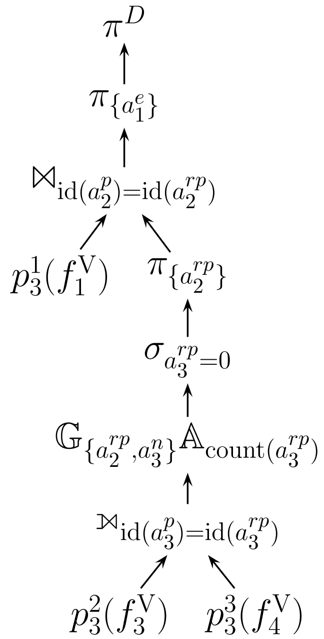

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_3^1(f_1^{\mathrm{V}})$
$\pi_{\{a_2^{rp}\}}$
$\sigma_{a_3^{rp}=0}$
$\mathbb{G}_{\{a_2^{rp},a_3^n\}}\mathbb{A}_{\mathrm{count}(a_3^{rp})}$
$⟕_{\mathrm{id}(a_3^p)=\mathrm{id}(a_3^{rp})}$
$p_3^2(f_3^{\mathrm{V}})$
$p_3^3(f_4^{\mathrm{V}})$

Figure 5.22: DEP for query q_3

rather than to the pattern node that matches the proxy nodes corresponding to the cross-fragment join. In the present example the relevant pattern node is the one with the node test `book`, which is labeled a_3^n during decomposition.

To illustrate this, consider how the correct plan for query q_3 (shown in Figure 5.22) proceeds. First, a left outer join is performed between the results of $p_3^2(f_3^{\mathrm{V}})$ and $p_3^3(f_4^{\mathrm{V}})$. This yields the following three tuples. Note that the tuples contain null values for the cases where no join partner was found.

$$[a_2^{rp} = RP_{12}^{1\to3}, a_3^n = \texttt{book}_1, a_3^p = P_{18}^{3\to4}, a_3^{rp} = \mathrm{NULL}]$$
$$[a_2^{rp} = RP_{16}^{1\to3}, a_3^n = \texttt{book}_2, a_3^p = P_{20}^{3\to4}, a_3^{rp} = \mathrm{NULL}]$$
$$[a_2^{rp} = RP_{16}^{1\to3}, a_3^n = \texttt{book}_2, a_3^p = P_{21}^{3\to4}, a_3^{rp} = RP_{21}^{3\to4}]$$

Next, this result is grouped by the attributes a_2^{rp} and a_3^n. At the same time the non-null values of a_3^{rp} are counted for each group. This results in the following two tuples.

$$[a_2^{rp} = RP_{12}^{1\to 3}, a_3^n = \texttt{book}_1, a_3^{rp} = 0]$$
$$[a_2^{rp} = RP_{16}^{1\to 3}, a_3^n = \texttt{book}_2, a_3^{rp} = 1]$$

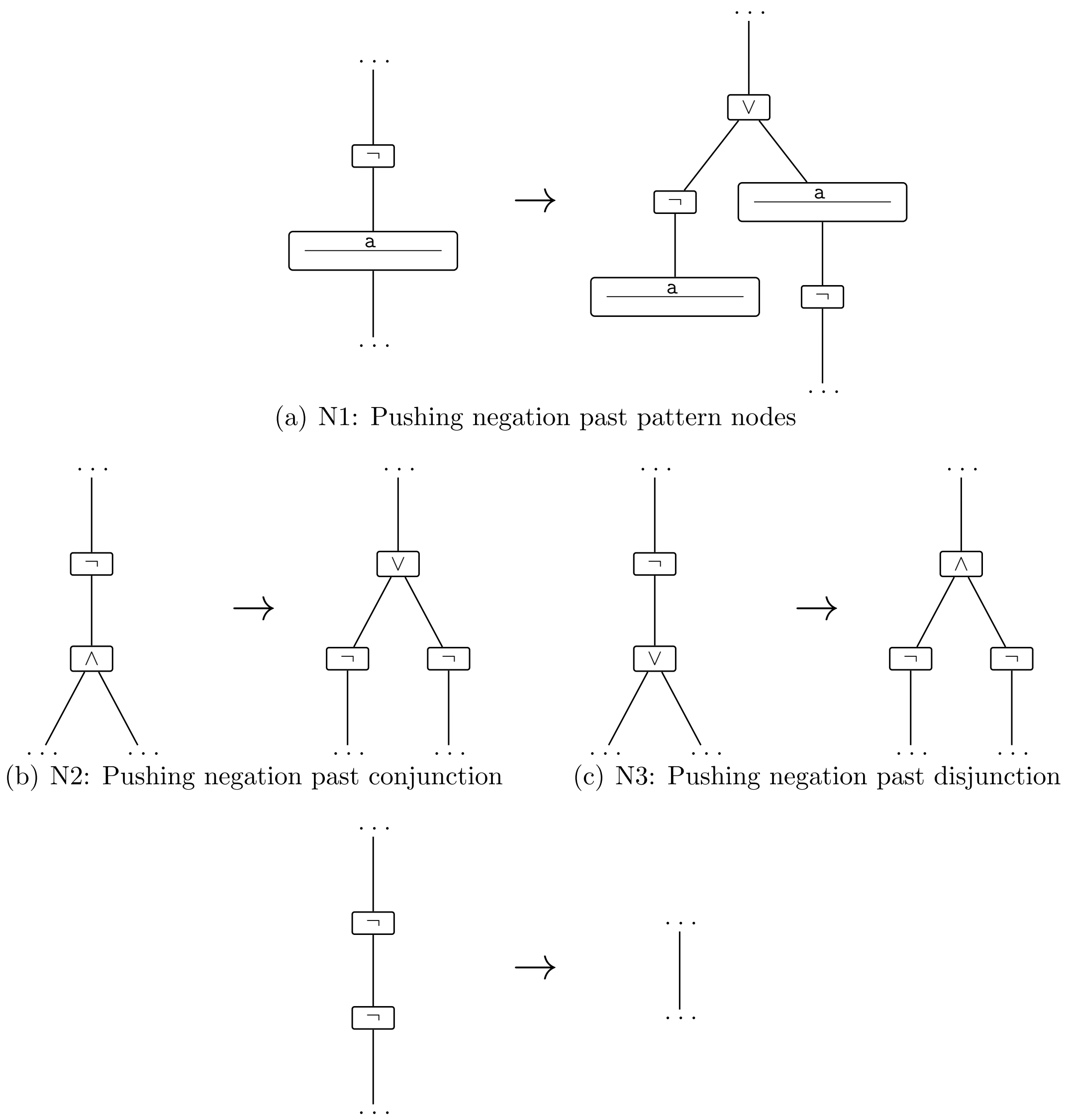
Finally, the selection returns only those tuples for which $a_3^{rp} = 0$, resulting in the tuple $[a_2^{rp} = RP_{12}^{1\to 3}]$ (after projection). This is then joined with the result of $p_3^1(f_1^{\mathrm{V}})$ leading to the correct result consisting of the single tuple $[a_1^e = \texttt{author}_1]$.

5.2.6.2 Negation Rewrites

If there is a cross-fragment edge in the pattern branch below a ¬ logic node but not directly adjacent to the logic node, then it is not possible to directly fold this negation into the cross-fragment join. In this case, the query is rewritten until either there are no cross-fragment steps below the negation (in which case no special treatment is necessary) or there is a cross-fragment step directly adjacent to the negation (in which case the negation can be folded into the cross-fragment join as shown in the previous section).

Figure 5.23 shows the rewrite rules that make this possible. Rule N1 shows how a negation can be pushed past a pattern node. When this rule is applied, the pattern node with the node test `a` is known to be annotated with the same fragment f_i^{V} as the ¬ logic node (if it was annotated with a different fragment, the negation could have been folded into the cross-fragment join without further rewrites, corresponding to case 2 above). N1 introduces a disjunction consisting of two branches. The branch shown on the left-hand side consists of a negation followed by a copy of the pattern node with the node test `a`. This branch consists solely of nodes annotated with fragments f_i^{V} and therefore needs no further rewriting (case 1 above). The right-hand side branch may need to be rewritten further until either case 1 or case 2 is satisfied.

It is important to note that, unlike the other rewrite rules presented here, N1 is not an equivalence. This is because in the right-hand branch of the rewritten pattern, the pattern node over which the negation is quantified has changed. To illustrate this, consider the XPath expression `/author/pubs[not(book/chapter)]`, which corresponds to a query



(a) N1: Pushing negation past pattern nodes

(b) N2: Pushing negation past conjunction

(c) N3: Pushing negation past disjunction

(d) N4: Merging negations

Figure 5.23: Negation rewrite rules

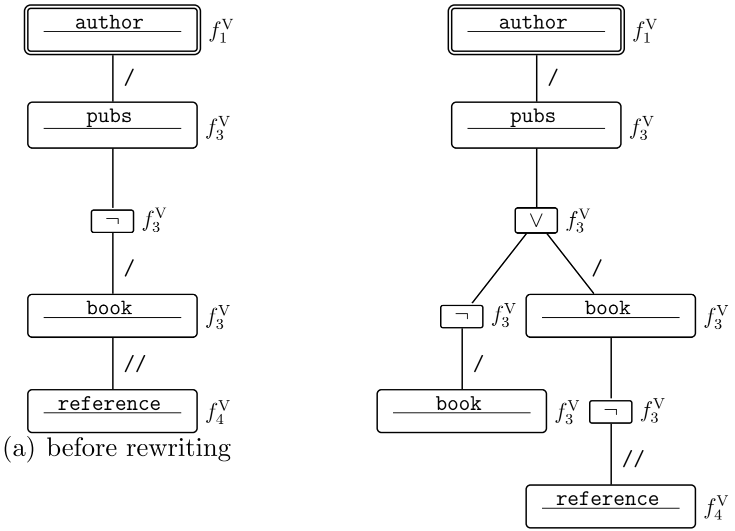
for the pubs element of all authors that do not have any book with at least one chapter. Applying rewrite N1 to this query yields a QTP that is equivalent to the expression /author/pubs[not(book) or book[not(chapter)]], which yields the pubs element of authors that either do not have a book at all or that have at least one book without a chapter (but may have other books that do have chapters).

To address this problem, and to ensure that the correct query result is returned in these cases, the grouping operator used in the DEP groups by the attribute that corresponds to the pattern node that originally preceded the negation before the rewrite was applied. Thus, in the example given in the previous paragraph, results would be grouped by the attribute that matches pubs nodes, rather than the attribute that matches book nodes.

While N1 makes it possible to push a negation past a pattern node, it may also be necessary to push a negation past another logic node. Rules N2 and N3 show how this is done for conjunction and disjunction, respectively. Note that when pushing a negation past a conjunction, the conjunction is transformed into a disjunction. Similarly, pushing a negation past a disjunction turns the disjunction into a conjunction. Both N2 and N3 follow directly from De Morgan's laws [110]. Two adjacent negations cancel each other out and, as shown in rule N4, can be removed from the pattern.

Together, rules N1–N4 make it possible to push negations past any kind of pattern or logic node. Therefore, it is always possible to rewrite a given QTP such that each negation contained in this QTP is either local and thus requires no special treatment (case 1) or can be folded into a cross-fragment join as described in Section 5.2.6.1 (case 2).

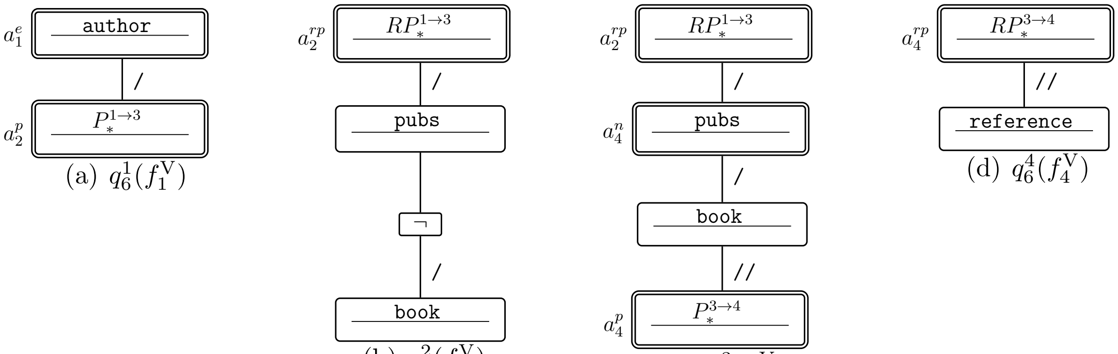
For example, consider query q_6, whose annotated QTP is shown in Figure 5.24(a). Before this query can be decomposed, the $\neg$ logic node has to be pushed past the pattern node with the node test book. This is done by applying rewrite rule N1. The result of rewriting the QTP is shown in Figure 5.24(b). Note that the rewritten QTP contains two negations. The negation in the branch on the left-hand side now satisfies case 1 and can be handled locally within a single QTP. The negation in the right-hand branch, in contrast, can be folded into the cross-fragment join. The rewritten QTP can then be decomposed into the local QTPs shown in Figure 5.25 (note that the disjunction requires that two local QTPs be generated for fragment f_3^{V}).



(a) before rewriting

(b) after rewriting

Figure 5.24: Annotated QTP corresponding to query q_6 before and after rewriting



(a) $q_6^1(f_1^{\mathrm{V}})$

(b) $q_6^2(f_3^{\mathrm{V}})$

(c) $q_6^3(f_3^{\mathrm{V}})$

(d) $q_6^4(f_4^{\mathrm{V}})$

Figure 5.25: Local QTPs corresponding to query q_6

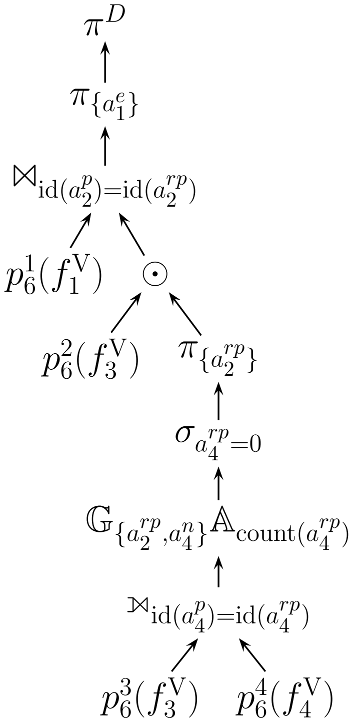

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_6^1(f_1^{\mathrm{V}})$
$\odot$
$p_6^2(f_3^{\mathrm{V}})$
$\pi_{\{a_2^{rp}\}}$
$\sigma_{a_4^{rp}=0}$
$\mathbb{G}_{\{a_2^{rp},a_4^n\}}\mathbb{A}_{\mathrm{count}(a_4^{rp})}$
$⟕_{\mathrm{id}(a_4^p)=\mathrm{id}(a_4^{rp})}$
$p_6^3(f_3^{\mathrm{V}})$
$p_6^4(f_4^{\mathrm{V}})$

Figure 5.26: DEP for query q_6

Note that when decomposing a rewritten QTP, a^n_v is assigned to the pattern node that originally preceded the negation (i.e., before the rewrites were applied). This ensures that the correct query result is obtained in cases where rewrite N1 has been applied. For query q_6, this is the pattern node with the node test `pubs` in local QTP $q^3_6(f^{\mathrm{V}}_3)$ (and not the pattern node with the node test `book`, which precedes the negation after rewriting). As shown in Figure 5.25, this pattern node is labeled a^n_4 and designated as an extraction point. This ensures that the negation is related to the correct pattern node by the grouping and aggregation. Thus, the query result is correct and consists of authors whose publications do not contain a book with a reference (rather than authors whose publications contain a book without a reference).

After decomposition, query q_6 is evaluated using the DEP shown in Figure 5.26, which is generated as described in the previous section.

5.3 Summary

This chapter has described the fundamental techniques for evaluating XQ queries over an XML collection that has been fragmented horizontally and/or vertically and then distributed across multiple sites in a system. While these techniques alone are sufficient to answer queries correctly, there is room for improving the performance of distributed query evaluation. The next chapter addresses this and introduces a suite of techniques that can be used in combination with the fundamental techniques presented here to achieve improved query performance and scalability.

Chapter 6

Techniques for Improving Distributed Execution Plans

The focus of this chapter is on techniques for improving the performance of distributed query evaluation over fragmented XML collections. Starting from the DEPs described in the Chapter 5, a suite of techniques for both horizontal and vertical fragmentation is introduced. As with all techniques presented in this thesis, a major focus is on improving parallelism, which is a key consideration when optimizing for scalable query execution in a data centre. Together, the techniques presented in this chapter help to further improve the query performance achieved by DEPs.

In relational systems, distributed query optimization techniques usually work over an algebraic representation of a distributed query [115]. For many of the techniques presented here, however, the QTP represents a simpler abstraction that contains all the necessary information.

The remainder of this chapter is organized as follows. Section 6.1 introduces the techniques that can be applied to distributed query evaluation over horizontally fragmented collections. Section 6.2 focuses on techniques for vertically fragmented collections. All the techniques presented here are designed to be fully orthogonal. Therefore, when a collection is fragmented in a hybrid fashion, it is possible to combine the horizontal and vertical techniques presented in this chapter in a single DEP.

6.1 Horizontal Fragmentation

In the case of horizontal fragmentation, the approach for improving query performance relies on pruning the set of fragments that need to be accessed to answer a given query. This can help reduce resource contention between multiple queries and thereby improve query throughput. Techniques for pruning fragments from a DEP with horizontal fragmentation are presented in Section 6.1.1.

Another improvement concerns the sorting step that is performed after results from individual fragments are combined. Section 6.1.2 discusses how the need for sorting can be eliminated by choosing the right implementation for the operator that merges local query results.

6.1.1 Pruning Fragments

As discussed in Section 5.1.2, DEPs for horizontally fragmented collections generally need to access all fragments to answer a query. However, there are some cases where this is not necessary, because some horizontal fragments cannot possibly contribute to the query result.

For example, consider query q_7, whose QTP representation is shown in Figure 6.1. This query retrieves all references in books written by an author whose first name is William and whose last name is Shakespeare. When evaluating q_7 over the horizontally fragmented collection shown in Figure 6.2, it is easy to see that only the documents in fragment f_3^{H}, corresponding to authors whose last name starts with the letter 'S', need to be considered. All the other fragments are irrelevant for the query.

This section introduces a procedure that detects irrelevant fragments and prunes them from the DEP[1]. This procedure relies on the schema of the collection and the FTPs that define the fragmentation. Both of these are static over time, do not depend on the size of the collection, and can be encoded in a compact manner. This makes it feasible to

[1] The work presented in this section has been published as a formal paper [83], and as technical reports [80, 81].

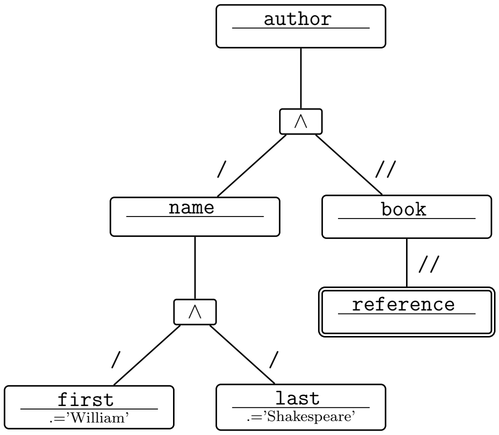
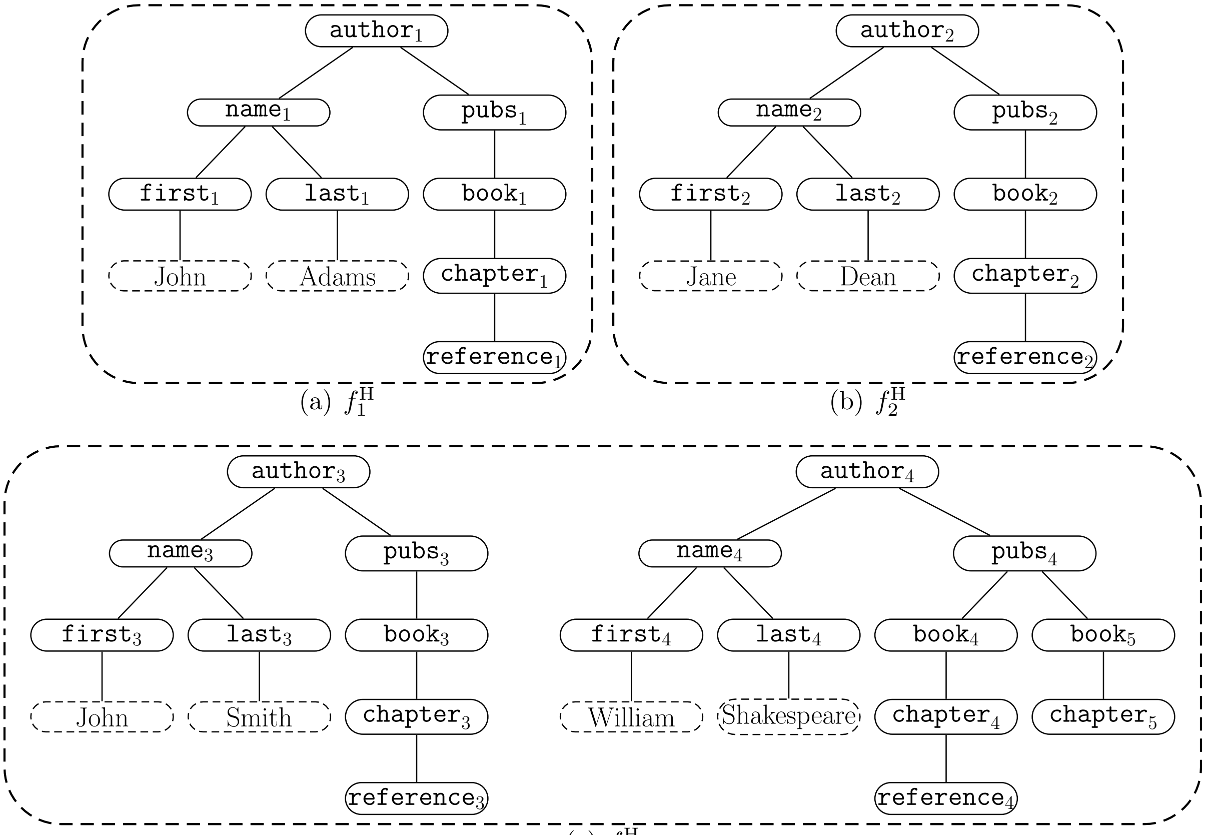


Figure 6.1: QTP representation of query q_7



(a) f_1^{H}

(b) f_2^{H}

(c) f_3^{H}

Figure 6.2: A horizontally fragmented collection

replicate them at all sites as metadata. Thus, pruning can easily be performed locally before contacting the sites holding the relevant fragments.

The pruning algorithm operates on the QTP representation of the query and is applied before the QTP is converted to an algebraic query plan. This makes it possible to reduce the problem of pruning horizontal fragments to that of determining, for each fragment, whether its FTP can be satisfied at the same time as the QTP.

To eliminate a fragment from the distributed query plan, it is necessary to show that the FTP corresponding to this fragment and the QTP representation of the query are *mutually exclusive*. This is the case exactly when there cannot exist a document (corresponding to the schema) that yields a match for both the QTP and the FTP.

Definition 6.1. Let S be the schema of the collection. Two tree patterns q_i and q_j are *mutually exclusive* iff for any document d corresponding to S, $q_i(d)$ does not yield a match or $q_j(d)$ does not yield a match. ■

While the problem of detecting mutually exclusive tree patterns could be solved by a general-purpose query intersection algorithm, this section presents a schema-aware algorithm that supports QTPs with multiple extraction points as are frequently encountered in hybrid fragmentation. For a discussion of this, see Section 3.2.3.3.

To determine whether a QTP and an FTP are mutually exclusive, the algorithm performs the following sequence of steps.

- First, both the QTP and the FTP are transformed into a simplified form. While this form is less expressive than general tree patterns, it is sufficient to detect contradictions.

- The simplified QTP and the simplified FTP are then traversed in parallel. For each pair of corresponding pattern nodes, the value constraints associated with both pattern nodes are checked for contradictions.

- If at least one contradiction is found, QTP and FTP are mutually exclusive, and the fragment corresponding to the FTP is eliminated from the DEP. On all other fragments, the original QTP (i.e., not the simplified QTP) is evaluated as in the case without pruning.

6.1.1.1 Transformation to Simplified Form

In general, pattern nodes in a QTP may match more than one node in a given document tree. Consider, for example, query q_7 (shown in Figure 6.1). When evaluating this query over the horizontally fragmented collection shown in Figure 6.2, the pattern node with the node test `book` matches two nodes in the document rooted at $\texttt{author}_4$: $\texttt{book}_4$ and $\texttt{book}_5$. In the following, pattern nodes that match more than one node in the same document are referred to as *ambiguous pattern nodes*.

Definition 6.2. Let $q = \langle N, L, r, E, \nu, c, \varepsilon, \lambda, T\rangle$ be a tree pattern and S be the schema of the collection. A pattern node $n \in N$ is an *ambiguous pattern node* iff there exists a document d conforming to the schema such that there exist two matches μ_1 and μ_2 for q in d, μ_1 assigns node $o_1 \in d$ to pattern node n, μ_2 assigns $o_2 \in d$ to pattern node n, and $o_1 \neq o_2$. ■

A value constraint associated with an ambiguous pattern node is satisfied if at least one of the matching nodes in the document conforms to the constraint. Therefore, in the general case, the presence of contradictory value constraints is not sufficient to determine that two tree patterns are mutually exclusive. This is because, even if the constraints themselves are contradictory, they may be satisfied by different nodes in the same document.

There are several features that, when present in a tree pattern, result in ambiguous pattern nodes:

Node types reached via MULT edges Node types that are reachable from the root of the schema via an edge with the cardinality MULT may occur multiple times in the same context. The schema shown in Figure 6.3, for example, allows multiple `book` nodes to occur as the child of a single `pubs` node. Thus, a pattern node with the node test `book` is ambiguous.

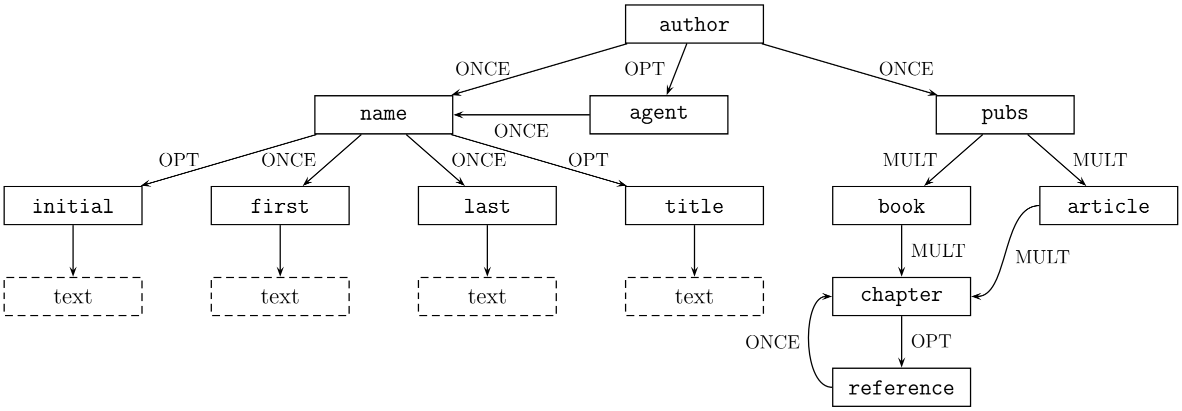
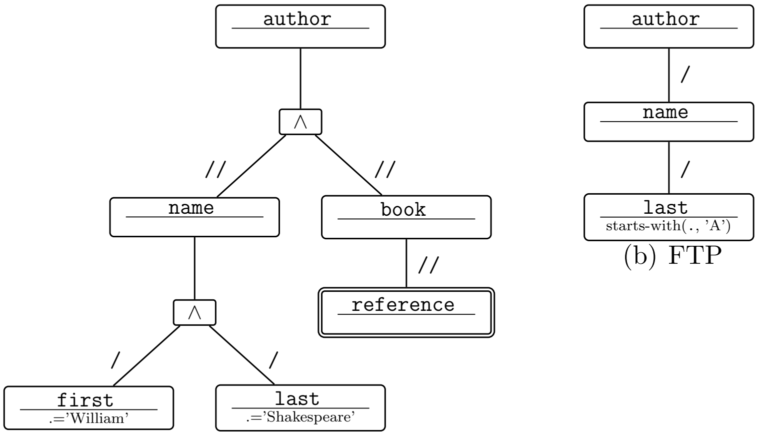


Figure 6.3: An XML schema graph

Descendant steps can also lead to ambiguous pattern nodes, since they may be satisfied by multiple paths in the schema. The QTP q_8, shown in Figure 6.4(a), for example, contains a descendant edge between the pattern node with the node test `author` and the pattern node with the node test `name`. As can be seen in the schema (shown in Figure 6.3), the pattern node with the node test `name` can be matched to a `name` node that is the direct child of an `author` node or to a `name` node that is the child of an intermediate `agent` node. Therefore, the QTP shown in Figure 6.4(a) and the FTP shown in Figure 6.4(b) are not mutually exclusive, despite the fact that the value constraints on the pattern nodes with the node test `last` are contradictory (since a last name cannot, at the same time, start with the letter 'A' and be equal to 'Shakespeare'). The documents in the fragment corresponding to the FTP in Figure 6.4(b) contain information about authors whose last names start with the letter 'A'. The QTP, on the other hand, matches books that are either authored by 'William Shakespeare' or by someone whose agent is 'William Shakespeare' and whose last name might well start with the letter 'A'. Thus, the fragment corresponding to this FTP cannot be pruned from a query plan for query q_8.

Wildcards are another source of ambiguous pattern nodes. This is because pattern nodes with a wildcard node test may be matched to document nodes of different types, thus creating ambiguity.



(a) QTP q_8

(b) FTP

Figure 6.4: QTP and FTP that are not mutually exclusive

To address the problem of ambiguous pattern nodes, both tree patterns (i.e., the QTP and the FTP) are transformed into a simplified form before they are inspected for contradicting value constraints. The simplified form is guaranteed not to contain ambiguous pattern nodes. Therefore, each pattern node in a simplified pattern matches at most one node within a given document.

In addition to ambiguous pattern nodes, the detection of mutually exclusive tree patterns is also complicated by negation. While negation that occurs within value constraints can easily be handled when inspecting value constraints for contradictions, negation logic nodes (denoted by ¬) result in pattern nodes that are not matched to any node from the collection for a given pattern match. Thus, negation logic nodes are also excluded from simplified tree patterns.

Formally, a simplified tree pattern can be defined as a tree pattern that does not contain any of the problematic primitives.

Definition 6.3. Let $S = \langle \Sigma, \Psi,\ s, m, \rho \rangle$ be the schema of the collection. Then a tree pattern $\langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$ is a *simplified tree pattern* iff $T = \emptyset$ $\forall n \in N,\ \nu(n) \in \Sigma$, $\forall l \in L,\ \lambda(l) \neq \neg$, and $\forall (x,y) \in E, \varepsilon((x,y)) = \texttt{child} \wedge (\nu(x), \nu(y)) \in \Psi \wedge s((\nu(x), \nu(y)))$ $\neq$ MULT. ■

For the detection of mutually exclusive tree patterns, it is irrelevant which pattern nodes are designated as extraction points. Thus, for simplicity, the remainder of this section will assume that in a simplified tree pattern no pattern nodes are designated as extraction points (as denoted by $T = \emptyset$ in Definition 6.3).

To convert a tree pattern into a simplified tree pattern, all instances of the disallowed primitives have to be either removed or converted into an equivalent simplified form. It is important to note that simplified tree patterns are strictly less expressive than arbitrary tree patterns. Therefore, the transformation to a simplified tree pattern changes the semantics of the tree pattern. However, this is not a problem for the pruning strategy presented here because the transformation retains the pattern nodes that are necessary to detect mutually exclusive patterns. It is important to point out that the transformation does not compromise the correctness of the query result, since subsequent query processing after pruning is performed based on the original QTP query rather than the simplified form. Nevertheless, it is important that the transformation retains as much of the information present in the original pattern as possible so that this information can be exploited for pruning.

Algorithm 3 performs the transformation of a tree pattern into a simplified tree pattern based on the following principles:

- Using schema information, descendant steps are unrolled into equivalent paths consisting entirely of child steps (procedure shown as Algorithm 4). If there is more than one path, logic nodes representing a disjunction ($\vee$) are inserted and the branch below the descendant step becomes reachable via more than one path.

- Wildcard node tests are converted to non-wildcard node tests wherever this is unambiguously possible. Otherwise, the corresponding pattern nodes are removed along with their descendants.

- Pattern nodes matching node types reachable via MULT edges in the schema are removed along with the branches below them.

- Pattern nodes designated as extraction points in the original tree pattern do not receive this designation in the simplified tree pattern.

Algorithm 3: pattern transformation algorithm

```
input  : tree pattern $\langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$, schema $\langle \Sigma, \Psi, s, m, \rho \rangle$
output : simplified tree pattern $\langle N', L', r', E', \nu', c', \varepsilon', \lambda', T' \rangle$
$r' \leftarrow$ new node
$\nu'(r') \leftarrow \nu(r)$
$c'(r') \leftarrow c(r)$
$N' \leftarrow \{r'\}$
$E' \leftarrow \emptyset$
$T' \leftarrow \emptyset$
$Q \leftarrow \{\langle r, r' \rangle\}$ // represents nodes whose children have yet to be checked
while $Q \neq \emptyset$ do
    $\langle q, q' \rangle \leftarrow$ some $\langle q, q' \rangle \in Q$ // while there are pattern or logic nodes to be processed, pick one
    $Q \leftarrow Q \setminus \{\langle q, q' \rangle\}$
    for $y \in$ children$(x)$ do
        $y' \leftarrow$ new node
        if $y \in L, \lambda(y) \neq \neg$ then
            $L' \leftarrow L' \cup \{y'\}$, $E' \leftarrow E' \cup \{\langle q', y' \rangle\}$, $\lambda'(y') \leftarrow \lambda(y)$
            $Q \leftarrow Q \cup \{\langle y, y' \rangle\}$
        else if $y \in N$ then
            if $q' \in L'$ then
                $q'_p \leftarrow$ nearest pattern node ancestor of $q'$
            $\sigma' \leftarrow \nu(q'_p)$
            $c'(y') \leftarrow c(y)$
            if $\varepsilon(e) = /$ then
                if $\nu(y) \neq *$ then
                    $\nu'(y') = \nu(y)$
                else if $\exists \langle \sigma_1, \sigma_2 \rangle \in \Psi$ unique with $\sigma_1 = \sigma'$ then
                    $\nu'(y') \leftarrow \sigma_2$
                else
                    continue
                if $\psi = \langle \nu(x), \nu(y) \rangle \in \Psi, s(\psi) \neq$ MULT then
                    $N' \leftarrow N' \cup \{y'\}$, $E' \leftarrow E' \cup \{\langle q', y' \rangle\}$
                    $Q \leftarrow Q \cup \{\langle y, y' \rangle\}$
            else if $\nu(y) \neq *$ then
                $\Sigma' \leftarrow \{\sigma \in \Sigma \mid \sigma$ reachable from $\sigma', \nu(y)$ reachable from $\sigma$ in $\langle \Sigma, \Psi \rangle\}$
                $\Psi' \leftarrow \{\langle \sigma_1, \sigma_2 \rangle \in \Psi \mid \sigma_1, \sigma_2 \in \Sigma'\}$
                if $\langle \Sigma', \Psi' \rangle$ is acyclic and $\nexists \psi \in \Psi'$ with $s(\psi) =$ MULT then
                    $\nu'(y') \leftarrow \nu(y)$
                    $\langle N'', L'', E'' \rangle \leftarrow$ unrolldesc$(q', \sigma', y', \Sigma', \Psi')$
                    $N' \leftarrow N' \cup N'' \cup \{y'\}$, $L' \leftarrow L' \cup L''$, $E' \leftarrow E' \cup E''$
                    $Q \leftarrow Q \cup \{\langle y, y' \rangle\}$
$\forall \langle x', y' \rangle' \in E', y' \in N' : \varepsilon'(\langle x', y' \rangle) \leftarrow /$
return$\langle N', L', r', E', \nu', c', \varepsilon', \lambda', T' \rangle$
```

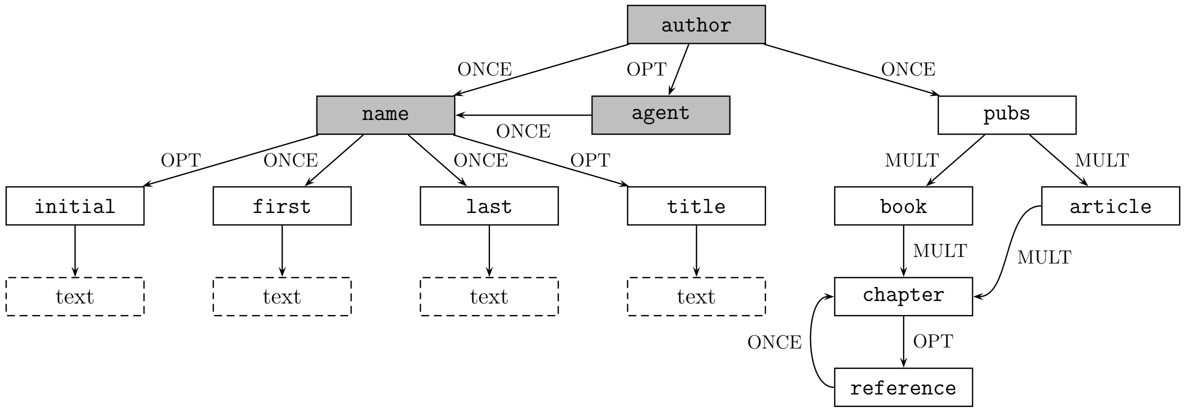


Figure 6.5: Node types reachable from `author` from which `name` is reachable

6.1.1.2 Unrolling Descendant Steps

The unrolling of descendant steps can be succinctly implemented as a manipulation of the directed graph representation of the schema (Algorithm 3, lines 32–38). To unroll a descendant step from a pattern node with the node test `a` to a pattern node with the node test `b`, a subgraph of the schema graph is considered. This subgraph consists of all node types that are both reachable from node type `a` and from which node type `b` is reachable. Thus the subgraph contains all node types that are encountered on any downward path from `a` to `b`. For an example, consider Figure 6.5, which shows the schema from Figure 6.3 with the subgraph used to unroll the descendant step `author//name` highlighted. As can be seen, this subgraph consists of the node types `author`, `agent`, and `name`.

If the schema subgraph contains a cycle, it is not possible to unroll the descendant step into a finite number of child steps. Therefore, the descendant step is discarded along with all nodes that occur below it in the tree pattern (Algorithm 3, line 34). Conceptually, this scenario corresponds to a pattern node that may match nodes that occur at different levels in the document tree. This causes ambiguity, rendering the affected branch unusable for detecting mutual exclusion.

For an example of this scenario, consider the step `book//reference`. As is highlighted in Figure 6.6, there is a cycle involving the node types `chapter` and `reference`. Thus, assuming that the pattern node with the node test `book` is matched to a given `book` node

Algorithm 4: unrolldesc($\sigma_o, \sigma_t, \Sigma', \Psi'$) unrolls descendant step

```
input  : origin node $q_o$, origin node type $\sigma_o$, target node $q_t$, reachable schema nodes $\Sigma'$, reachable schema edges $\Psi'$
output : pattern nodes $N''$, logic nodes $L''$, pattern edges $E''$
$N'' \leftarrow \emptyset$
$L'' \leftarrow \emptyset$
$E'' \leftarrow \emptyset$
$S \leftarrow \{q_o\}$ // pattern nodes to be processed
for $s \in S$ do
    if $s = q_0$ then
        $\sigma_s \leftarrow \sigma_o$
    else
        $\sigma_s \leftarrow \nu(s)$
    if $\exists \langle \sigma_1, \sigma_2 \rangle, \langle \sigma_3, \sigma_4 \rangle \in \Psi', \sigma_2 \neq \sigma_4, \sigma_s = \sigma_1 = \sigma_3$ then
        // more than one outgoing edge from $s$, insert disjunction
        $l_\vee \leftarrow$ new node
        $\lambda'(l_\vee) \leftarrow \vee$
        $L'' \leftarrow L'' \cup \{l_\vee\}$
        $E'' \leftarrow E'' \cup \{\langle s, l_\vee \rangle\}$
        $s \leftarrow l_\vee$
    // insert edges
    for $\langle \sigma_1, \sigma_2 \rangle \in \Psi', \sigma_1 = \sigma_s$ do
        if $\sigma_2 = \nu(q_t)$ then
            $n_{\sigma_2} \leftarrow q_t$
        else
            $n_{\sigma_2} \leftarrow$ new node
            $\nu'(n_{\sigma_2}) \leftarrow \sigma_2$
            $c'(n_{\sigma_2}) \leftarrow \bot$
            $N'' \leftarrow N'' \cup \{n_{\sigma_2}\}$
            $S \leftarrow S \cup \{n_{\sigma_2}\}$
        $E'' \leftarrow E'' \cup \{\langle n_\sigma, n_{\sigma_2} \rangle\}$
return$\langle N'', L'', E'' \rangle$
```

in the collection, the pattern node with the node test `reference` may be matched to a `reference` node that is reachable from this `book` node via a single, intermediate `chapter` node or to a `reference` node reachable via a chain of multiple `chapter` and `reference` nodes.

If the subgraph is acyclic (as in the example shown in Figure 6.5), the unrolling algorithm introduces a new pattern node for each of the intermediate schema nodes such that the node test of the pattern node matches the name of the corresponding schema node

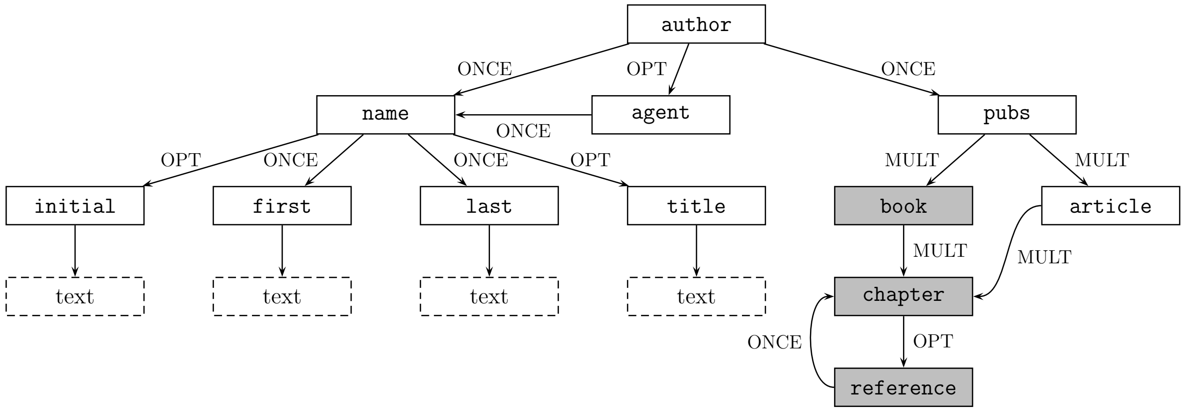


Figure 6.6: Node types reachable from `book` from which `reference` is reachable

(Algorithm 4, lines 21–26). In cases where a schema node has more than one child, a logic node representing a disjunction is inserted (Algorithm 4, lines 11–16). This signifies that there are multiple paths through the schema that the descendant step could correspond to. In order for the pattern to be satisfied for a given document, only one of these paths needs to be matched to the document.

Figure 6.7 shows the QTP representation of query q_8 after unrolling descendant steps. For this QTP, all of the scenarios discussed above are encountered. The step `author//book` is unrolled into a linear sequence of child steps by adding an intermediate pattern node with the node test `pubs`. Unrolling `author//name`, on the other hand, requires the insertion of a disjunction. This is because, as is shown in Figure 6.5, there are two paths from the node type `author` to the node type `name`. Finally, the step `book//reference` is discarded altogether due to the cycle in the schema involving these node types.

Unrolling descendant steps, as formally described in Algorithms 3 and 4 transforms the pattern from a tree into a directed acyclic graph (DAG). This is because, rather than duplicating pattern nodes reachable via more than one path in the schema, multiple pattern edges leading to these pattern nodes are inserted. It is straightforward to convert these DAGs back into a tree representation (as was done for the example shown in Figure 6.7). However, for performance reasons it may be preferable to traverse the more compact DAG representation directly when checking for mutually exclusive patterns, which leads to the same result as traversing the tree representation.

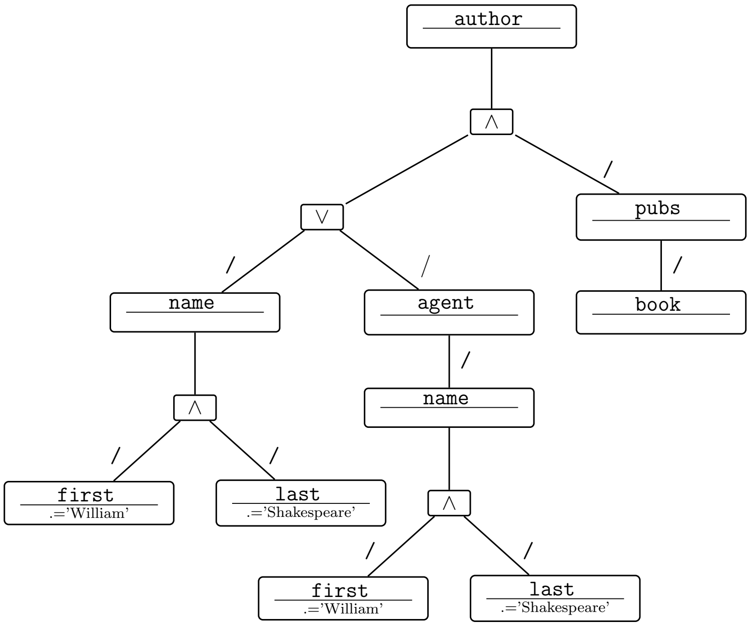
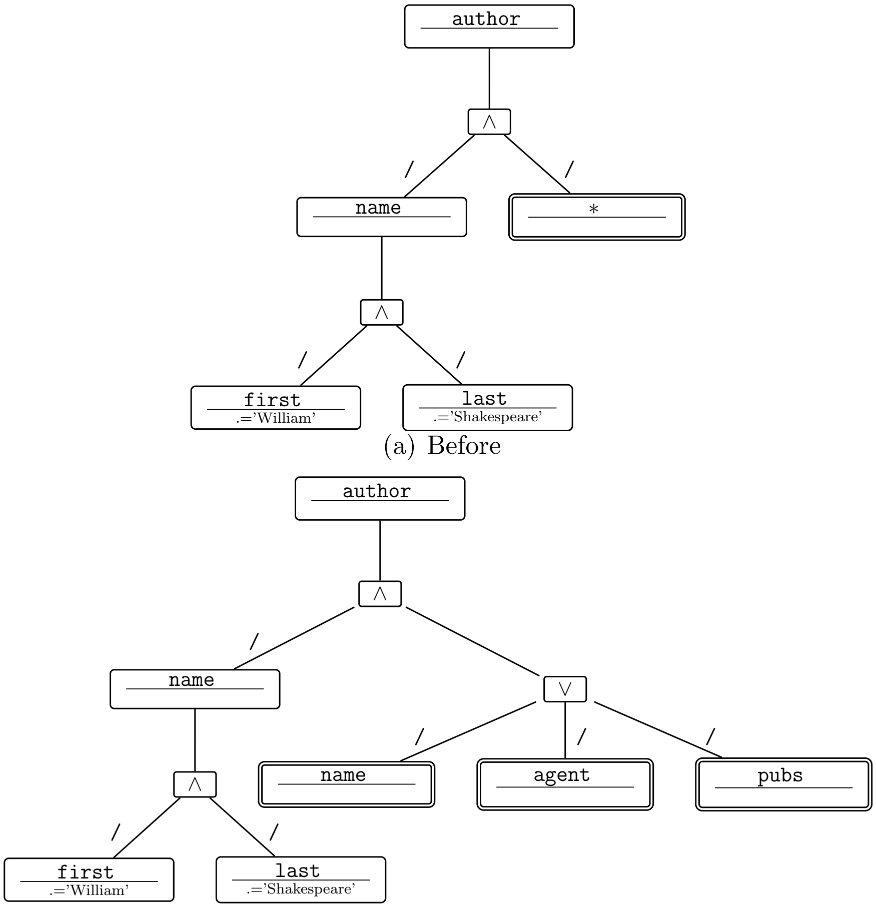


Figure 6.7: QTP representation of query q_8 after unrolling descendant steps

6.1.1.3 Removing Wildcard Nodes

Algorithm 3 converts wildcard node tests in pattern nodes to explicit node tests whenever this is unambiguously possible (lines 24 and 25). For a child step of the form `a/*`, this is the case if the node type `a` only has a single outgoing edge in the schema. Based on the schema shown in Figure 6.3, for example, the step `agent/*` can be converted to `agent/name`, since the node type `agent` has a single outgoing edge leading to the node type `name`. Similarly, a descendant step of the form `a//*` can be converted if there is only one node type in the schema that is reachable from node type `a`.

In cases where this simple transformation cannot be applied (because the wildcard node test may match nodes of more than one type), a disjunction node can be inserted into the simplified pattern, with each branch of the disjunction consisting of a pattern node with a node test that explicitly matches one of the node types to which the wildcard may refer. This is done using a procedure that resembles the strategy for annotating wildcard nodes in QTPs (as described in Section 5.2.1). In the case of a child step of the form `a/*`, unrolling



(a) Before

(b) After

Figure 6.8: Unrolling wildcard node test in query q_4

the wildcard results in one pattern node for each node type `b` such that there is a direct edge from `a` to `b` in the schema. For a descendant step `a//*`, on the other hand, a pattern node has to be inserted for each node type `b` that is reachable from `a` in the schema.

Figure 6.8 shows how the wildcard node in query q_4 can be unrolled. As can be seen, three pattern nodes are inserted, one for each node type that the wildcard node could match.

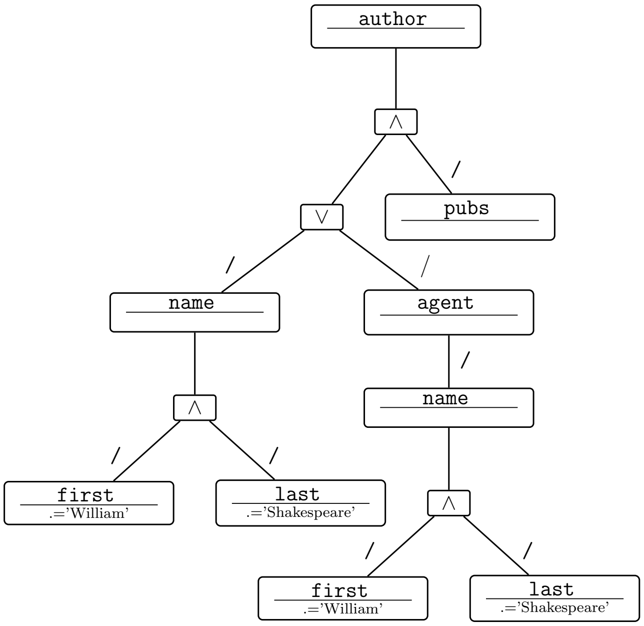


Figure 6.9: QTP representation of query q_8 after removing pattern nodes with multiple matches

6.1.1.4 Removing Pattern Nodes With Matches Reached via MULT Edges

In general, it is not possible to perform a meaningful conversion of pattern nodes that match collection nodes with multiple occurrences in the same context (i.e., node reached via MULT edges in the schema). Thus, these nodes are eliminated from the pattern when the pattern is converted to the simplified form.

In the example from Figure 6.7, the `book` node needs to be removed since the schema indicates that a `pubs` node may have multiple children of type `book`. The resulting simplified pattern is shown in Figure 6.9.

6.1.1.5 Removing Negation Logic Nodes

While in certain cases it may be possible to fold the negation in a logic node into the value constraint, in general there is no practical way of converting these nodes while retaining the semantics of the query. Therefore, the solution presented here deals with negation logic nodes in essentially the same way as with node tests that match nodes reachable via MULT edges. The negation logic nodes are simply removed from the tree pattern along with the pattern branch below them (Algorithm 3, line 13).

6.1.1.6 Traversal and Pruning

After both the QTP and the FTP have been transformed into the simplified form, both patterns are traversed simultaneously as described in Algorithm 5. During this traversal, a pattern node in one pattern is only visited if there is a corresponding pattern node in the other pattern that occurs in the same position of the pattern and that has the same node test. Note that when dealing with tree patterns, position refers solely to the path from the root of the pattern through which a pattern node is reached. Since tree patterns are un-ordered trees, the relative order of siblings as depicted in the examples shown here is immaterial.

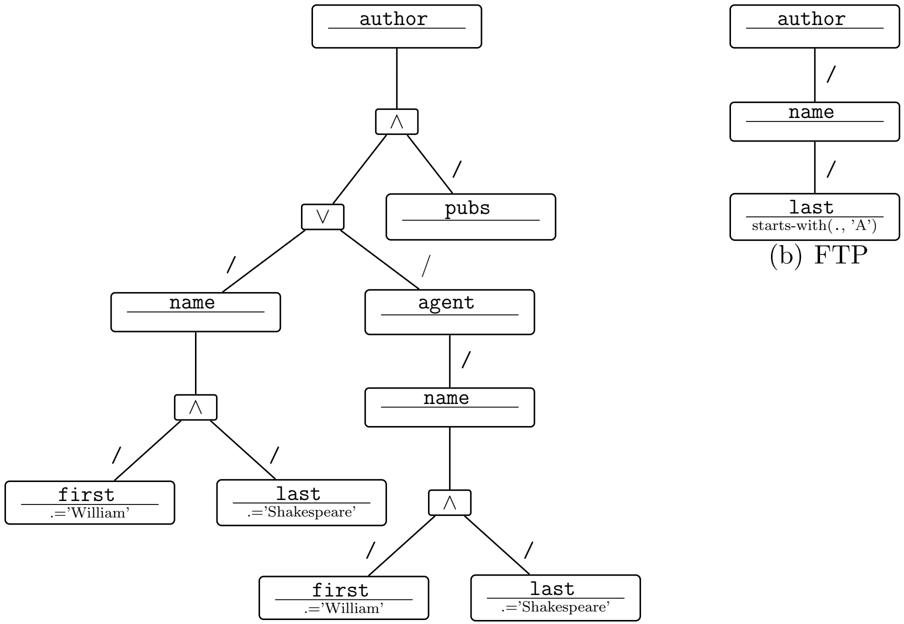
For each pair of corresponding pattern nodes, the value constraints in both patterns are inspected to determine whether they are contradictory (Algorithm 5, line 30). After simplification, each pattern node in a tree pattern corresponds to a unique collection node within the context of a single document tree. Therefore, a contradiction between the value constraints of a pair of corresponding pattern nodes immediately renders the tree patterns mutually exclusive. Thus, as soon as a contradiction is found, the traversal can be terminated and the fragment corresponding to the FTP can be excluded from the distributed query plan.

Special care has to be taken when a logic node is encountered. In the case of a conjunction, all branches have to be inspected and a contradiction even for a single branch means that the patterns are mutually exclusive. In the case of a disjunction, on the other hand, the patterns are mutually exclusive only if there is a contradiction for each branch of the disjunction. If there is at least one branch without a contradiction, which may be a branch that is not present in the other pattern, then it is not possible to conclude that the tree patterns are mutually exclusive (Algorithm 5, lines 3–7, 15–19).

In the example shown in Figure 6.10, the traversal algorithm proceeds as follows. First,

Algorithm 5: traverse(q, q') *detects whether patterns are compatible*

```
constant: FTP $\langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$ , QTP $\langle N', L', r', E', \nu', c', \varepsilon', \lambda', T' \rangle'$
input   : root of FTP sub-pattern $q$, root of QTP sub-pattern $q'$
output  : true if patterns are compatible, false otherwise
if $q \in L$ then
    // logic node in predicate pattern
    if $\lambda(q) = \vee$ then
        // check if at least one branch of disjunction is free of contradictions
        $result \leftarrow$ false
        for $x \in$ children($q$) do
            $result \leftarrow result \vee$ traverse($x, q'$)
    else
        // check if all branches of conjunction are free of contradictions
        $result \leftarrow$ true
        for $x \in$ children($q$) do
            $result \leftarrow result \wedge$ traverse($x, q'$)
else if $q' \in L'$ then
    // logic node in query pattern
    if $\lambda'(q') = \vee$ then
        // check if at least one branch of disjunction is free of contradictions
        $result \leftarrow$ false
        for $x' \in$ children($q'$) do
            $result \leftarrow result \vee$ traverse($q, x'$)
    else
        // check if all branches of conjunction are free of contradictions
        $result \leftarrow$ true
        for $x' \in$ children($q'$) do
            $result \leftarrow result \wedge$ traverse($q, x'$)
else
    // pattern node in both patterns
    if $\nu(q) \neq \nu'(q')$ then
        // pattern nodes do not correspond, no contradiction
        $result \leftarrow$ true
    else if $c(q) \wedge c'(q')$ is not satisfiable then
        // contradiction in corresponding pattern nodes $result \leftarrow$ false
    else if $\exists$child($q$) $\in (N \cup L)$, child($q'$) $\in (N' \cup L')$ then
        // no contradiction, need to check children $result \leftarrow$ traverse(child($q$), child($q'$))
    else
        // no more children on at least one side
        $result \leftarrow$ true
return $result$
```

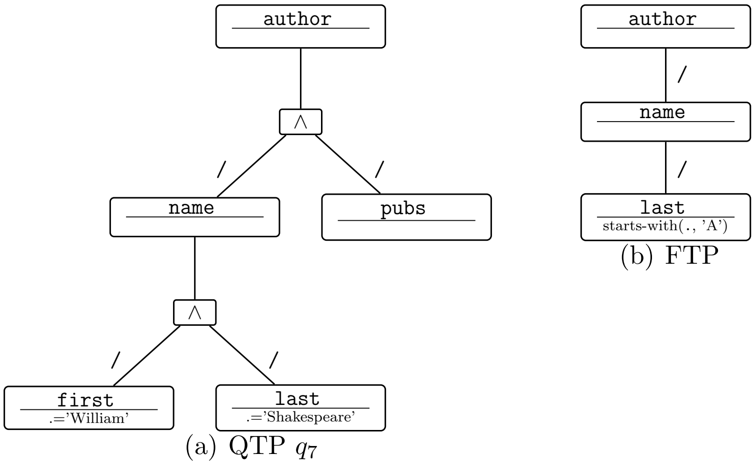


(a) QTP q_8

(b) FTP

Figure 6.10: Simplified QTP and FTP that are not mutually exclusive

the `author` nodes in the QTP and the FTP are visited. Since there is no value constraint associated with this node in either pattern, there is no conflict, therefore the algorithm moves on to the conjunction and its children. The `pubs` node is only present in the QTP and is therefore not visited. As the other child of the conjunction node, the QTP contains a disjunction. Now both branches have to be checked for contradictions. The left branch leads to the `name` node, for which there is an equivalent node in the FTP. In both patterns the `name` node has a child with node test `last`. When inspecting the value constraints associated with the `last` nodes, the algorithm detects a contradiction because the content of the corresponding document node cannot be equal to the string ‘Shakespeare’ and at the same time start with the letter ‘A’. Therefore, the algorithm determines that there is a contradiction for the left branch of the disjunction node. In order for there to be a global contradiction, however, the patterns have to be contradictory for both branches of the disjunction node. Therefore, the algorithm still has to inspect the right branch, for which it encounters a node with the node test `agent`. For this node, there is no equivalent in the FTP and therefore no contradiction. Since the algorithm only found a contradiction



(a) QTP q_7

(b) FTP

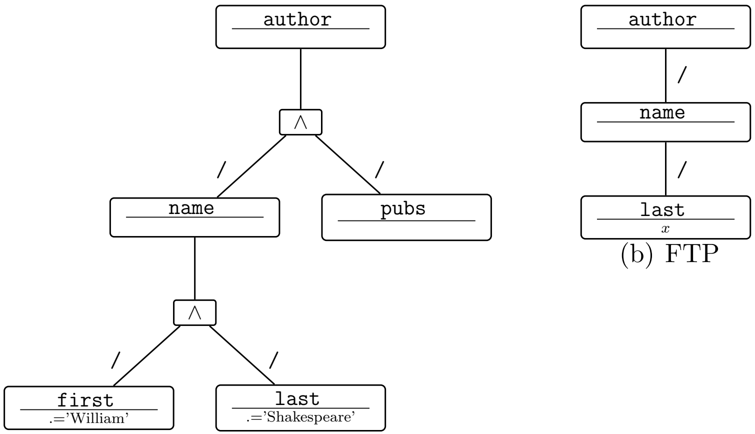
Figure 6.11: Simplified QTP and FTP that are mutually exclusive

for one branch of the disjunction node, there is no global contradiction and the fragment corresponding to the FTP cannot be pruned.

For the example in Figure 6.11, on the other hand, the traversal algorithm does detect a contradiction. After inspecting the `author` and `name` nodes in both patterns, the algorithm reaches the `last` nodes and their contradicting value constraints. This time, the `last` node does not occur as the descendant of a disjunction so this contradiction is sufficient to prune the fragment corresponding to the FTP.

6.1.1.7 Efficient Implementation

Since horizontal fragmentation is defined as a partitioning of the data collection, FTPs need to be disjoint and cover the entire collection. Because of this, it is reasonable to expect that in many cases the FTPs will only differ in their value constraints but not in their structure. It is therefore possible to simplify the traversal process by traversing the QTP together with a single abstract FTP rather than with each FTP in the fragmentation. In this abstract FTP, value constraints are replaced with variables. Traversal of QTP and abstract FTP results in an expression that describes the conditions under which the QTP and FTP are mutually exclusive. Figure 6.12(b) shows an abstract FTP, in which a value constraint has been replaced with the variable x. Traversing this abstract FTP with the



(b) FTP

(a) QTP q_7

Figure 6.12: Simplified QTP and abstract FTP

QTP in Figure 6.12(a) shows that there is a contradiction if $\neg(.=\text{'Shakespeare'} \wedge x)$ holds.

The variable x can now be instantiated with the corresponding value constraint from each of the original simplified FTPs, i.e., with the expressions

$$\text{starts-with}(., \text{'A'}),\ \ldots, \text{starts-with}(., \text{'S'}),\ \ldots, \text{starts-with}(., \text{'Z'})$$

Solving this formula yields a contradiction for all of these cases except $x = \text{startswith('S')}$. A similar optimization is possible for QTPs, assuming that the structure of a query is known at compile time whereas the constants used in value constraints are only known at query run time. With this optimization, the traversal can be performed at query compile time, and at query run time, only the formula needs to be solved.

6.1.1.8 Analysis

While it may seem that the transformation and traversal of QTP and FTPs could pose a significant overhead, there are several considerations that reduce this impact. The transformation of the FTPs only has to be performed once when the fragmentation is set up. Therefore, it does not pose a run-time overhead during query execution.

For the transformation of the QTP, the following can be observed: child steps are either copied from the QTP to the simplified QTP or omitted. Both the size of the simplified QTP and the time consumed by the transformation are therefore linear in $|E^{\mathrm{QTP}}_{\mathtt{child}}|$, which is the number of child steps in the QTP. For each descendant step, in the worst case, Algorithm 4 introduces one disjunction node and one pattern node for each σ in Σ. Therefore, the size of the simplified QTP is linear in $|E^{\mathrm{QTP}}_{\mathtt{desc}}|\ |\Sigma|$. In order to analyze the time complexity, one has to take into account the time consumed to compute the reachable schema subgraph and to inspect it for cycles. The subgraph consisting of nodes that are reachable from node a and from which b is reachable can be computed by first marking all nodes reachable from a, then marking all nodes from which b is reachable and finally choosing all nodes that were marked both times. Assuming a suitable representation of the graph, this can be done in $O(|\Sigma| + |\Psi|)$ time. Using Tarjan's algorithm [130], cycles can be detected in $O(|\Sigma| + |\Psi|)$ time. Therefore, the transformation of a QTP takes $O(|E^{\mathrm{QTP}}_{\mathtt{child}}| + |E^{\mathrm{QTP}}_{\mathtt{desc}}|\ (|\Sigma| + |\Psi|))$ time and yields a result containing $O(|E^{\mathrm{QTP}}_{\mathtt{child}}| + |E^{\mathrm{QTP}}_{\mathtt{desc}}|\ |\Sigma|)$ nodes. Since the result is also a directed graph, in which nodes may be shared among multiple branches, the equivalent tree pattern is of size $O(|E^{\mathrm{QTP}}_{\mathtt{desc}}|\ |\Sigma|\ |E^{\mathrm{QTP}}_{\mathtt{child}}| + |E^{\mathrm{QTP}}_{\mathtt{desc}}|^2\ |\Sigma|^2)$. This is important, because the time consumed by the subsequent traversal step depends on the size of the equivalent tree.

The time required to traverse the QTP and the FTPs is linear in the size of the tree representations of the simplified QTP and the FTPs. Because the traversal has to be performed for each fragment, it is also linear in the number of fragments. This leads to an overall time complexity of $O((|E^{\mathrm{QTP}}_{\mathtt{desc}}|\ |\Sigma|\ |E^{\mathrm{QTP}}_{\mathtt{child}}| + |E^{\mathrm{QTP}}_{\mathtt{desc}}|^2\ |\Sigma|^2)\ (|E^{\mathrm{FTP}}_{\mathtt{desc}}|\ |\Sigma||E^{\mathrm{FTP}}_{\mathtt{child}}| + |E^{\mathrm{FTP}}_{\mathtt{desc}}|^2\ |\Sigma|^2)\ |F|)$. Note that the run time of the pruning algorithm depends solely on the size of the patterns, the number of fragments and the size of the schema. It is independent of the size of the collection.

6.1.2 Avoiding Sorting

As discussed in Section 5.1.2, DEPs for horizontal fragmentation evaluate the original query over each relevant fragment. Each fragment yields a sequence of pattern matches, which are then combined to obtain the overall query result. This can be done by concatenating the sequences of results or by interleaving the pattern matches received from the individual sites and returning them as soon as they come in.

In general, it is necessary to sort the results after they have been combined and before they can be returned. This is because, following the semantics of XPath, XQ query results must be returned in document order. Within the context of a single document, document order corresponds to a pre-order traversal of the document tree. When dealing with multiple-document collections, the relative ordering of documents is not specified, however, the XQuery data model requires that this order be stable [48]. Thus, when answering a query over a collection consisting of two documents, d_1 and d_2, the following requirements must be satisfied.

1. **Results returned in document order** If a query q yields two matches μ_1 and μ_2 for document d_1 and the extraction point node in μ_1 occurs before the extraction point node in μ_2 in a pre-order traversal of d_1, then μ_1 must be returned before μ_2. Note that this assumes a single extraction point as specified in XQ rather than the multiple extraction points encountered in local QTPs in the context of vertical fragmentation.

2. **No interleaving of results from multiple documents** If a query q yields a set of matches M_1 for document d_1 and another set of matches M_2 for document d_2, then all of the matches in M_1 have to be returned before any of the matches in M_2 can be returned (or vice versa).

3. **Stable document order** If for a query q_1, results derived from document d_1 are returned before results derived from d_2, then the results for another query q_2 must also be returned in this order.

The local query evaluation strategies discussed in Section 3.2.2 ensure that these requirements are met for the results derived from a single fragment. However, care needs to be taken when the results from multiple fragments are combined. In the DEP shown in Figure 6.13, the sorting operator ensures that all three requirements are met regardless of how the sequences of results derived from the individual fragments are merged by the $\odot$

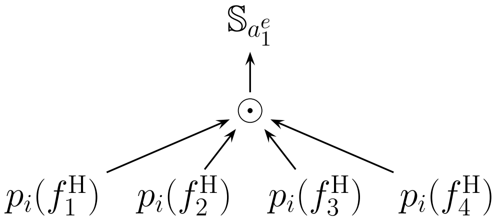


Figure 6.13: DEP with sorting

operator. However, sorting not only presents a significant overhead at the site where the results are combined, it also makes it necessary to buffer the query result until all local results have been received; only then can the results be sorted and returned.

To address this problem, this section discusses four possible implementations for the merge operator ($\odot$) and their impact on the requirements listed above, when used in a DEP without sorting (such as the one shown in Figure 6.14).

Full interleaving ($\odot^{\mathrm{FI}}$) This is a simple implementation of the $\odot$ operator that yields high performance. With full interleaving, a pattern match received from one of the fragments is returned immediately. This results in an overall query result in which the pattern matches from multiple fragments are interleaved.

When considering the impact of this interleaving on the requirements listed above, the following observations can be made. Requirement 1 is concerned with the relative ordering of matches derived from a single document. Horizontal fragmentation, as defined in Section 4.1, places each document, in its entirety, into exactly one fragment. Therefore, since the strategy of full interleaving does not re-order the results received from a single fragment, matches from a single document are not reordered and requirement 1 is met. Requirement 2, on the other hand, is violated since the matches derived from two documents in two different fragments may be interleaved. For the same reason, requirement 3 is also violated.

Document-wise interleaving ($\odot^{\mathrm{DI}}$) This is an improvement of the full interleaving strategy. Instead of returning each match as soon as it is received, matches are accumulated within the $\odot$ operator until all matches for a given document have been

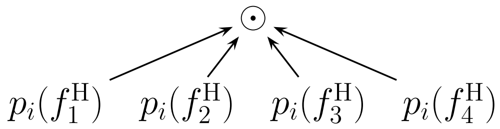


Figure 6.14: DEP without sorting

received. Once the last match for a document has been received, all matches for that document are returned.

With this strategy, all matches for a given document are returned contiguously. Therefore, requirement 2 is met. Other than that, the results received from the fragments can still be interleaved arbitrarily. Therefore, there is no stable order among documents in different fragments, and requirement 3 is not met. At the same time, this interleaving limits the amount of buffering that needs to be done. At most, the $\odot$ operator needs to buffer all the matches for one document in each fragment.

Concatenation ($\odot^{\mathrm{C}}$) With this strategy, the matches received from a given fragment are buffered until all of the matches from that fragment have been received, at which point they are returned contiguously. While this increases the amount of buffering that needs to be done, it ensures that the matches received from a given fragment are returned contiguously and therefore the matches for a given document are also returned contiguously (since, within a horizontal fragmentation, each document is stored in exactly one fragment). However, the order of fragments is not guaranteed to be stable across multiple queries. Thus, requirement 3 is not met.

Stable concatenation ($\odot^{\mathrm{SC}}$) This strategy adds another requirement to the strategy of concatenation. As before, matches from a fragment are buffered until all matches from that fragment have been received. However, in contrast to the previous strategy, stable concatenation enforces a stable order among fragments. Thus, the matches received from fragment f_j^{H} are only returned once the matches for all fragments f_i^{H}, $i < j$ have been returned. This ensures that the order among all documents in the collection is stable. Therefore, stable concatenation meets all three requirements set out above. However, this comes at the price of more buffering in the $\odot$ operator. In the worst case, all results will have to be buffered, in which case stable concatenation has the same buffering behaviour as a solution with sorting. Even in this case, however, using stable concatenation has the advantage that it eliminates the computational overhead of sorting when applied within the context of a horizontal fragmentation.

Table 6.1 shows an overview of the strategies presented here. In general, for full conformance to the requirements set out above, stable concatenation is the best choice. For many use cases, foregoing a stable order among documents may be a reasonable trade-off. In these cases, it is possible to relax requirement 3, and use document-wise interleaving as a higher-performance alternative to stable concatenation. For unordered queries, or if one is willing to relax the ordering constraint further, buffering can be avoided altogether by using full interleaving.

Technique	Requirements			Buffering
	1	2	3	
Full interleaving	✓	×	×	none
Document-wise interleaving	✓	✓	×	low
Concatenation	✓	✓	×	moderate
Stable concatenation	✓	✓	✓	high
Any + sorting	✓	✓	✓	full

Table 6.1: Comparison of strategies for combining results from horizontal fragments

It is important to point out that, when a horizontal fragmentation step occurs nested within a vertical fragmentation, the assumption that all nodes of a document are stored within a single horizontal fragment no longer holds. Thus, in this case, sorting of merged matches may be necessary regardless of which merge operator is chosen.

6.2 Vertical Fragmentation

This section introduces a set of techniques for improving the performance of distributed query evaluation over vertically fragmented collections. Since DEPs for vertical fragmentation differ significantly from DEPs seen with horizontal fragmentation, the vertical techniques are fundamentally different from the techniques seen in the horizontal scenario.

First, a pruning technique is presented. The goal of this technique is similar to that of pruning in the horizontal case: to eliminate irrelevant fragments from a DEP. Even without further optimization, DEPs for vertical fragmentation avoid accessing fragments that are not reached by the query. Section 6.2.1 presents two pruning techniques that go beyond this and prune certain fragments that are only needed to answer structural constraints. Pruning the LQPs corresponding to these fragments is complicated by the joins in the DEP. Thus, when an LQP is eliminated from the DEP, these joins have to be adjusted in order to preserve the correctness of the result. For the pruning techniques presented here, this is achieved by annotating the proxy and root proxy nodes with additional information and exploiting this information when joining local query results.

After pruning, the performance of query evaluation over vertically fragmented collections can be improved further by pushing the cross-fragment joins from the DEP into the individual LQPs and thereby skipping a portion of the sub-trees contained within a fragment. A similar optimization can be performed by applying a selection that filters the sub-trees in a fragment based on their structural relationship with nodes in other fragments. Both of these optimization techniques are presented in Section 6.2.2. Special attention is paid to maximizing pipelining in distributed query execution, since this allows for maximum parallelism.

DEPs for vertical fragmentation consist of complex join trees. Therefore, another important aspect of optimizing their performance is the order in which these joins are performed. This problem is discussed in Section 6.2.3.

As described in Section 5.2.5, before queries with disjunction can be evaluated over a vertically fragmented collection, certain sub-queries may have to be split in order to eliminate the disjunction. In some cases, this can result in local sub-queries that share large, common portions. Evaluating these common portions repeatedly is inefficient, in particular since all sub-queries with a shared portion are evaluated at the same site. To address this, Section 6.2.4 presents a technique that makes it possible to evaluate the common portions of these sub-queries once and then share the results obtained from these portions for all sub-queries.

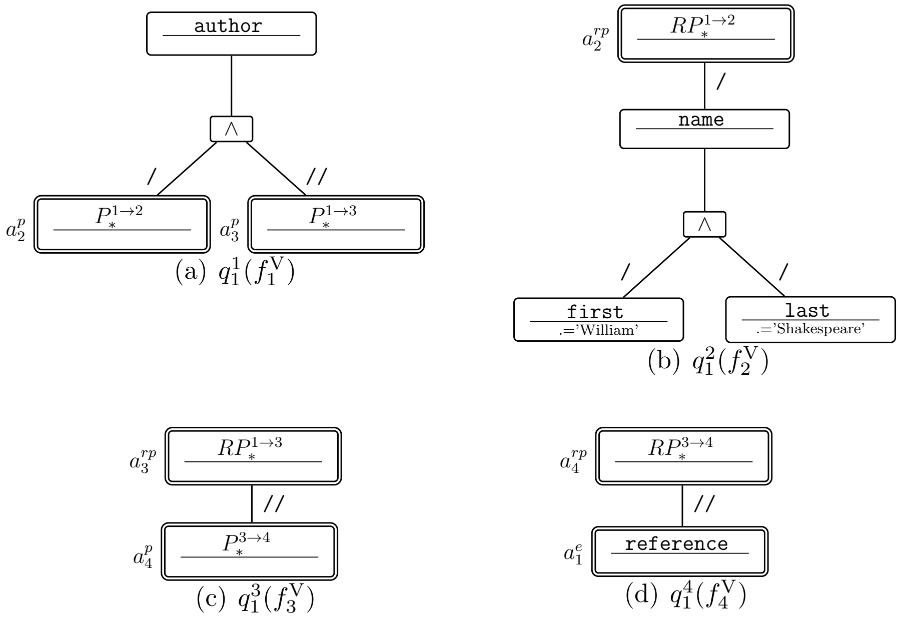
Finally, Section 6.2.5 presents a technique that avoids duplicate elimination in a DEP wherever this is possible. For the remaining cases, duplicate elimination is pushed into the DEP to reduce the size of intermediate results and thereby improve query performance.

6.2.1 Pruning Fragments

As in the horizontal scenario, the first technique for improving DEPs over vertically fragmented collections is based on pruning the set of fragments accessed to answer a given query. The localization strategy for vertical fragmentation (as described in Section 5.2) avoids accessing fragments whose node types are not reached by the global QTP. It does not, however, address a scenario where node types in a fragment are reached by the global QTP but no constraints are placed on nodes of these types. Consider, for example, the local QTP q_1^3 (shown in Figure 6.15(c)), which is evaluated over fragment f_3^{V}. The sole purpose of this local QTP is to determine which proxy nodes in f_1^{V} lead to which root proxy nodes in fragment f_4^{V}. No further constraints are placed on f_3^{V}.

The remainder of this section introduces a technique that makes it possible to avoid accessing such intermediate fragments, and thereby prune the local QTPs corresponding to these fragments from the DEP[2]. This is achieved by storing, in each root proxy node, information that makes it possible to identify all of its ancestor proxy nodes. Using this information, it is then possible to determine for any root proxy node in f_4^{V} which proxy node in f_1^{V} is its ancestor. Exploiting this, query q_1 can then be answered without accessing f_3^{V} and without evaluating the local QTP q_1^3. The benefit of this is twofold. First, the load on the site holding fragment f_3^{V} is reduced, since this fragment is not accessed. This has the potential to open up processing capacity for other queries and increases the overall query throughput of the system. Second, the cost of computing the results of q_1^3 and joining them with the results of other LQPs is avoided altogether, thereby improving the performance of query q_1.

[2] The work presented in this section has been published as a paper [83], and as technical reports [80, 81].



(a) $q_1^1(f_1^{\mathrm{V}})$

(b) $q_1^2(f_2^{\mathrm{V}})$

(c) $q_1^3(f_3^{\mathrm{V}})$

(d) $q_1^4(f_4^{\mathrm{V}})$

Figure 6.15: Local QTPs corresponding to query q_1

6.2.1.1 Encoding Ancestor-Descendant Relationships

A straightforward way of making the ancestor-descendant relationships between all proxy and root proxy nodes available to distributed query evaluation would be to store this information in a monolithic index structure. However, if this index structure is stored in a single, central location, it could easily become a bottleneck, severely limiting the scalability of distributed query evaluation. If this problem is addressed by replicating the index throughout the system, then a heavy performance penalty must be paid whenever the collection is updated (since all copies of the index have to be updated accordingly).

This problem is addressed by encoding the ancestor-descendant relationships in the numeric IDs assigned to each proxy/root proxy pair. By numbering proxy pairs according a numbering scheme based on the Dewey decimal system [44], it is possible to store the ancestor-descendant relationship in a distributed fashion, without relying on external index structures.

Definition 6.4. The *Dewey numbering scheme* assigns IDs to proxy nodes as follows.

- All proxy nodes occurring in the sub-trees of the root fragment are assigned sequential integer IDs.

- For a sub-tree s in non-root fragment f_i^{V}, let id(root(s)) denote the ID of the root proxy node in s. Then each proxy node in s is assigned an ID of the form $\mathrm{id}(\mathrm{root}(s)).y$, where y is an integer that is assigned uniquely and sequentially within the context of f_i^{V}.

Each root proxy node is assigned the ID of its corresponding proxy node. ■

In the following, each number in a Dewey ID is referred to as an *item* and the number of items in a Dewey ID is referred to as the ID's *length*. The ID $a = 1.23.4$, for example, consists of the items 1, 23, and 4; and $\mathrm{length}(a) = 3$. For a more concise notation, $a[r]$ refers to the rth item of a, such that $a[1] = 1$, $a[2] = 23$, and $a[3] = 4$

When ordering Dewey IDs, their hierarchical nature is taken into account, with the significance of items decreasing from first to last.

Definition 6.5. Let a and b be Dewey IDs. Then $a < b$ if

- there exists an integer $w \leq min\{\mathrm{length}(a), \mathrm{length}(b)\}$ such that for $r = 1, \ldots, w-1$, $a[r] = b[r]$ and $a[w] < b[w]$, or

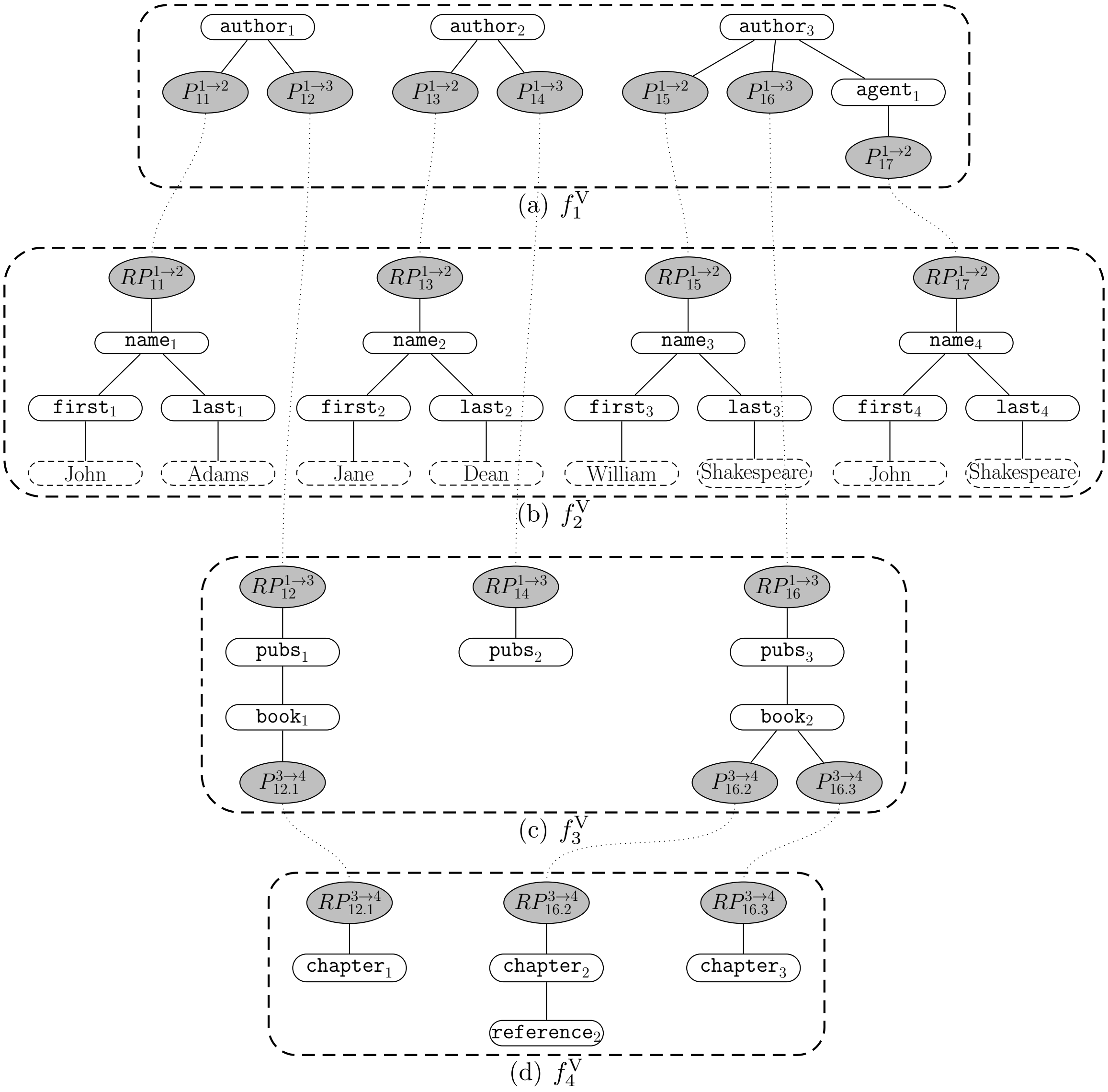
- $\mathrm{length}(a) < \mathrm{length}(b)$ and for $r = 1, \ldots, \mathrm{length}(a)$, $a[r] = b[r]$.

■

Similarly, the notion of a prefix of a Dewey ID is based on entire items, rather than the string representation of the ID.

Definition 6.6. Let a and b be Dewey IDs. Then a is a *prefix* of b if $\mathrm{length}(a) < \mathrm{length}(b)$ and for $r = 1, \ldots, \mathrm{length}(a)$, $a[r] = b[r]$. ■

The proxy pairs in the vertically fragmented collection shown in Figure 6.16 have been assigned IDs according to this numbering scheme. As can be seen, the proxy nodes in the



(a) f_1^V

(b) f_2^V

(c) f_3^V

(d) f_4^V

Figure 6.16: A vertically fragmented collection with Dewey IDs

root fragment f_i^{V} have sequential numerical IDs ranging from 11 to 17. Therefore, the root proxies corresponding to these proxy nodes have the same IDs (as can be seen in fragments f_2^{V} and f_3^{V}). For the proxy nodes in fragment f_3^{V}, the second case of the definition applies. Thus, the proxy node in the sub-tree rooted at the root proxy $RP_{12}^{1\to 3}$ is assigned the ID 12.1 and the proxy nodes in the sub-tree rooted at $RP_{16}^{1\to 3}$ are assigned the IDs 16.2 and 16.3. Finally, the root proxy nodes in fragment f_4^{V} are assigned the same IDs as their corresponding proxy nodes in f_3^{V}.

6.2.1.2 Pruning Intermediate Fragments

Assuming that all proxy/root proxy pairs in the collection have been assigned IDs according to the Dewey numbering scheme, it is possible to determine whether a given root proxy node $RP_b^{i\to j}$ is a descendant of a proxy node $P_a^{k\to l}$ by inspecting their IDs. $RP_b^{i\to j}$ is a descendant of $P_a^{k\to l}$ precisely when $\mathrm{id}(P_a^{k\to l})$ is a prefix of $\mathrm{id}(RP_b^{i\to j})$. If this is the case, then $P_a^{k\to l}$ occurs on a path from the root of one of the documents in the collection to $RP_b^{i\to j}$ and therefore $RP_b^{i\to j}$ is a descendant of $P_a^{k\to l}$. If $\mathrm{id}(P_a^{k\to l})$ is not a prefix of $\mathrm{id}(RP_b^{i\to j})$, then $P_a^{k\to l}$ does not occur on such a path and therefore $RP_b^{i\to j}$ cannot be one of its descendants.

During distributed query evaluation, this observation can be exploited to prune a local QTP from the query plan if it contains no value or structural constraints and no extraction

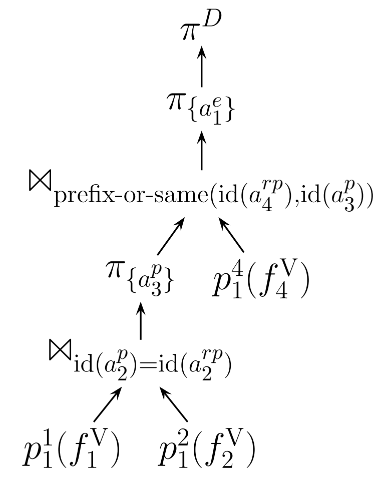

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\mathrm{id}(a_4^{rp}),\mathrm{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$p_1^4(f_4^{\mathrm{V}})$
$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$

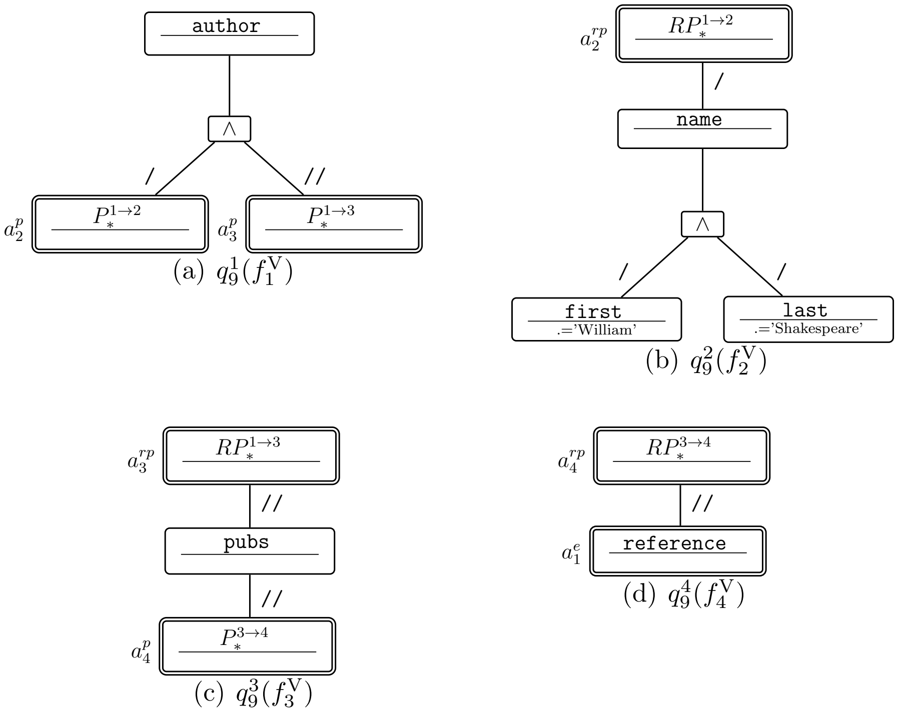
Figure 6.17: DEP for query q_1 after pruning

point pattern nodes other than those matching proxies or root proxies. For query q_1, this means that sub-query q_1^3 can be eliminated from the DEP. This is because sub-query q_1^3 is only needed to determine which root proxy node in fragment f_4^{V} is a descendant of which proxy node in fragment f_1^{V}, and this information can now be inferred from the Dewey IDs.

Therefore, the DEP for query q_1, can be rewritten as shown in Figure 6.17. Note that this plan does not include p_1^3 (the LQP representation of q_1^3). Also note how the join predicate prefix-or-same(id(a_4^{rp}), id(a_3^p)) checks whether a root proxy node from fragment f_4^{V} is a descendant of a proxy node from fragment f_1^{V}.

6.2.1.3 Pruning Fragments With Structural Constraints

The pruning technique presented in the previous section successfully prunes local QTPs that place no constraints on the nodes stored in their corresponding fragments. It does not,



(a) $q_9^1(f_1^{\mathrm{V}})$

(b) $q_9^2(f_2^{\mathrm{V}})$

(c) $q_9^3(f_3^{\mathrm{V}})$

(d) $q_9^4(f_4^{\mathrm{V}})$

Figure 6.18: Local QTPs corresponding to query q_9

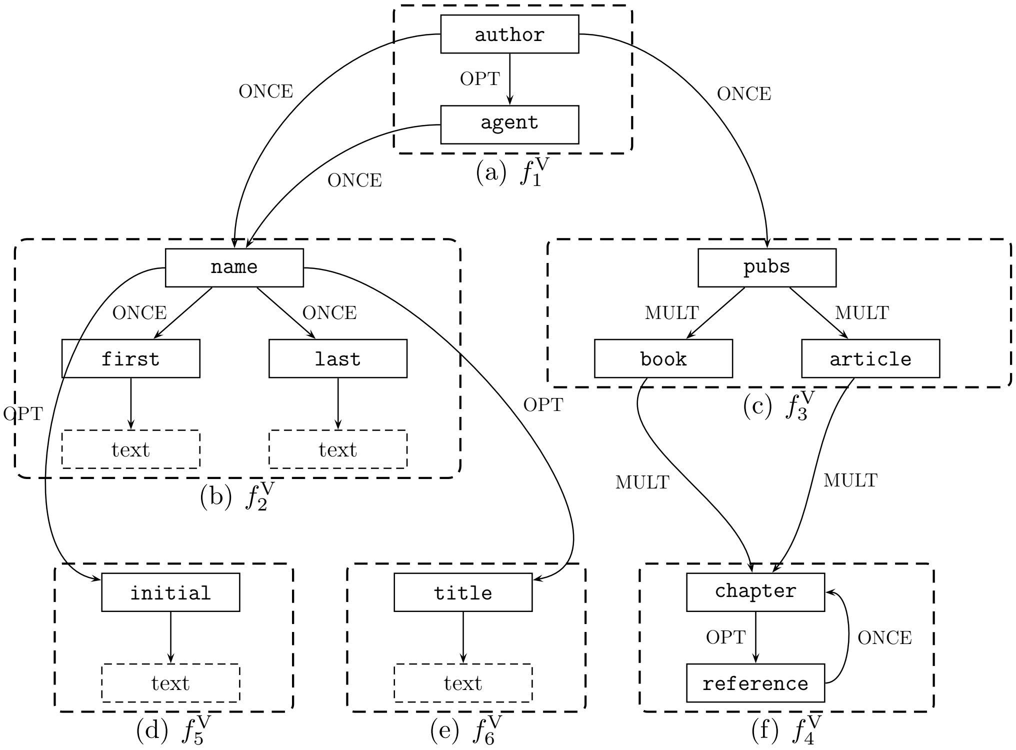


Figure 6.19: A vertical fragmentation schema

however, handle the scenario where a local QTP contains structural constraints. Consider, for example, query q_9, whose local QTPs are shown in Figure 6.18. For this query, the local QTP corresponding to fragment f_3^{V}, q_9^3 contains an additional structural constraint requiring a `pubs` node to occur on a path from a root proxy node to a proxy node. However, inspecting the fragmentation schema (shown in Figure 6.19) reveals that this `pubs` node is guaranteed to be present on any path through f_3^{V}. Formally, this can be expressed as follows.

Definition 6.7. Let q_k^u be a local QTP corresponding to fragment f_i^{V} and let q_k^v be a child QTP of q_k^u corresponding to fragment f_j^{V}. Then q_k^u is *structurally unambiguous* with respect to q_k^v if it matches all paths from a root proxy node in f_i^{V} to a proxy node $P_b^{i \to j}$ in f_i^{V} that correspond to an edge from fragment f_i^{V} to fragment f_j^{V}. Otherwise, q_k^u is *structurally ambiguous* with respect to q_k^v. ■

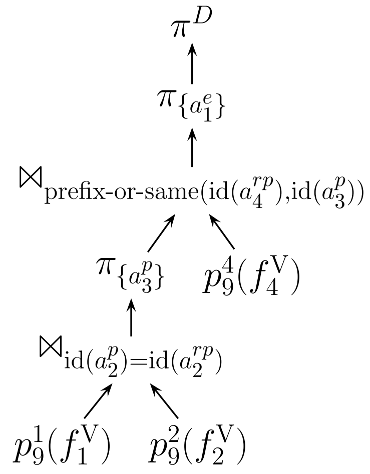

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$p_9^4(f_4^{\text{V}})$
$\bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})}$
$p_9^1(f_1^{\text{V}})$
$p_9^2(f_2^{\text{V}})$

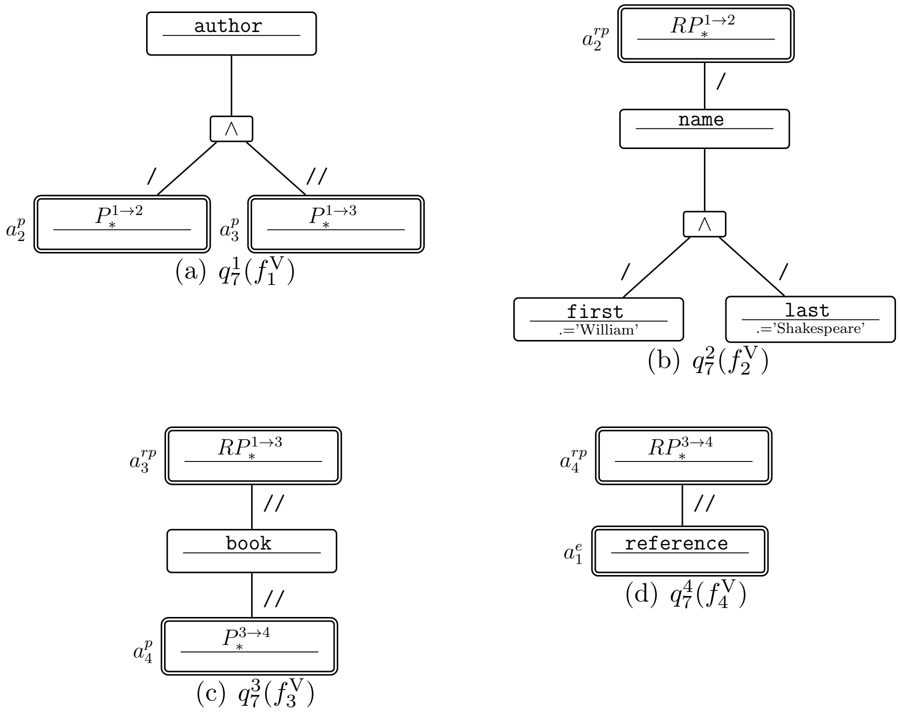
Figure 6.20: DEP for query q_9 after pruning

Whenever a local QTP is structurally unambiguous with respect to all of its child QTPs, then it does not pose any effective constraints on the nodes in its corresponding fragment. Therefore, it can be pruned from the DEP if it is traversed by a descendant step without additional constraints.

As can be seen in the fragmentation schema shown in Figure 6.19, local QTP q_9^3 is structurally unambiguous with respect to its child local QTP q_9^4. Therefore, LQP p_9^3 (the LQP corresponding to local QTP q_9^3) can be pruned from the DEP for query q_9. This results in the DEP shown in Figure 6.20.

6.2.1.4 Pruning Structurally Ambiguous LQPs

This section proposes a technique for pruning structurally ambiguous LQPs. Dewey IDs alone are insufficient to eliminate these LQPs from the DEP. Consider, for example, query q_7, whose local QTPs are shown in Figure 6.21. When evaluating this query over a vertically fragmented collection that conforms to the fragmentation schema shown in Figure 6.19, it is not possible to prune q_7^3. This is because fragment f_3^{V} contains two types of publications, `book` and `article`. This makes q_7^3 structurally ambiguous with respect to q_7^4, since it is needed to differentiate between references occurring in books (which are relevant for the query) and references occurring in articles (which are not).



(a) $q_7^1(f_1^{\mathrm{V}})$

(b) $q_7^2(f_2^{\mathrm{V}})$

(c) $q_7^3(f_3^{\mathrm{V}})$

(d) $q_7^4(f_4^{\mathrm{V}})$

Figure 6.21: Local QTPs corresponding to query q_9

To enable pruning in this scenario, node type path information is added to the ID of each proxy node $P_b^{i\to j}$ in fragment f_i^{V} to which a structurally ambiguous LQP corresponds. This information consists of the node types encountered on a path from the root of a sub-tree in f_i^{V} to the proxy node $P_b^{i\to j}$ in this sub-tree. This node type path information forms part of the locally unique identifier of the Dewey numbering scheme. Therefore, it is also part of the prefix of the IDs of all root proxy nodes that are descendants of $P_b^{i\to j}$. Formally, the Dewey numbering scheme with node type paths is defined as follows.

Definition 6.8. The *Dewey numbering scheme with node type paths* assigns IDs to proxy nodes as follows.

- All proxy nodes occurring in the sub-trees of the root fragment are assigned sequential integer IDs.

- If s_w is a sub-tree in non-root fragment f_i^{V} and there is a local QTP q_k^u, corresponding

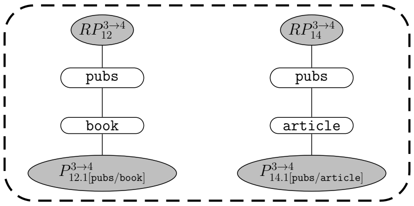

$RP_{12}^{3\to 4}$
pubs
book
$P_{12.1[\texttt{pubs/book}]}^{3\to 4}$
$RP_{14}^{3\to 4}$
pubs
article
$P_{14.1[\texttt{pubs/article}]}^{3\to 4}$

Figure 6.22: Fragment f_3^{V} with node type path IDs

to fragment f_i^{V} that is structurally ambiguous to q_k^v corresponding to fragment f_j^{V}, then each proxy $P_b^{i\to j}$ is assigned an ID of the form id(root(s_w)).y[$ntpath$], where y is an integer that is assigned uniquely and sequentially within the context of f_i^{V} and $ntpath$ is the sequence of node types encountered on the path from the root of s_w to $P_b^{i\to j}$. This can be stored compactly by assigning numeric IDs to each node type in the schema and by storing only the disambiguating node types (i.e., the node types that do not occur on every path through the fragment).

- If there is no structurally ambiguous local QTP that corresponds to fragment f_i^{V}, then each proxy node $P_b^{i\to j}$ in a sub-tree s_w in fragment f_i^{V} is assigned an ID of the form id(root(s_w)).y, where y is an integer that is assigned uniquely and sequentially within the context of f_i^{V}.

Each root proxy node is assigned the ID of its corresponding proxy node. ■

Figure 6.22 shows a sample instance of fragment f_3^V containing a `book` node and an `article` node. The proxy nodes in this fragment have been assigned IDs according to the Dewey numbering scheme with node type paths.

Node type paths contain enough information to evaluate structural constraints occurring within a linear path query. Therefore, using node type paths, it is possible to prune the sub-query q_7^3 from the DEP for query q_7.

Figure 6.23 shows the DEP for query q_7 after sub-query q_7^3 has been pruned. In addition to a join predicate that checks whether the ID of a root proxy node from fragment f_4^{V} has the ID of a proxy node from fragment f_2^{V} as its prefix, a selection is inserted that filters the result of q_7^4 based on the node type paths encoded in the IDs of the root proxy nodes.

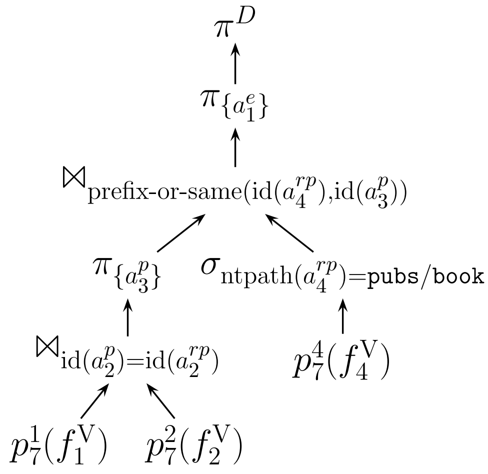

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$\sigma_{\text{ntpath}(a_4^{rp})=\texttt{pubs/book}}$
$\bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})}$
$p_7^4(f_4^{\text{V}})$
$p_7^1(f_1^{\text{V}})$
$p_7^2(f_2^{\text{V}})$

Figure 6.23: DEP for query q_7 after pruning using node type paths

6.2.1.5 Analysis

Proxy nodes are numbered according to the Dewey numbering scheme at fragmentation time. When a sub-tree is inserted into or deleted from the collection, only the proxy IDs in this sub-tree and in the siblings of the root of the inserted sub-tree are affected.

The vertical pruning techniques proposed here operate solely on the QTP representation of the query and on the fragmentation schema. Thus the performance of pruning is independent of the size of the data and of the constants used in value constraints. This makes it possible to perform pruning at query compile time, thereby minimizing its runtime overhead.

Node type paths are not only useful for pruning but also for skipping irrelevant sub-trees within a fragment. This use of node type paths is discussed in detail in Section 6.2.2.4.

6.2.2 Pipelining DEPs

As discussed in Section 5.2.4, DEPs for vertically fragmented collections combine the results of LQPs using cross-fragment joins. LQPs occur only as leaves in the DEP. Therefore, each LQP is evaluated independently without regard to how its result is joined with the results

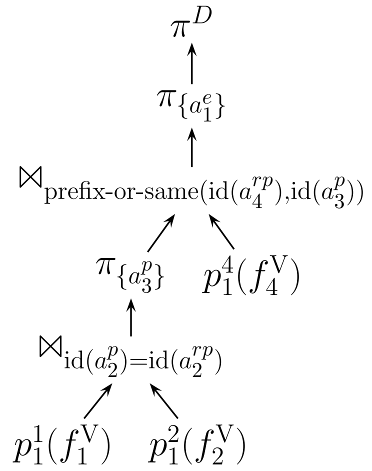

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$p_1^4(f_4^{\text{V}})$
$\bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})}$
$p_1^1(f_1^{\text{V}})$
$p_1^2(f_2^{\text{V}})$

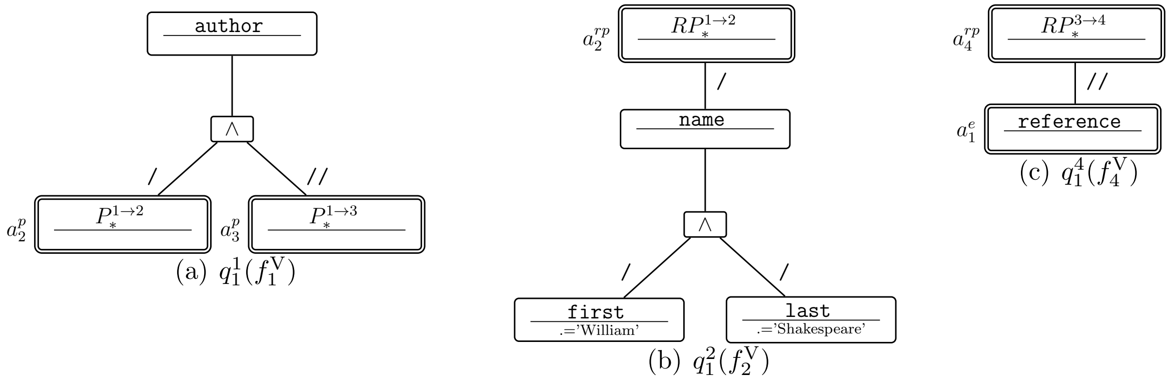
Figure 6.24: DEP for query q_1 after pruning

of other LQPs. In many cases, this leads to a scenario where a large number of matches are computed, only to be subsequently discarded by a cross-fragment join.

For an example of this situation, consider the DEP for query q_1 shown in Figure 6.24, whose local QTPs are shown in Figure 6.25. While this plan avoids accessing fragment f_3^{V} due to the pruning technique described in the previous section, evaluating the local sub-query q_1^4 over fragment f_4^{V} potentially yields a large number of matches that are subsequently discarded. This is because matches are computed for all references in the collection. Then, when the matches obtained from f_4^{V} are joined with the results obtained from the other fragments, only those matches are retained that correspond to references in publications authored by William Shakespeare. All other matches, corresponding to references in publications by other authors, are discarded.

This section introduces a technique that avoids computing these unnecessary matches[3]. This is done by pipelining the matches generated by a parent LQP $p_k^u(f_i^{\text{V}})$ into a child LQP $p_k^v(f_j^{\text{V}})$ and then filtering out the sub-trees in the child fragment f_j^{V}, for which $p_k^u(f_i^{\text{V}})$ does not yield a matching proxy node. The benefit of this pipelining strategy is twofold. First, it reduces the size of the result of p_k^v, which reduces the amount of data that needs to be transmitted. Second, it also makes it possible to skip a portion of the sub-trees in

[3] The work presented in this section has been published in [82].



(a) $q_1^1(f_1^{\mathrm{V}})$

(b) $q_1^2(f_2^{\mathrm{V}})$

(c) $q_1^4(f_4^{\mathrm{V}})$

Figure 6.25: Local QTPs corresponding to query q_1

f_j^{V}, thus reducing the I/O and processing cost of evaluating p_k^v. Together, these factors can significantly improve the performance and scalability of distributed query evaluation over vertically fragmented collections beyond the level achieved by the pruning technique presented in the previous section alone.

6.2.2.1 Pushing Cross-Fragment Joins

As discussed in Section 3.2.2, each LQP contains a scan of the roots of the sub-trees in the corresponding fragments. For non-root query plans this corresponds to a scan of the root proxy nodes in the fragment. Therefore, the cross-fragment join between the parent LQP $p_k^u(f_i^{\mathrm{V}})$ and the child LQP $p_k^v(f_j^{\mathrm{V}})$ can be pushed into p_k^v so that the join is performed between the result of $p_k^u(f_i^{\mathrm{V}})$ and the root proxy nodes in f_j^{V}. This filters out all root proxy nodes for which no join partner is found in the result of $p_k^u(f_i^{\mathrm{V}})$. p_k^v is then evaluated only over the sub-trees rooted at the remaining root proxy nodes.

To express this formally, the remainder LQP of an LQP is defined as the LQP with the scan of the root proxy nodes removed.

Definition 6.9. Let p_k^v be an LQP corresponding to fragment f_j^{V} and let fragment f_j^{V} be a child fragment of fragment f_i^{V}. Then $\bar{p}_k^v$ is the *remainder LQP* of p_k^v iff $p_k^v = \bar{p}_k^v(\mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}})$. ■

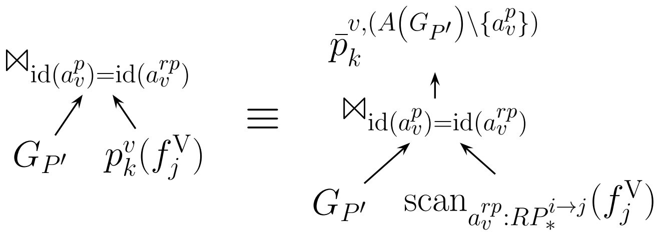

$\bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})}$
$G_{P'}$
$p_k^v(f_j^{\mathrm{V}})$
$\equiv$
$\bar{p}_k^{v,(A(G_{P'})\setminus\{a_v^p\})}$
$\bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})}$
$G_{P'}$
$\mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}(f_j^{\mathrm{V}})$

Figure 6.26: Cross-fragment join pushing rewrite

It is now possible to push a cross-fragment join into p_k^v by inserting it between the scan of the root proxy nodes and the remainder LQP $\bar{p}_k^v$. However, it may be necessary to pass additional attributes (corresponding to extraction points in other fragments) through $\bar{p}_k^v$ and the projections contained in $\bar{p}_k^v$ may prevent this. To address this issue, a modified remainder LQP is defined where projections are changed to pass through the necessary attributes.

Definition 6.10. Let $\bar{p}_k^v$ be the remainder LQP of an LQP p_k^v corresponding to fragment f_j^{V} and let fragment f_j^{V} be a child fragment of fragment f_i^{V}. Then $\bar{p}_k^{v,A}$ is a *modified remainder LQP* passing through the set of attributes A if $\bar{p}_k^{v,A}$ is equivalent to $\bar{p}_k^v$, except that for each projection $\pi_{A'}$ in $\bar{p}_k^v$, $\bar{p}_k^{v,A}$ contains the projection $\pi_{(A'\cup A)\setminus\{a_v^{rp}\}}$. ■

Based on this definition, it is now possible to define a rewrite that pushes the cross-fragment join between the scan of the root proxy nodes and the modified remainder LQP $\bar{p}_k^{v,A}$.

Definition 6.11. The cross-fragment join $G_{P'} \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} p_k^v(f_j^{\mathrm{V}})$ in a DEP can be rewritten to the *pushed cross-fragment join*

$$\bar{p}_k^{v,(A(G_{P'})\setminus\{a_v^p\})}\left(G_{P'} \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}(f_j^{\mathrm{V}})\right)$$

where $A\left(G_{P'}\right)$ denotes the set of attributes returned by $G_{P'}$. ■

A graphical representation of this rewrite is shown in Figure 6.26. Note that a cross-fragment join can only be pushed if its right-hand side is a single LQP (rather than another cross-fragment join). This means that, while all the cross-fragment joins in a left-deep plan can be rewritten, the same is not true for other plan shapes.

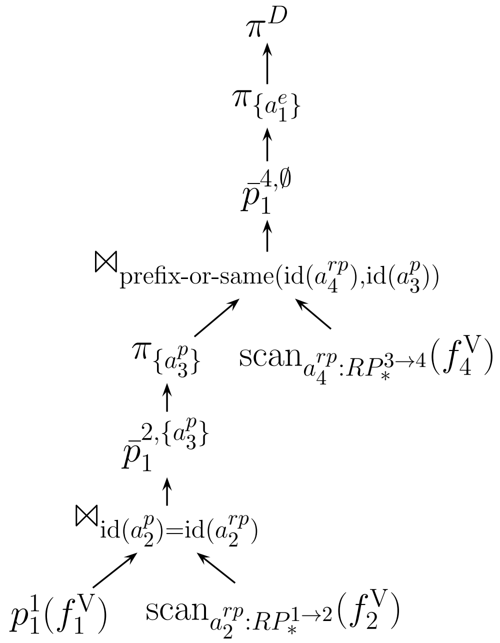

π^D
$\pi_{\{a_1^e\}}$
$\bar{p}_1^{4,\emptyset}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$\text{scan}_{a_4^{rp}:RP_*^{3\to 4}}(f_4^{\text{V}})$
$\bar{p}_1^{2,\{a_3^p\}}$
$\bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})}$
$p_1^1(f_1^{\text{V}})$
$\text{scan}_{a_2^{rp}:RP_*^{1\to 2}}(f_2^{\text{V}})$

Figure 6.27: DEP for query q_1 with pushed joins

Applying this rewrite to the cross-fragment joins in the DEP for query q_1 yields the DEP shown in Figure 6.27. Because the join between p_1^1 and p_1^2 is pushed, only those document sub-trees in fragment f_2^{V} are accessed, for which $p_1^1(f_1^{\text{V}})$ yields a matching proxy node. Similarly, a cross-fragment join is pushed into p_1^4 (note the modified join predicate due to the use of pruning), filtering out irrelevant sub-trees in fragment f_4^{V}.

6.2.2.2 Supporting Cross-Fragment Join Pushing

For performance and autonomy reasons, each site is free to choose the local query evaluation strategy that is most appropriate to the fragments stored at this site and to the sub-queries evaluated over these fragments. Therefore, in general, DEPs are optimized without access to the individual LQPs. Nevertheless, in order to be able to push cross-fragment joins into LQPs, the sites holding the individual fragments are required to implement an API that accepts the following parameters:

- a flag indicating whether a regular LQP or a modified remainder LQP should be generated, and

- if a modified remainder LQP is specified,
  - the set of attributes A that are to be passed through the modified remainder LQP,
  - the cross-fragment join that is to be inserted between the scan of the root proxy nodes and the modified remainder LQP, and
  - the site and fragment from which the left-hand side input of the cross-fragment join will be pipelined.

Using this API, which can easily be implemented at each site, it is possible to perform cross-fragment join pushing as described in this section without having access to the individual LQPs.

6.2.2.3 Maintaining Parallelism

Without cross-fragment join pushing, LQPs are evaluated completely independently of each other. Therefore, in general, all LQPs can be evaluated in parallel. Cross-fragment join pushing, however, introduces dependencies between the local plan evaluation at different sites. This section describes how a high level of parallelism can be maintained in the presence of these dependencies.

Whenever an LQP contains a pushed cross-fragment join, its execution has to be delayed until results from the DEP sub-plan on the left-hand side of the pushed cross-fragment join have arrived. Only then can the cross-fragment join be performed and the LQP be evaluated over the relevant document sub-trees. Waiting for the entire result of the left-hand side of the join, however, would effectively serialize the evaluation of LQPs and eliminate parallelism in distributed query execution.

To address this problem, pipelined execution is used. With this approach, LQP evaluation with a pushed cross-fragment join has to wait only for the first result tuple from the left-hand side of the cross-fragment join before it can start identifying the first relevant sub-tree. This significantly reduces the delay introduced by pushing cross-fragment joins.

To enable pipelined execution, a physical join operator has to be chosen that does not materialize the result of its left-hand side input. Assuming that local query results are returned ordered by their proxy IDs (which is easily achieved by the techniques mentioned in Section 3.2.2), and that the sub-trees in a fragment are stored ordered by the IDs of their root proxy nodes, a merge join operator with full pipelining on both inputs can be used.

If these conditions are not met, a hash join can be used, which builds a hash table on its right-hand side input (i.e., the root proxy nodes) and then probes this table for each tuple from the left-hand side input. Using a hash join operator is not detrimental to pipelining because the hash table on a fragment's root proxy nodes is not query-dependent, and can thus be built ahead of time. This yields an index on the root proxy nodes in a fragment that makes it possible to efficiently retrieve the relevant sub-trees.

6.2.2.4 Node Type Path Filtering

While pushing cross-fragment joins can lead to significantly improved performance, it can only be fully applied to left-deep plans. Furthermore, waiting for the first input tuple for each cross-fragment join might result in a non-trivial reduction of parallelism in certain cases. Consider, for example, a case where the parent LQP produces only a single tuple. Evidently, the child LQP has to wait for this tuple, which effectively serializes the execution of both LQPs.

In these cases, it may be better not to push certain cross-fragment joins in the DEP. This section introduces a technique that can deliver part of the performance advantage of pushing cross-fragment joins but places no constraints on the shape of a DEP and has no impact on parallelism.

The idea behind this technique is based on node type paths as defined in Section 6.2.1.4. If each root proxy node is annotated with the sequence of node types encountered on a path from the root of a document to the root proxy node, then this information can be exploited for filtering out sub-trees that cannot possibly contribute to the result of the query.

$$\begin{tikzcd}
 & & \bar{p}_k^{v,} \\
p_k^v(f_j^{\mathrm{V}}) & \equiv & \sigma_{\mathrm{ntpath}(a_v^{rp}) \in L_k^v} \arrow[u] \\
 & & \mathrm{scan}_{a_v^{rp}:RP_*^{i \rightarrow j}}(f_j^{\mathrm{V}}) \arrow[u]
\end{tikzcd}$$

Figure 6.28: Node type path rewrite

By unrolling descendant steps in the query into child steps and inserting disjunctions for path alternatives as necessary (using Algorithm 4 from page 121), it is possible to obtain, for each LQP $p_k^v(f_j^{\mathrm{V}})$, the set of node type paths L_k^v from the root of a document to the root of a relevant sub-tree in fragment f_j^{V}. The root proxy nodes in fragment f_j^{V} can then be filtered based on their node type paths and the remainder LQP $\bar{p}_k^v$ can be evaluated over the sub-trees whose root proxy nodes match one of the node type paths in L_k^v. Formally, this can be expressed as follows.

Definition 6.12. Let p_k^v be a non-root LQP of query q_k corresponding to fragment f_j^{V}. Let L_k^v be the set of node type plans leading to sub-trees of f_j^{V} that are relevant for p_k^v within the context of q_k. Then p_k^v can be rewritten to the *node-type path filtered LQP*

$$\hat{p}_k^v(f_j^{\mathrm{V}}) := \bar{p}_k^v\left(\sigma_{\mathrm{ntpath}(a_v^{rp}) \in L_k^v}\left(\mathrm{scan}_{a_v^{rp}:RP_*^{i \rightarrow j}}(f_j^{\mathrm{V}})\right)\right)$$

■

Applying this rewrite ensures that only those document sub-trees are considered during LQP evaluation whose root proxy nodes have node type paths that are compatible with the query. Figure 6.28 shows a graphical version of the node type path rewrite.

Applying this rewrite to LQP p_7^4 for query q_7 yields the DEP shown in Figure 6.29. For p_7^4, the set of node type paths that is compatible with the query consists of the single path `/author/pubs/book`. Therefore, the root proxy nodes in fragment f_4^{V} are selected based on this node type path. For the sub-trees corresponding to the root proxies that pass through the selection, the remainder LQP $\bar{p}_7^4$ is evaluated. Note that with node-type path filtering, no additional attributes are passed through the remainder LQP. Therefore, it is not necessary modify the projections in the remainder LQP.

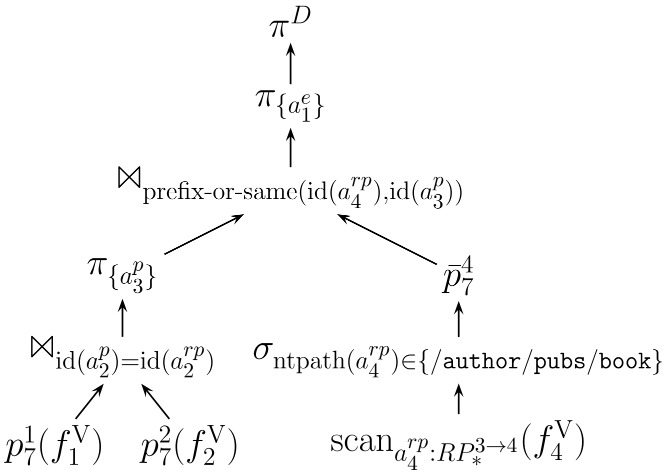

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^p))}$
$\pi_{\{a_3^p\}}$
$\bar{p}_7^4$
$\bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})}$
$\sigma_{\text{ntpath}(a_4^{rp})\in\{\texttt{/author/pubs/book}\}}$
$p_7^1(f_1^{\text{V}})$
$p_7^2(f_2^{\text{V}})$
$\text{scan}_{a_4^{rp}:RP_*^{3\to 4}}(f_4^{\text{V}})$

Figure 6.29: DEP for query q_7 with node type path filtering

6.2.2.5 Analysis

Pushing cross-fragment joins reduces the size of intermediate results that have to be shipped and combined with results from other sites. In this respect, the effect of this technique is similar to that of using a semi-join, as is frequently done in distributed relational systems [115]. In relational systems, semi-joins are mainly used as a means to reduce the communication cost of distributed query evaluation. These approaches use a semi-join to reduce the size of a partial result before it is shipped across the network, however, they generally require an additional inner join to assemble the overall query result. Therefore, with these techniques, the reduced communication cost achieved by semi-joins comes at the expense of increased processing cost.

With cross-fragment join pushing, on the other hand, only a single join operator is used. More importantly, in addition to reducing communication cost by achieving smaller intermediate result sizes, cross-fragment join pushing also reduces the cost of local query evaluation by skipping irrelevant sub-trees within a fragment.

The reason why pushing cross-fragment joins works well in XML database systems are the complex (and therefore expensive) structural constraints in the XML query model. In relational systems, it is usually preferable to push selections past join operations in order to reduce the cost of the join. Interestingly, here, the opposite is true: it is potentially

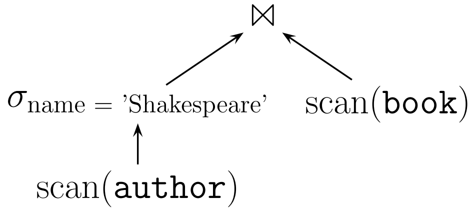


Figure 6.30: Relational plan for which magic set optimization is possible

beneficial to push a join past the operators that evaluate the structural constraints of a query (corresponding, conceptually, to a selection).

There are also some parallels between cross-fragment join pushing and a query optimization technique using magic sets [19, 111]. In both cases, tuples are filtered if it can be determined that they will be eliminated during a subsequent join. For magic sets, this can be inferred from selection predicates on the other input of the join. For example, when evaluating the simple relational query plan shown in Figure 6.30, it might be possible to scan only those rows of the `book` table whose author is Shakespeare if an appropriate index on the table `book` is available.

In contrast to this, no such inference is necessary in the case of cross-fragment join pushing. Rather than predicting which sub-trees of a fragment might lead to results that survive the subsequent cross-fragment join, the results of the other input to the join are streamed into an LQP and only sub-trees for which such a result is received are accessed.

6.2.3 Join Ordering

Another important factor when optimizing a DEP is the problem of determining in which order the results of LQPs should be joined together.

The problem of join ordering has been studied extensively within the context of relational database systems [115] and much of this work is applicable here. In broad terms, join ordering focuses on reducing the size of intermediate results by performing the most selective joins (i.e., the joins that return the smallest number of output tuples for a given number of input tuples) first. This has the effect of reducing the cost of subsequent joins.

In the scenario considered in this work, this effect is particularly pronounced when cross-fragment join pushing is used, since pipelining a large number of results through an LQP increases the processing cost of evaluating the LQP. Performing a highly selective join before pipelining its result through the same LQP, on the other hand, can significantly reduce the cost of LQP evaluation.

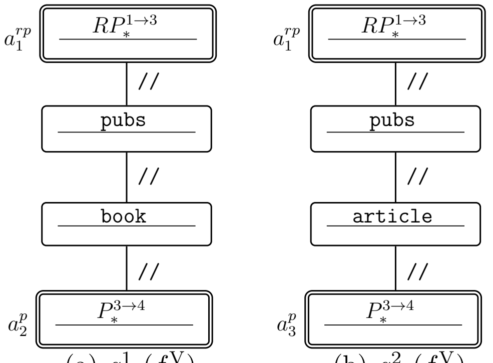
At the same time, cross-fragment join pushing places restrictions on the join orders that can be used in a DEP. To push all the cross-fragment joins in a DEP, it is necessary that the DEP be left-deep. In some cases, this requires the optimizer to trade off the performance benefit achieved by optimizing join order with the performance impact of cross-fragment join pushing. Due to this complexity, the most advantageous join order for a given query and distributed collection is best determined by a cost model, which is the topic of Chapter 7.

6.2.4 Combining Local Sub-Queries

When evaluating queries over vertically fragmented collections, in many cases, it is necessary to split a local QTP that contains a disjunction logic node (cf. Section 5.2.5). This can result in multiple local QTPs that correspond to the same fragment and that share a large common portion. Evaluating these local QTPs separately from each other duplicates the processing that corresponds to the shared portion of the local QTPs.

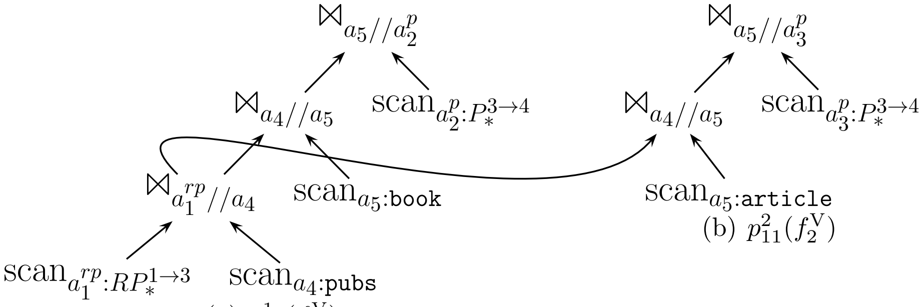
For example, consider the local QTPs $q_{11}^{1}(f_{2}^{\mathrm{V}})$ and $q_{11}^{2}(f_{2}^{\mathrm{V}})$, shown in Figure 6.31. These QTPs share the descendant step between the root proxy pattern node and the pattern node with the node test `pubs`. Thus, evaluating this step twice is redundant and eliminating this redundancy could help improve performance.

To avoid the overhead of evaluating shared portions of local QTPs repeatedly, it is possible to evaluate these portions once and then share the result for each local QTP. As long as the shared portion is contiguous and occurs at the root of the local QTP (as is the case for shared portions resulting from splitting local QTPs with disjunction logic nodes), this can be achieved for both navigational and structural-join based local query evaluation techniques.



(a) $q_{11}^1(f_2^V)$ (b) $q_{11}^2(f_2^V)$

Figure 6.31: Local QTPs with shared portion


$\bowtie_{a_5//a_2^p}$
$\bowtie_{a_4//a_5}$
$\text{scan}_{a_2^p:P_*^{3\to4}}$
$\bowtie_{a_1^{rp}//a_4}$
$\text{scan}_{a_5:\texttt{book}}$
$\text{scan}_{a_1^{rp}:RP_*^{1\to3}}$
$\text{scan}_{a_4:\texttt{pubs}}$
$\bowtie_{a_5//a_3^p}$
$\bowtie_{a_4//a_5}$
$\text{scan}_{a_3^p:P_*^{3\to4}}$
$\text{scan}_{a_5:\texttt{article}}$

(b) $p_{11}^2(f_2^V)$

(a) $p_{11}^1(f_2^V)$

Figure 6.32: Combined structural-join based LQPs for $q_{11}^1(f_2^V)$ and $q_{11}^2(f_2^V)$

Structural-join based local query evaluation (as described in Section 3.2.2.2) is based on scanning the nodes in a collection and then performing joins between them. To share the result of evaluating a common portion of a QTP, it is therefore possible to duplicate the sequence of tuples returned by one of these joins and use the same sequence as an input to multiple structural joins in different LQPs. Figure 6.32 shows an example of a combined LQP for $q_{11}^1(f_2^V)$ and $q_{11}^2(f_2^V)$. As can be seen, the output of the join $\bowtie_{a_1^{rp}//a_4}$ serves as the input for two operators, one in each LQP shown.

With navigational plans the same optimization is possible. In this case, the sequence of tuples returned by an unnest map operator is duplicated and used in multiple LQPs.

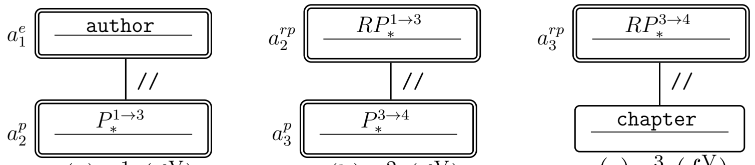
Optimizing the evaluation of LQPs with shared portions can be implemented at the individual sites. This is possible because this optimization is independent of how the results of the LQPs are used in the DEP. A similar optimization strategy could also be used for LQPs that belong to different queries. This would make it possible to share results if two queries with overlapping local QTPs are running simultaneously.

6.2.5 Duplicate Elimination

Another opportunity for improving DEPs is the way duplicate results are handled. While LQPs perform their own duplicate elimination, additional duplicates can be created when the results of LQPs are joined together. DEPs handle this problem by performing a duplicate elimination after the results of all LQPs have been combined (cf. Section 5.2.4). However, by leaving duplicate elimination until the end of distributed query evaluation, some intermediate results may be unnecessarily large, which increases the cost of evaluating subsequent cross-fragment joins. This problem is particularly pronounced when duplicate results are pipelined into an LQP through cross-fragment join pushing, since in this case, the LQP may need to be evaluated multiple times for the same sub-tree, thus increasing the cost of local query processing in addition to increasing the cost of performing the cross-fragment joins.

A cross-fragment join may produce duplicate results whenever the sub-plan on its right-hand side produces result tuples that consist only of the root proxy nodes on which the join is performed. When joining with a single LQP, this corresponds to the scenario where the QTP representation of the local plan does not contain any extraction points other than the root proxy pattern node at its root.

For an example of this, consider query q_{10}, whose local QTPs are shown in Figure 6.33. After pruning, this query can be evaluated using the DEP shown in Figure 6.34.



(a) $q_{10}^1(f_1^{\mathrm{V}})$ (b) $q_{10}^2(f_3^{\mathrm{V}})$ (c) $q_{10}^3(f_4^{\mathrm{V}})$

Figure 6.33: Local QTPs corresponding to query q_{10}

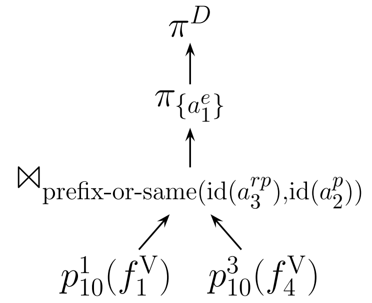

π^D
$\pi_{\{a_1^e\}}$
$\bowtie_{\text{prefix-or-same}(\text{id}(a_3^{rp}),\text{id}(a_2^p))}$
$p_{10}^1(f_1^{\text{V}})$
$p_{10}^3(f_4^{\text{V}})$

Figure 6.34: DEP for query q_{10}

Evaluating p_{10}^1 and p_{10}^3 over their respective fragments (shown in Figure 6.16 on page 140), yields the sequences of tuples shown in Figure 6.35 (denoted as $R(p_{10}^1(f_1^{\text{V}}))$ and $R(p_{10}^3(f_4^{\text{V}}))$, respectively).

Performing the cross-fragment join $p_{10}^1(f_1^{\text{V}}) \bowtie_{\text{prefix-or-same}(\text{id}(a_3^{rp}),\text{id}(a_2^p))} p_{10}^3(f_4^{\text{V}})$ results in three tuples, which are shown in Figure 6.36. As can be seen, the result containing the node `author`$_3$ is returned twice. This is because, within the document corresponding to this author, there are two `chapter` nodes and therefore two sub-trees within fragment f_4^{V}. Since $p_{10}^3(f_4^{\text{V}})$ produces a match for each of these sub-trees, the cross-fragment join yields two results for the node `author`$_3$, introducing a duplicate.

To address this problem, it is possible to push the duplicate elimination into the DEP such that it is performed immediately after the join that introduces the duplicates. This eliminates duplicates as soon as they are introduced, thus avoiding any negative performance impact on subsequent joins or LQPs with pushed cross-fragment joins.

Using semi-joins in place of full inner joins when combining the results of multiple

$[a_1^e = \texttt{author}_1, a_2^p = P_{12}^{1\to3}]$
$[a_1^e = \texttt{author}_2, a_2^p = P_{14}^{1\to3}]$
$[a_1^e = \texttt{author}_3, a_2^p = P_{16}^{1\to3}]$

(a) $R(p_{10}^1(f_1^{\text{V}}))$

$[a_2^{rp} = RP_{12}^{1\to3}, a_3^p = P_{12.1}^{3\to4}]$
$[a_2^{rp} = RP_{16}^{1\to3}, a_3^p = P_{16.2}^{3\to4}]$
$[a_2^{rp} = RP_{16}^{1\to3}, a_3^p = P_{16.3}^{3\to4}]$

(b) $R(p_{10}^2(f_3^{\text{V}}))$

$[a_3^{rp} = RP_{12.1}^{3\to4}]$
$[a_3^{rp} = RP_{16.2}^{3\to4}]$
$[a_3^{rp} = RP_{16.3}^{3\to4}]$

(c) $R(p_{10}^3(f_4^{\text{V}}))$

Figure 6.35: LQP results for query q_{10}

$$[a_1^e = \texttt{author}_1]$$

$$[a_1^e = \texttt{author}_3]$$

$$[a_1^e = \texttt{author}_3]$$

Figure 6.36: $R\left(p_{10}^1(f_1^{\mathrm{V}}) \bowtie_{\text{prefix-or-same}(\text{id}(a_3^{rp}),\text{id}(a_2^{p}))} p_{10}^3(f_4^{\mathrm{V}})\right)$

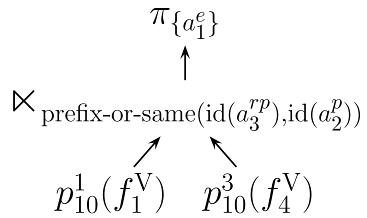

$\pi_{\{a_1^e\}}$
$\ltimes_{\text{prefix-or-same}(\text{id}(a_3^{rp}),\text{id}(a_2^{p}))}$
$p_{10}^1(f_1^{\mathrm{V}})$
$p_{10}^3(f_4^{\mathrm{V}})$

Figure 6.37: DEP for query q_{10} with semi-join

LQPs can be helpful to reduce the number of duplicate results. In the case of query q_{10}, duplicates can be avoided entirely by replacing the cross-fragment join with a semi-join, as is shown in the DEP in Figure 6.37. Since the schema requires that for each `author` node in fragment f_1^{V}, there can be at most one proxy node $P_*^{1\to3}$, a semi-join is sufficient to avoid generating duplicates.

Unfortunately, this technique does not completely avoid duplicates in all cases. To illustrate this, consider what happens when q_{10} is evaluated without pruning $p_{10}^2(f_3^{\mathrm{V}})$. Figure 6.38 shows a DEP for this scenario. Joining $p_{10}^1(f_1^{\mathrm{V}})$ and $p_{10}^2(f_3^{\mathrm{V}})$ yields the tuples shown in Figure 6.39. Performing a semi-join between these tuples and the result of $p_{10}^3(f_4^{\mathrm{V}})$ then yields the same result as shown in Figure 6.36, including the duplicate node $\texttt{author}_3$. Therefore, the duplicate elimination shown at the root of the DEP in Figure 6.38 is necessary to avoid returning duplicate results.

The reason why the semi-join fails to avoid duplicates in this case is because the left-hand side input of the semi-join (shown in Figure 6.39) already contains multiple tuples with the same node $\texttt{author}_3$. This, in turn, is caused by the fact that the schema specifies that for each root proxy node in fragment f_3^{V}, there can be multiple proxy nodes $P_*^{3\to4}$.

In general, inserting a duplicate elimination as close as possible to the cross-fragment

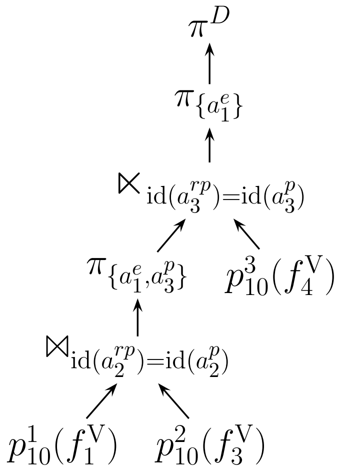

π^D
$\pi_{\{a_1^e\}}$
$\ltimes_{\mathrm{id}(a_3^{rp})=\mathrm{id}(a_3^p)}$
$\pi_{\{a_1^e,a_3^p\}}$
$p_{10}^3(f_4^{\mathrm{V}})$
$\bowtie_{\mathrm{id}(a_2^{rp})=\mathrm{id}(a_2^p)}$
$p_{10}^1(f_1^{\mathrm{V}})$
$p_{10}^2(f_3^{\mathrm{V}})$

Figure 6.38: Un-pruned DEP for query q_{10} with semi-join

$$[a_1^e = \texttt{author}_1, a_3^p = P_{12.1}^{3\to4}]$$
$$[a_1^e = \texttt{author}_3, a_3^p = P_{16.2}^{3\to4}]$$
$$[a_1^e = \texttt{author}_3, a_3^p = P_{16.3}^{3\to4}]$$

Figure 6.39: $R\left(p_{10}^1(f_1^{\mathrm{V}}) \bowtie_{\mathrm{id}(a_2^{rp})=\mathrm{id}(a_2^p)} p_{10}^2(f_3^{\mathrm{V}})\right)$

join that introduces duplicates can help improve the performance of distributed query evaluation. While semi-joins can be useful for avoiding duplicates in certain scenarios, using them cannot replace a duplicate elimination in all cases.

6.3 Summary

This chapter has introduced a suite of techniques that can be used to improve the performance and scalability of query evaluation over fragmented and distributed XML collections. For both horizontal and vertical fragmentation, pruning techniques are proposed that avoid accessing certain fragments altogether if it can be shown that they are not needed to answer the query. In the case of vertical fragmentation, additional pruning is accomplished by storing structural information in the IDs associated with each proxy and root proxy node. Another technique for improving the performance of query evaluation over vertically fragmented collections is based on pushing cross-fragment joins into the LQPs. This effectively skips irrelevant sub-trees within a fragment and thereby improves the performance of local query evaluation and reduces the size of intermediate results. Additional improvement techniques presented in this chapter focus on avoiding sorting and duplicate elimination steps.

Chapter 7

Cost-Based Optimization of Distributed Execution Plans

Chapter 5 describes how an initial DEP can be generated that evaluates a query over a distributed collection. Based on this, Chapter 6 introduces a suite of techniques that can improve the performance of these initial DEPs. Some of these techniques, such as the horizontal and vertical pruning techniques presented in Sections 6.1.1 and 6.2.1, respectively, are always beneficial and can therefore be applied indiscriminately, regardless of the query that is being evaluated. In the case of the pruning techniques, this is because removing irrelevant LQPs (and their corresponding fragments) from a DEP can never have a negative impact on query performance and does not interfere with the applicability of further techniques for improving distributed execution plans.

In contrast to this, most of the other techniques described in Chapter 6 may have a negative or a positive impact on query performance, depending on the characteristics of the query and the distributed collection. Cross-fragment join pushing, for example, can improve query performance by skipping sub-trees that only yield results that would subsequently be discarded. However, applying this technique indiscriminately might also reduce performance in cases where few such sub-trees can be skipped and the decrease in parallelism caused by this technique leads to significant delays. Thus, to obtain the best query performance, it is necessary to choose the query evaluation techniques that are appropriate for a given query and distributed collection when constructing a DEP.

An additional problem arises from the fact that the techniques described in Chapters 5 and 6 only specify the logical operators that are used in a DEP. For each of these logical operators, there are frequently multiple physical operators that produce the same result but that differ in their performance characteristics. For the best query performance, the most advantageous physical implementation of each operator must be chosen based on the characteristics of the query and the distributed collection.

As discussed in Section 6.2.3, there are frequently multiple possible ways to order the cross-fragment joins in a DEP. Join order affects the size of intermediate results and the applicability of cross-fragment join pushing. Thus, choosing the right join order is an important consideration when constructing a DEP.

This chapter introduces a cost-based optimization strategy that addresses these problems. For a given query and distributed collection, the most advantageous DEP is constructed based on the techniques presented in Chapters 5 and 6. In addition, for each logical operator specified by these techniques, a suitable physical implementation (or physical operator) is chosen, resulting in an optimized *physical DEP*. The optimization strategy works by enumerating the space of candidate DEPs representing the various techniques that can be used to evaluate a given query over a given distributed collection. Then, the cost of each candidate DEP is estimated and the DEP with the lowest estimated cost is chosen.

The notion of cost used here is based on the end-to-end response time of a DEP. By defining cost in terms of response time, all components that contribute to the cost of processing a query are taken into account. At the same time, modeling cost based on response time naturally captures parallelism, which is one of the key objectives of the distributed query evaluation techniques presented in this thesis.

In the data centre environment considered in this work, the most significant contributor to the response time of a DEP is the time it takes to evaluate the LQPs contained in this DEP. Thus, the cost estimation formulas presented in this chapter estimate the cost of a DEP by composing the costs of its constituent LQPs, while taking into account the degree to which these LQPs can be parallelized. This approach is well suited to the distributed query evaluations presented in this thesis since these techniques vary in the amount of parallelism they permit. For example, distributed execution plans without join pushing allow for the completely independent (and therefore freely parallelizable) execution of all LQPs. In the presence of join pushing, in contrast, the result of one LQP is pipelined into another and, therefore, a producer-consumer relationship is induced which reduces the amount of parallelism that is achievable.

To illustrate how the response time cost of a DEP can be estimated, consider Figure 7.1, which shows a DEP consisting of two LQPs (p_1^1 and p_1^2) connected by a structural join ($\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$). Assuming that it takes 10 seconds to evaluate p_1^1 and 8 seconds to evaluate p_1^2, the total computation time spent for both LQPs is 18 seconds. However, since p_1^1 and p_1^2 can be evaluated in parallel, it is possible to evaluate both LQPs in 10 seconds (the maximum of 10 seconds for p_1^1 and 8 seconds for p_1^2).

To obtain the overall response time of the DEP, it is necessary to take into account the cost of joining the results of the two LQPs. It is important to note that the cost of performing the join $\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$ depends on which physical join operator is selected to evaluate this join. Figure 7.2 shows two alternative physical DEPs that correspond to the logical DEP shown in Figure 7.1. Assuming that the tuples obtained by evaluating p_1^1 are ordered by the attribute a_2^p and that the tuples obtained from p_1^2 are ordered by a_2^{rp}, it is possible to use a merge join (denoted as $\bowtie^{\mathrm{M}}$), which can be fully parallelized with the evaluation of the LQPs. Using the merge join operator, join tuples are produced as soon as the first input tuples have been received from p_1^1 and p_1^2. Assuming that both LQPs are fully pipelined internally and that the results of the LQPs are pipelined into the join, this happens after a delay that is much shorter than the full response time of 8 or 10 seconds, respectively. Thus, the overall response time for evaluating the physical DEP shown in Figure 7.2(a) is 10 seconds (assuming that the overhead of generating the last join tuple and transmitting tuples between sites is negligible). If the order requirements for a merge join are not met, one option is to use a simple, one-sided hash join (denoted as $\bowtie^{\mathrm{H}}$), which

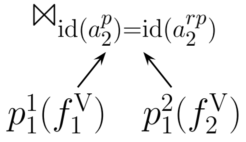

$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$

Figure 7.1: A logical DEP

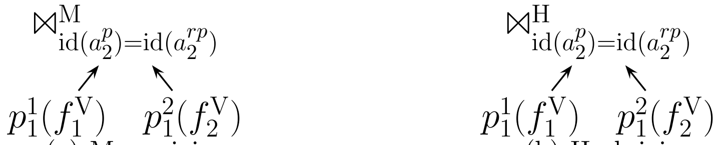

$\bowtie^{\mathrm{M}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$
$\bowtie^{\mathrm{H}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$
(a) Merge join (b) Hash join

Figure 7.2: Two physical DEPs

fully materializes one of its inputs in a hash table before join processing begins and then probes this hash table for all tuples received from the other input. In this case (shown in Figure 7.2(b)), the overall response time of the physical DEP additionally contains the time spent probing the hash table and can thus be expected to be greater than 10 seconds.

The remainder of this chapter introduces a technique for determining the DEP with the lowest estimated response time for a given query and distributed collection. After stating the assumptions of this technique in Section 7.1, five main components are presented:

Plan properties Response time cost estimates for a DEP are calculated in a bottom-up fashion based on a set of properties that are tracked for each part of the plan (i.e., for the sub-plans of the DEP including the LQPs contained in it). Section 7.2 describes these properties in detail.

Some properties depend only on the logical DEP. For example, the cardinality of the output of a join is independent of which algorithm is used to execute this join. Other properties, however, do depend on the physical operators chosen for a given physical DEP. For instance, the response time cost of a DEP containing a join depends on the join strategy chosen, as does the order in which result tuples are returned.

Optimizing LQPs and obtaining LQP properties Section 7.3 describes how the LQPs contained in a DEP are optimized. Since each LQP is evaluated at a single site, this is done using existing, centralized optimization and cost estimation techniques. Many of the properties needed by the distributed optimizer are also inferred using these models, however, to determine the order properties of LQP results, additional steps are necessary. Once LQP optimization is complete, the properties of the best LQPs are relayed back to the dispatcher, where they are used to optimize the DEP.

Obtaining DEP properties Section 7.4 describes how the plan properties of a DEP are computed. Since DEPs consist of LQPs and operators that combine the results of these LQPs to the overall query result (e.g., cross-fragment join or merge), this can be done in a bottom-up fashion, one operator at a time. For many of the operators discussed in Chapters 5 and 6, there exist multiple physical implementations with varying requirements and performance characteristics. This is taken into account and properties that are affected by this choice are computed separately for each physical operator.

Special attention is paid to dependencies in the execution of the individual parts of a DEP. If there are dependencies (such as in the case where the result of one LQP is pipelined into another through cross-fragment join pushing), the time that various parts of the DEP spend waiting for each other needs to be taken into account when estimating the cost of the DEP.

Enumerating DEP alternatives Section 7.5 describes how the possible physical DEPs for a query can be enumerated using existing plan enumeration techniques. This makes it possible to estimate the overall response time cost for each of the candidate DEPs for a given query and finally choose the DEP with the lowest estimated cost.

Execution and dynamic adaptation of DEPs Choosing the DEP with the lowest estimated cost performs well when cost estimates reasonably match actual query cost. However, there are cases when the cost estimates would be inaccurate. This could occur, for example, in the presence of heavily skewed data or outdated statistics (and therefore inaccurate cost estimates for the LQPs in a DEP). To increase the robustness of DEP performance in these situations, Section 7.6 outlines a technique that can dynamically change DEPs during query execution to adapt to cost estimation errors.

Figure 7.3 shows an overview of how these components work together during query processing. As described in Chapters 5 and 6, the query is first decomposed into local QTPs for each fragment and irrelevant local QTPs are pruned. Then the remaining local QTPs are sent to the sites holding their corresponding fragments. For each local QTP

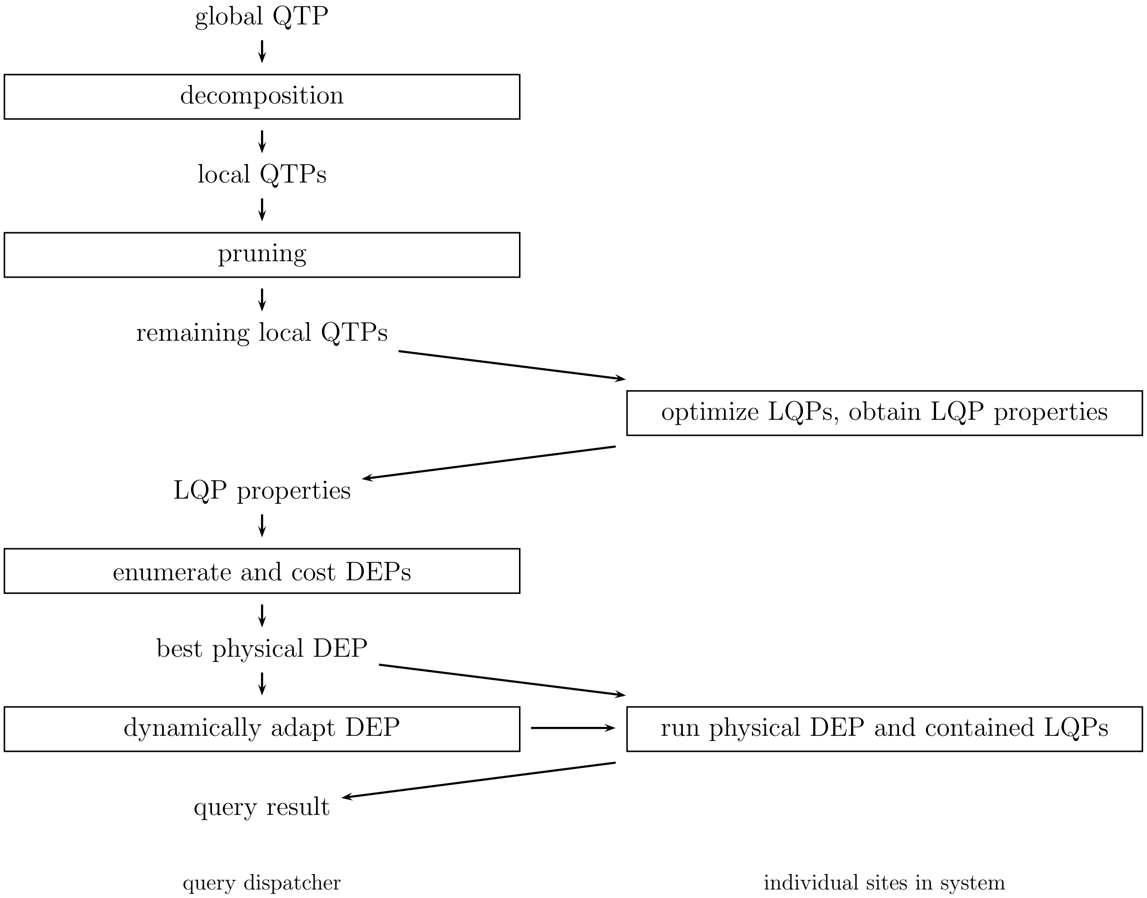


Figure 7.3: Distributed query processing overview

received, the site holding the corresponding fragment determines the best *physical LQPs*. Since physical LQPs for the same local QTP may differ in the order in which their results are returned and since this order may have an impact on the performance of the DEP as a whole, in general the cost estimates for multiple candidate physical LQPs are relayed to the dispatcher. Using this information, the dispatcher then determines the best physical DEP by enumerating candidate DEPs and comparing their estimated cost. Finally, the chosen physical DEP is evaluated across the sites of the system. Simultaneously, the dispatcher monitors the performance of the individual components of the DEP and dynamically modifies the DEP if necessary. As the query result becomes available, it is sent to the dispatcher. Query execution ends once the dispatcher has received the entire query result.

7.1 Assumptions

To make it feasible to optimize a DEP and to derive accurate cost estimates, a few assumptions are necessary. While these assumptions represent a simplification of the characteristics of distributed query evaluation, they do not prevent cost-based optimization from finding good plans.

Independent execution of LQPs All LQPs in a DEP are assumed to be executed without interference from each other (other than the dependencies induced by the DEP, such as the producer-consumer relationship between LQPs when join pushing is used). This is a realistic assumption, since, within the context of a single query, the number of LQPs over the same fragment tends to be small, and fragments are assumed to be on independent machines. In fact, for many queries, only one LQP is generated for each fragment, in which case this assumption is fully satisfied. Even in the case where multiple LQPs are generated for the same fragment (and hence are executed at the same site), assuming sufficient resources at the corresponding site (such as multiple processor cores and sufficient I/O bandwidth, as are commonly encountered in a data centre) interference caused by resource contention can be expected to be small.

Independent execution of combining operators As with LQPs, the combining operators (e.g., cross-fragment join or merge) in a DEP are assumed to be executed independently of each other without affecting each other's performance. This can easily be achieved by limiting the number of such operators that are executed on the same machine. While this may increase the amount of data that needs to be transmitted over the network, in the pipelining model with high-throughput connections considered here, this does not pose a significant problem.

Intermediate results returned at steady rate To simplify the cost model, for each sub-plan of a DEP it is assumed that results are returned at a steady rate from the time the first tuple is returned until the last tuple is returned. While, in practice, there will be fluctuations in this rate, for most cases this represents a reasonable approximation of reality. Note that this assumption does not imply that all sub-plans are fully pipelined. As will be shown later, for sub-plans that materialize results before returning them, the time span between first tuple and last tuple will be short and thus the steady rate assumption covers only a short period of time.

No correlations between the results of LQPs When estimating the cardinality of a cross-fragment join, it is necessary to make the usual independence assumption between the operands of the join. In the scenario seen here, this means that the results of LQPs are assumed to be free of correlations and skew. In cases where this assumption does not hold, the quality of cost estimates may be reduced. To address this problem, it is possible to monitor distributed query execution and dynamically change the DEP as necessary (cf. Section 7.6).

7.2 Plan Properties

The objective of the cost model presented here is to determine the overall response time cost of a physical DEP. To estimate this response time cost accurately, several additional properties are tracked for each portion of a physical DEP. Each of these portions, which are referred to in the following as physical sub-plans, consists of a subset of the LQPs in the DEP and the physical operators that combine the results of these LQPs. The properties of the sub-plans are then combined in a bottom-up fashion to obtain the overall cost of the physical DEP.

Certain tracked properties depend only on the logical operators used in a DEP. It is therefore possible to compute these properties based on logical DEPs and their logical sub-plans and then share them for all physical DEPs corresponding to the same logical DEP. Thus, the plan properties considered by the cost model described here can be divided into two categories:

Logical plan properties depend on the logical operators (e.g., join or merge) used in a DEP. These can be computed based on the logical DEP and shared among all physical DEPs corresponding to this logical DEP.

Physical plan properties depend on the physical operators chosen (e.g., merge join or hash join) and thus have to be computed separately for each physical DEP. However, this does not preclude the optimizer from re-using the physical properties of a sub-plan if this sub-plan occurs in more than one physical DEP.

7.2.1 Logical Plan Properties

Logical plan properties are independent of the physical implementation chosen for a given operator. Thus, two sub-plans that differ only in their physical operators but that use the same logical operators in the same order will share the same logical properties. This makes it possible to determine these properties once and use them for both sub-plans.

Definition 7.1. Let $G_{P'}$ be a logical sub-plan of a logical DEP G_P. Then $\langle \mathrm{card}(G_{P'}), A\left(G_{P'}\right)\rangle$ is the *logical plan property vector* of $G_{P'}$, and the logical plan properties of $G_{P'}$ are defined as follows:

- $\mathrm{card}(G_{P'}) = |R(G_{P'})|$ is the estimated *cardinality* of $G_{P'}$, i.e., the number of tuples in $R(G_{P'})$ (the result of $G_{P'}$ when evaluated over the collection), and
- $A\left(G_{P'}\right)$ is the set of attributes of which the tuples returned by $G_{P'}$ consist.

■

7.2.2 Physical Plan Properties

Unlike logical plan properties, physical plan properties depend on the physical operators chosen in a sub-plan. For example, a sub-plan using a merge join will generally have physical properties that are different from those of a sub-plan that uses a hash join. This makes it necessary to compute physical properties separately for each physical sub-plan.

Definition 7.2. Let $G^{\mathrm{P}}_{P'}$ be a physical sub-plan of a physical DEP G^{P}_{P}. Then $\langle \mathrm{cost}(G^{\mathrm{P}}_{P'}),$ $\text{cost-first}(G^{\mathrm{P}}_{P'}), O\left(G^{\mathrm{P}}_{P'}\right)\rangle$ is the *physical plan property vector* of $G_{P'}$, and the physical plan properties of $G^{\mathrm{P}}_{P'}$ are defined as follows:

- $\mathrm{cost}(G^{\mathrm{P}}_{P'})$ is the estimated *cost* of $G^{\mathrm{P}}_{P'}$, i.e., the total end-to-end response time of evaluating $G^{\mathrm{P}}_{P'}$ over the collection,
- $\text{cost-first}(G^{\mathrm{P}}_{P'})$ is the estimated *time to the first tuple* returned by $G^{\mathrm{P}}_{P'}$, i.e., the portion of $\mathrm{cost}(G^{\mathrm{P}}_{P'})$ that elapses up to the point when the first result tuple is returned, and
- $O\left(G^{\mathrm{P}}_{P'}\right) \subseteq A\left(G^{\mathrm{P}}_{P'}\right)$ is the *set of order properties* of G^{P}_{P}, i.e., the subset of the attributes in $A\left(G_{P'}\right)$ that are in forward document order in $R(G^{\mathrm{P}}_{P'})$ for any distributed collection. For any attribute $a \in O\left(G^{\mathrm{P}}_{P'}\right)$, the nodes bound to attribute a in $R(G^{\mathrm{P}}_{P'})$ are in forward document order.

■

While $\mathrm{cost}(G^{\mathrm{P}}_{P'})$ provides an estimate of the response time cost of the sub-plan $G^{\mathrm{P}}_{P'}$ and $\text{cost-first}(G^{\mathrm{P}}_{P'})$ is useful for estimating to what extent the tuples resulting from $G^{\mathrm{P}}_{P'}$ can be pipelined into other parts of G^{P}_{P}, $O^{\mathrm{P}}_{G_{P'}}$ (the set of ordered attributes) is useful for two reasons:

- First, since XQ follows the XPath and XQuery semantics, the overall result of G^{P}_{P} (i.e., the nodes matched to the extraction point a^{e}_{1} in the global QTP representation of the query) has to be returned in document order. While in some cases this may require a sorting step before the query result can be returned, the overhead associated with sorting can be avoided if the result is already ordered as required (i.e., if $a^{e}_{1} \in O\left(G^{\mathrm{P}}_{P}\right)$).
- Second, the order of intermediate results affects the choice of physical operators for combining these intermediate results to the overall query result. For example, if both inputs to a cross-fragment join are ordered by the attribute on which the join is performed (i.e., the IDs of the proxy/root proxy nodes involved), then a fully pipelined merge join can be employed whereas otherwise this physical operator would require the insertion of an additional sort operator.

$$[a_1^e = \texttt{name}_1, a_v^p = P_8^{i \to j}]$$
$$[a_1^e = \texttt{name}_1, a_v^p = P_9^{i \to j}]$$
$$[a_1^e = \texttt{name}_2, a_v^p = P_4^{i \to j}]$$
$$[a_1^e = \texttt{name}_3, a_v^p = P_1^{i \to j}]$$
$$[a_1^e = \texttt{name}_3, a_v^p = P_3^{i \to j}]$$

Figure 7.4: Sequence of tuples R_1, hierarchically ordered by $[a_1^e, a_v^p]$

Keeping track of the order properties of sub-plans is a common feature of optimizers used in a relational context. As was first described by Selinger et al. [121], relational optimizers tend to consider only a restricted set of *interesting orders* that can be shown to be potentially useful for query optimization. For a sub-plan $G_{P'}^{\mathrm{P}}$ of the DEP G_P^{P}, each attribute in $A\left(G_{P'}^{\mathrm{P}}\right)$ represents a potentially useful order property and therefore an interesting order. Since cross-fragment joins are always combined with a projection that removes attributes that are no longer needed (cf. Definition 5.2 on page 91), each attribute a in $A\left(G_{P'}^{\mathrm{P}}\right)$ is either used in a join predicate somewhere further up in G_P^{P} (in which case having a in document order may allow the optimizer to use a merge join) or a is the overall extraction point of the query a_1^e and the overall query result must therefore be ordered by this attribute (in which case it would be necessary to sort by a if the result is not already ordered by this attribute).

It is important to point out what it means for $O\left(G_P^{\mathrm{P}}\right)$ to contain multiple attributes. Traditionally, order properties have usually been described as a hierarchy of attributes expressed as tuples. For example, if $R(G_P^{\mathrm{P}})$ is ordered by the sequence of attributes $[a_1, a_2]$, then it is assumed that $R(G_P^{\mathrm{P}})$ is ordered by a_1 (the most significant attribute in the order property $[a_1, a_2]$), and tuples in $R(G_P^{\mathrm{P}})$ that assign the same node to a_1 are then ordered by a_2. Figure 7.4 shows an example of this: the tuples in sequence R_1 are first ordered by a_1^e and tuples with the same `name` node in a_1^e are then ordered by a_v^p. As can easily be seen, the fact that $R(G_P^{\mathrm{P}})$ is ordered by $[a_1^e, a_v^p]$ does not imply that it is ordered by a_v^p on its own.

This work follows a different approach. Instead of keeping track of hierarchies of at-

$$[a_1^e = \texttt{name}_1, a_v^p = P_4^{i \to j}]$$
$$[a_1^e = \texttt{name}_1, a_v^p = P_4^{i \to j}]$$
$$[a_1^e = \texttt{name}_2, a_v^p = P_5^{i \to j}]$$
$$[a_1^e = \texttt{name}_3, a_v^p = P_6^{i \to j}]$$
$$[a_1^e = \texttt{name}_3, a_v^p = P_7^{i \to j}]$$

Figure 7.5: Sequence of tuples R_2, independently ordered by a_1^e and a_v^p

tributes by which a sequence of tuples is ordered, $O\left(G_P^{\mathrm{P}}\right)$ is a set of individual attributes, such that the sequence of tuples is fully ordered by each attribute in $O\left(G_P^{\mathrm{P}}\right)$, independently of the other attributes in $O\left(G_P^{\mathrm{P}}\right)$. According to these semantics, sequence R_1 is ordered by attribute a_1^e only. Since this sequence is not fully ordered by a_v^p, this attribute is not considered to be part of the order properties of this sequence.

For $O\left(G_P^{\mathrm{P}}\right)$ to contain multiple attributes, the result sequence $R\left(G_P^{\mathrm{P}}\right)$ must be fully ordered by each of these attributes on its own. For an example of this, consider sequence R_2, shown in Figure 7.5. This sequence is the result of a physical DEP G_P^{P} with $O\left(G_P^{\mathrm{P}}\right) = \{a_1^e, a_v^p\}$. As can be seen the tuples in this sequence are independently ordered by both attribute a_1^e and attribute a_v^p.

Another key factor that distinguishes the order properties considered here from order properties in traditional relational optimization is related to the fact that in the query execution model considered here, the attributes of each tuple have XML nodes as their values, rather than simple numeric or textual values. While it is possible to order XML nodes based on their values (for example, by comparing their node types or text content), in this work, they are instead ordered by their document order. For proxy and root proxy nodes, document order directly corresponds to the IDs of these nodes. Thus, the nodes assigned to attribute a_v^p in the example shown in Figure 7.5 are in document order since their IDs are 4, 4, 5, 6, and 7, respectively[1]. For other nodes, a subscript indicating position in document order is used for notational convenience, with $\texttt{name}_k$ denoting the kth node of

[1] Note that $P_4^{i \to j}$ is duplicated.

type `name` encountered in a pre-order traversal of the collection. Thus, the nodes assigned to attribute a_1^e in sequence R_2 are in document order and their relative positions are 1, 1, 2, 3, and 3, respectively. In the following, document order is expressed as follows: If o_i and o_j are nodes in a collection, then $o_i <_{\text{doc}} o_j$ denotes that o_i occurs before o_j in document order. Similarly, $o_i \leq_{\text{doc}} o_j$ denotes that either o_i occurs before o_j in document order or $o_i = o_j$.

7.3 Optimizing LQPs and Obtaining LQP Properties

LQPs form the building blocks of DEPs and their properties are used to estimate the overall cost of a DEP in a bottom-up fashion. Therefore, to accurately estimate the cost of a DEP, it is essential to obtain good estimates of the properties of the LQPs used within this DEP. This section describes how the properties of the LQPs in a DEP can be inferred using two main strategies. Many of the properties described in Section 7.2 (namely, total cost, time to first tuple, and set of attributes), can be obtained using existing cost estimation techniques. For other properties, such as the number of sub-trees accessed (which is important for estimating the impact of cross-fragment join pushing) and the order properties of the result, additional steps are necessary, which are described in this section.

While distributed query optimization relies on accurate LQP properties, it is independent of the exact LQPs used. Since LQPs are evaluated over a single fragment, the evaluation of an LQP is performed at a single site (the site where the corresponding fragment is stored). Therefore, LQPs can be optimized independently at the site holding their corresponding fragment. This is done using existing, centralized cost estimation techniques relying on locally available statistics. The properties of the best LQPs are then reported to the query dispatcher, where they are used during distributed optimization.

While the goal of this optimization is to find the best LQP corresponding to a given local QTP, due to the multidimensional nature of the properties it is not always possible to identify a single best LQP. For example, one LQP might have a lower cost, but another LQP might offer a more expansive set of order properties. In this case, it is impossible to decide which of these LQPs will allow the query dispatcher to construct the better DEP. Thus, a scenario might arise in which there are multiple LQPs for the same QTP, none of which dominates the others. The local optimizer handles this scenario by reporting the properties of each of the potentially optimal LQPs back to the query dispatcher. During distributed optimization, the dispatcher then determines which of these LQPs results in the best DEP.

The remainder of this section is organized as follows. First, Section 7.3.1 describes how the logical properties of LQPs are obtained. Then, Section 7.3.2 describes how physical LQPs are optimized and how their order properties can be inferred.

7.3.1 Logical LQP Properties

In this work, the logical properties of an LQP are the properties that are the same for any LQP p_k^u corresponding to a given local QTP q_k^u. These properties include the properties mentioned in Definition 7.1, i.e., the set of attributes of the tuples returned by p_k^u (denoted as $A\left(p_k^u\right)$) and the cardinality of p_k^u's result (denoted as $\mathrm{card}(p_k^u)$).

Obtaining $A\left(p_k^u\right)$ is straightforward. Since p_k^u is required to return exactly one attribute for each extraction point in its corresponding local QTP q_k^u, $A\left(p_k^u\right)$ consists of one attribute for each such extraction point (including the extraction points added during decomposition).

To obtain $\mathrm{card}(p_k^u)$, existing cardinality estimation techniques for the centralized evaluation of XML queries can be applied directly (e.g., [143, 38, 7, 135, 136, 8, 55, 131]). An overview of these techniques is provided in Section 3.3.1.2.

In addition to the logical properties tracked for DEPs, for LQPs, it is also useful to keep track of the number of sub-trees accessed. This is needed to predict the result sizes of cross-fragment joins accurately.

Definition 7.3. Let p_k^u be a logical LQP. Then $\mathrm{nsubt}(p_k^u)$ denotes the *number of sub-trees* accessed by p_k^u. ■

To determine $\mathrm{nsubt}(p_k^u)$, it is necessary to distinguish between three cases:

- If p_k^u is evaluated over a horizontal fragment f_i^{H}, then $\mathrm{nsubt}(p_k^u) = \mathrm{nsubt}(f_i^{\mathrm{H}})$.
- Similarly, if p_k^u is evaluated over the root fragment f_ρ^{V} in a vertical fragmentation then $\mathrm{nsubt}(p_k^u) = \mathrm{nsubt}(f_\rho^{\mathrm{V}})$.
- If p_k^u is evaluated over a non-root vertical fragment f_j^{V} and p_k^u's parent LQP is evaluated over the fragment f_i^{V}, then $\mathrm{nsubt}(p_k^u)$ only includes those sub-trees in f_j^{V} that are rooted at a root proxy node corresponding to an edge from fragment f_i^{V} to fragment f_j^{V}. Therefore, $\mathrm{nsubt}(p_k^u)$ is the number of root proxy nodes $RP_b^{i \to j}$ in fragment f_j^{V}.

While existing cost models for the centralized evaluation of queries over XML collections do not generally provide estimates for the number of sub-trees accessed by an LQP, this information can easily be obtained by tracking the number of root proxy nodes (i.e., the number of sub-trees) stored in a fragment as part of the distribution meta-data. In the case of non-root vertical fragments, the number of root proxy nodes in a fragment f_j^{V} has to be stored separately for each fragment from which there is an edge to f_j^{V}.

7.3.2 Physical LQPs

As discussed in Section 2.2.3.1, when a tree pattern is evaluated over a collection, pattern matches are returned such that the nodes that match the extraction point of the pattern are in document order. The XQ query model focuses on queries with a single extraction point. Therefore, the QTP representation of an entire query (referred to as the global QTP), also has a single extraction point (denoted as a_1^e). Thus, any centralized query plan evaluating this QTP has an order property consisting only of the attribute a_1^e (since the result of evaluating this plan is required to be ordered by a_1^e and, as a_1^e is the only extraction point, there are no other extraction points by which the result could be ordered).

In contrast to the global QTP, the local QTPs evaluated over the fragments of a vertically fragmented collection usually contain multiple extraction points. This is because additional extraction points matching proxy and root proxy nodes are inserted when the global QTP is decomposed (cf. Section 5.2). Figure 7.6 shows an example of a local QTP with two extraction points, corresponding to proxy nodes leading to two fragments.

A physical LQP can generally only ensure that its result is ordered by one of the extraction points in its corresponding local QTP. Thus, before generating a physical LQP, a single attribute (corresponding to a single extraction point) is chosen and designated as the *ordering attribute* (or the ordering extraction point when discussing the QTP representation). Only the nodes assigned to this attribute are guaranteed to be returned in document order. When evaluating the local QTP q_1^1 (shown in Figure 7.6), the result can either be ordered by extraction point a_2^p or it can be ordered by extraction point a_3^p, but generally not both.

Formally, physical LQPs can be defined as follows:

Definition 7.4. Let p_k^u be a logical LQP such that $A\left(p_k^u\right)$ is the set of attributes of the tuples returned by p_k^u. Then for each attribute $a \in A\left(p_k^u\right)$, ${}^{a}p_k^u$ denotes a physical LQP corresponding to p_k^u with *ordering attribute* a. ■

As mentioned before, for local QTP q_1^1, there are two possible choices for the ordering attribute. Each of these choices leads to a different physical LQP: The first choice, ${}^{a_2^p}p_1^1$, with ordering attribute a_2^p guarantees that, in its result, the nodes that match a_2^p are returned in document order. Conversely, the physical LQP ${}^{a_4^p}p_1^1$, whose ordering attribute is a_3^p yields a result in which the nodes that match a_3^p are in document order.

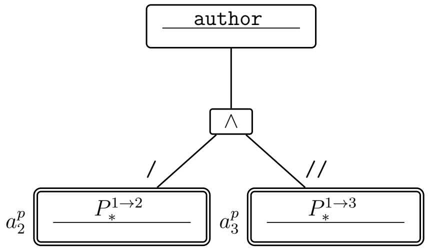


Figure 7.6: Local QTP $q_1^1(f_1^{\mathrm{V}})$

7.3.2.1 Physical LQP Properties

For each physical LQP ${}^{a}p_k^u$ corresponding to the logical LQP p_k^u, the physical properties defined in Definition 7.2 are tracked. These include cost$({}^{a}p_k^u)$, the total response time cost of ${}^{a}p_k^u$; cost-first$({}^{a}p_k^u)$, the time to the first tuple returned by ${}^{a}p_k^u$; and $O\left({}^{a}p_k^u\right)$, the set of order properties of ${}^{a}p_k^u$.

While cost$({}^{a}p_k^u)$ and cost-first$({}^{a}p_k^u)$ can easily be obtained using existing cost models for centralized query evaluation over XML collections (e.g., [140, 68, 67, 75], cf. Section 3.3.1.1), to obtain $O\left({}^{a}p_k^u\right)$ additional reasoning is necessary.

Since a is the ordering attribute of ${}^{a}p_k^u$, the result of ${}^{a}p_k^u$ is guaranteed to be ordered by a. From this follows that $a \in O\left({}^{a}p_k^u\right)$. However, it is possible that ${}^{a}p_k^u$ may also be ordered by additional attributes in $A\left(p_k^u\right)$.

For example, consider the local QTP shown in Figure 7.7. Evaluating ${}^{a_4^p}p_1^3$ (the physical LQP corresponding to q_1^3 with a_4^p chosen as the ordering attribute) yields result tuples such that the proxy nodes matched to a_4^p are returned in document order. However, note that by ordering the result tuples by a_4^p, they are also ordered by a_3^{rp}. This is because for each proxy node matched to a_4^p there is exactly one root proxy node matched to a_3^{rp}. Section 7.3.2.2 formalizes this and describes how the additional order properties of a physical LQP can be inferred.

When using pipelined execution with pushed cross-fragment joins (as described in Section 6.2.2), LQPs are only evaluated over some of the sub-trees in their corresponding fragment. To obtain accurate cost estimates for this scenario, it is necessary to estimate the cost of evaluating a physical LQP ${}^{a}p_k^u$ over a single sub-tree.

Definition 7.5. Let ${}^{a}p_k^u$ be a physical LQP. Then subtcost$({}^{a}p_k^u)$ is the *response time cost*

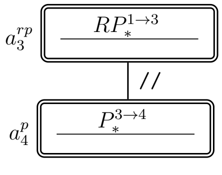


Figure 7.7: Local QTP $q_1^3(f_3^{\mathrm{V}})$

of evaluating ${}^{a}p_k^u$ over a single sub-tree in its corresponding fragment. ■

To estimate $\mathrm{subtcost}({}^{a}p_k^u)$, it is possible to apply the centralized cost model chosen to estimate $\mathrm{cost}({}^{a}p_k^u)$ directly.

7.3.2.2 Inferring LQP Order Properties

While only one attribute can be chosen as the ordering attribute of a physical LQP, in practice, ordering by one attribute frequently implies that the result of the LQP is also ordered by some of the other attributes in $A\left(p_k^u\right)$. This is the case when the order of these other attributes can be inferred from the ordering attribute.

To infer the order of additional attributes from the ordering attribute of a physical LQP, it is first necessary to define what it means to infer the order of one attribute from the order of another. Assume that the tuples returned by the physical LQP ${}^{a}p_k^u$ contain two attributes, a_{ord} and a_{imp}. To infer the order of a_{imp} from the order of a_{ord}, one needs to show that the following statement holds:

If $R({}^{a}p_k^u)$ is ordered by a_{ord}, $R({}^{a}p_k^u)$ must also be ordered by a_{imp}.

To show this, it is necessary to verify that for each sequence of tuples R produced by the LQP ${}^{a}p_k^u$ over any conceivable instance, if R is ordered by a_{ord}, then it must also be ordered by a_{imp}.

If R is ordered by a_{ord}, then for each pair of tuples $t_i, t_j \in R$ such that $i < j$ (denoting that t_i occurs before t_j in the sequence R), $t_i[a_{\mathrm{ord}}] \leq_{\mathrm{doc}} t_j[a_{\mathrm{ord}}]$ (i.e., the node from the collection assigned to attribute a_{ord} in tuple t_i occurs before or at the same location in document order as the node assigned to the same attribute in tuple t_j.

To show that R is also ordered by a_{imp}, one needs to show that this implies that $t_i[a_{\mathrm{imp}}] \leq_{\mathrm{doc}} t_j[a_{\mathrm{imp}}]$, i.e., that the node assigned to attribute a_{imp} in tuple t_i occurs before or at the same location in document order as the node assigned to this attribute in tuple t_j.

Formally, this can be expressed as follows:

$$[a_{\text{imp}} = RP^{1\to 3}_{1.2}, a_{\text{ord}} = P^{3\to 4}_{1.2.1}]$$
$$[a_{\text{imp}} = RP^{1\to 3}_{1.3}, a_{\text{ord}} = P^{3\to 4}_{1.3.1}]$$
$$[a_{\text{imp}} = RP^{1\to 3}_{1.4}, a_{\text{ord}} = P^{3\to 4}_{1.4.1}]$$
$$[a_{\text{imp}} = RP^{1\to 3}_{1.5}, a_{\text{ord}} = P^{3\to 4}_{1.5.1}]$$

Figure 7.8: Sequence of tuples R_3, $a_{\text{ord}} \rightsquigarrow a_{\text{imp}}$

Definition 7.6. Let ${}^{a}p^{u}_{k}$ be a physical LQP and let $a_{\text{ord}}, a_{\text{imp}} \in A\left(p^{u}_{k}\right)$ be attributes of the tuples returned by ${}^{a}p^{u}_{k}$. Then, the order of a_{ord} *implies* the order of a_{imp} (i.e., $a_{\text{ord}} \in O\left({}^{a}p^{u}_{k}\right) \implies a_{\text{imp}} \in O\left({}^{a}p^{u}_{k}\right)$) if for any sequence of tuples $R = [t_1, t_2, \ldots]$ resulting from evaluating ${}^{a}p^{u}_{k}$ over some instance of its corresponding fragment, the following holds:

$$\forall t_i, t_j \in R, i < j : t_i[a_{\text{ord}}] \leq_{\text{doc}} t_j[a_{\text{ord}}] \implies t_i[a_{\text{imp}}] \leq_{\text{doc}} t_j[a_{\text{imp}}]$$

■

As a shorthand, in the following, $a_{\text{ord}} \rightsquigarrow a_{\text{imp}}$ denotes that the order of a_{ord} implies the order of a_{imp}.

By applying this reasoning repeatedly, it is possible to infer the order of additional attributes from the order of the ordering attribute in many cases. To show that the order of one attribute implies the order of another, it is helpful to consider how document order can be expressed using the Dewey numbering scheme discussed in Section 6.4. Assuming that all nodes in a fragment are assigned Dewey IDs[2], then $o_1 \leq_{\text{doc}} o_2$ if and only if $\text{id}(o_1) \leq \text{id}(o_2)$.

Figure 7.8 shows an example of a sequence in which each node is annotated with its Dewey ID. As can be seen, the Dewey ID of each root proxy node assigned to attribute a_{imp} is a fixed-length prefix of the Dewey ID assigned to attribute a_{ord}. Assuming this sequence is known to be ordered by a_{ord} it is thus possible to infer that the sequence is also

[2] Note that when comparing Dewey IDs, their hierarchical nature must be taken into account. For details, refer to Definition 6.5 on page 139.

$$[a_{\text{imp}} = RP^{1\to 3}_{1.2.3}, a_{\text{ord}} = P^{3\to 4}_{1.2.3.5.9}]$$
$$[a_{\text{imp}} = RP^{1\to 3}_{1.2}, a_{\text{ord}} = P^{3\to 4}_{1.2.3.6}]$$

Figure 7.9: Sequence of tuples R_4, $a_{\text{ord}} \not\rightsquigarrow a_{\text{imp}}$

ordered by a_{imp}. The following lemma formalizes this notion and specifies the conditions that need to be satisfied for this to hold.

Lemma 7.1. Let ${}^{a}p^{u}_{k}$ be a physical LQP with $a_{\text{ord}}, a_{\text{imp}} \in A\left(p^{u}_{k}\right)$. Then $a_{\text{ord}} \rightsquigarrow a_{\text{imp}}$ if for any sequence of tuples R resulting from evaluating ${}^{a}p^{u}_{k}$ over its corresponding fragment f, one of the following two conditions holds:

1. (a) For any $t_i \in R$, $\text{id}(t_i[a_{\text{imp}}])$ is a prefix of $\text{id}(t_i[a_{\text{ord}}])$, and

   (b) there exists an integer $l > 0$ such that for any $t_i \in R$, $\text{length}(\text{id}(t_i[a_{\text{imp}}])) = l$, or

2. (a) for any $t_i \in R$, $\text{id}(t_i[a_{\text{ord}}])$ is a prefix of $\text{id}(t_i[a_{\text{imp}}])$, and

   (b) there exists an integer $l > 0$ such that for any $t_i \in R$, $\text{length}(\text{id}(t_i[a_{\text{ord}}])) = l$, and

   (c) for any two tuples $t_i, t_j \in R$, $\text{id}(t_i[a_{\text{ord}}]) = \text{id}(t_j[a_{\text{ord}}]) \implies \text{id}(t_i[a_{\text{imp}}]) = \text{id}(t_j[a_{\text{imp}}])$.

■

As can be seen, the lemma infers that the order of attribute a_{ord} implies the order of attribute a_{imp}. Condition 1 in Lemma 7.1 corresponds to the case where the Dewey ID of each node assigned to attribute a_{imp} is a prefix of the Dewey ID of the node assigned to attribute a_{ord} in the same tuple. This is precisely the scenario encountered in the example shown in Figure 7.8. As can be seen, the lemma additionally requires that the nodes that match a_{imp} occur at a fixed depth in their fragment (corresponding to Dewey IDs with a fixed number of items l).

To illustrate that the implication between ordering on a_{ord} and ordering on a_{imp} does not hold if the depth of the collection nodes matched to a_{imp} is not fixed, consider the Dewey IDs of the nodes in sequence R_4, shown in Figure 7.9. As can be seen, the sequence is ordered by attribute a_{ord} since 1.2.3.5.9 < 1.2.3.6 because 1.2.3.5.9[4] (the 4th item of this ID) is less than 1.2.3.6[4]. Also, it can be observed that for each tuple, the Dewey ID of the node assigned to attribute a_{imp} is a prefix of the Dewey ID of the node assigned to attribute a_{ord}. Nevertheless, R_4 is not ordered by a_{imp}, since 1.2.3 is longer than 1.2 and shares the same first two items.

Proof 7.1 formally shows why fixing the depth of the collection nodes matched to a_{imp} resolves this problem and why condition 1 in Lemma 7.1 is sufficient to determine that $a_{\mathrm{ord}} \rightsquigarrow a_{\mathrm{imp}}$.

Proof 7.1 (Lemma 7.1, condition 1)**.** Let $a_{\mathrm{ord}}, a_{\mathrm{imp}} \in A\left(p_k^u\right)$ such that condition 1 holds. Show that for any $t_i, t_j \in R$, $\mathrm{id}(t_i[a_{\mathrm{ord}}]) \leq \mathrm{id}(t_j[a_{\mathrm{ord}}]) \implies \mathrm{id}(t_i[a_{\mathrm{imp}}]) \leq \mathrm{id}(t_j[a_{\mathrm{imp}}])$. Assume the antecedent.

Case 1 $\mathrm{id}(t_i[a_{\mathrm{ord}}]) = \mathrm{id}(t_j[a_{\mathrm{ord}}])$:

Since Dewey IDs are unique, $t_i[a_{\mathrm{ord}}] = t_j[a_{\mathrm{ord}}]$

By condition 1(b), $\exists l > 0, \forall t' \in R, \mathrm{length}(\mathrm{id}(t'[a_{\mathrm{imp}}])) = l$.

Thus, $\mathrm{length}(\mathrm{id}(t_i[a_{\mathrm{imp}}])) = \mathrm{length}(\mathrm{id}(t_j[a_{\mathrm{imp}}])) = l$.

For $c = 1, \ldots, l, \mathrm{id}(t_i[a_{\mathrm{ord}}])[c] = \mathrm{id}(t_i[a_{\mathrm{imp}}])[c] = \mathrm{id}(t_j[a_{\mathrm{ord}}])[c] = \mathrm{id}(t_j[a_{\mathrm{imp}}])[c]$.

Therefore, $\mathrm{id}(t_i[a_{\mathrm{imp}}]) = \mathrm{id}(t_j[a_{\mathrm{imp}}])$.

Case 2 $\mathrm{id}(t_i[a_{\mathrm{ord}}]) < \mathrm{id}(t_j[a_{\mathrm{ord}}])$:

By condition 1(b), $\exists l > 0, \forall t' \in R, \mathrm{length}(\mathrm{id}(t'[a_{\mathrm{imp}}])) = l$.

Thus, $\mathrm{length}(\mathrm{id}(t_i[a_{\mathrm{imp}}])) = \mathrm{length}(\mathrm{id}(t_j[a_{\mathrm{imp}}])) = l$.

By Definition 6.6, $\forall t' \in R, \mathrm{length}(\mathrm{id}(t'[a_{\mathrm{ord}}])) > l$.

Therefore, $\mathrm{length}(\mathrm{id}(t_i[a_{\mathrm{ord}}])) > l$ and $\mathrm{length}(\mathrm{id}(t_j[a_{\mathrm{ord}}])) > l$.

Case 2.1 For $c = 1, \ldots, \text{length}(\text{id}(t_i[a_{\text{ord}}]))$, $\text{id}(t_i[a_{\text{ord}}])[c] = \text{id}(t_j[a_{\text{ord}}])[c]$ and $\text{length}(\text{id}(t_i[a_{\text{ord}}])) < \text{length}(\text{id}(t_j[a_{\text{ord}}]))$:

Since $l < \text{length}(\text{id}(t_i[a_{\text{ord}}]))$, for $c = 1, \ldots, l$, $\text{id}(t_i[a_{\text{ord}}])[c] = \text{id}(t_j[a_{\text{ord}}])[c]$.

Since, $\text{id}(t_i[a_{\text{imp}}])$ is a prefix of $\text{id}(t_i[a_{\text{ord}}])$ and $\text{length}(\text{id}(t_i[a_{\text{imp}}])) = l$ and since $\text{id}(t_j[a_{\text{imp}}])$ is a prefix of $\text{id}(t_j[a_{\text{ord}}])$ and $\text{length}(\text{id}(t_j[a_{\text{imp}}])) = l$, for $c = 1, \ldots, l$, $\text{id}(t_i[a_{\text{imp}}])[c] = \text{id}(t_i[a_{\text{ord}}])[c]$ and $\text{id}(t_j[a_{\text{imp}}])[c] = \text{id}(t_j[a_{\text{ord}}])[c]$.

Therefore, for $c = 1, \ldots, l$ $\text{id}(t_i[a_{\text{imp}}])[c] = \text{id}(t_j[a_{\text{imp}}])[c]$.

Thus, $\text{id}(t_i[a_{\text{imp}}]) = \text{id}(t_j[a_{\text{imp}}])$.

Case 2.2 $\exists w$ with $1 < w < \min\{\text{length}(\text{id}(t_i[a_{\text{ord}}])), \text{length}(\text{id}(t_j[a_{\text{ord}}]))\}$ s.t. for $c = 1, \ldots, w-1$, $\text{id}(t_i[a_{\text{ord}}])[c] = \text{id}(t_j[a_{\text{ord}}])[c]$ and $\text{id}(t_i[a_{\text{ord}}])[w] < \text{id}(t_j[a_{\text{ord}}])[w]$:

Case 2.2.1 $w \leq l$:

Since $\text{id}(t_i[a_{\text{imp}}])$ is a prefix of $\text{id}(t_i[a_{\text{ord}}])$ and $\text{id}(t_j[a_{\text{imp}}])$ is a prefix of $\text{id}(t_j[a_{\text{ord}}])$, for $c = 1, \ldots, l$, $\text{id}(t_i[a_{\text{ord}}])[c] = \text{id}(t_i[a_{\text{imp}}])[c]$ and $\text{id}(t_j[a_{\text{ord}}])[c] = \text{id}(t_j[a_{\text{imp}}])[c]$.

Thus, for $c = 1, \ldots, w-1$, $\text{id}(t_i[a_{\text{imp}}])[c] = \text{id}(t_j[a_{\text{imp}}])[c]$ and $\text{id}(t_i[a_{\text{imp}}])[w] < \text{id}(t_j[a_{\text{imp}}])[w]$.

Thus, $\text{id}(t_i[a_{\text{imp}}]) < \text{id}(t_j[a_{\text{imp}}])$.

Case 2.2.2 $w > l$:

Since $\text{id}(t_i[a_{\text{imp}}])$ is a prefix of $\text{id}(t_i[a_{\text{ord}}])$ and $\text{id}(t_j[a_{\text{imp}}])$ is a prefix of $\text{id}(t_j[a_{\text{ord}}])$, for $c = 1, \ldots, l$, $\text{id}(t_i[a_{\text{ord}}])[c] = \text{id}(t_i[a_{\text{imp}}])[c] = \text{id}(t_j[a_{\text{ord}}])[c] = \text{id}(t_j[a_{\text{imp}}])[c]$.

Thus, $\text{id}(t_i[a_{\text{imp}}]) = \text{id}(t_j[a_{\text{imp}}])$.

□

Condition 2 in Lemma 7.1 handles the opposite scenario. Whereas condition 1 shows that truncating each Dewey ID in a sequence to a fixed-length prefix preserves the order of the sequence, condition 2 shows that, given a sequence of fixed length Dewey IDs, appending a suffix to each of these IDs preserves the order. For this to work, the suffix appended to each ID has to be uniquely determined by the ID in the original sequence. Expressed in terms of nodes in the collection, there needs to be a unique mapping between the collection nodes matched to a_{ord} (the attribute with known ordering) and the collection

$$[a_{\text{imp}} = RP^{1\to3}_{1.2.9}, a_{\text{ord}} = P^{3\to4}_{1.2}]$$
$$[a_{\text{imp}} = RP^{1\to3}_{1.3.4}, a_{\text{ord}} = P^{3\to4}_{1.3}]$$
$$[a_{\text{imp}} = RP^{1\to3}_{1.3.4}, a_{\text{ord}} = P^{3\to4}_{1.3}]$$
$$[a_{\text{imp}} = RP^{1\to3}_{1.4.8}, a_{\text{ord}} = P^{3\to4}_{1.4}]$$

Figure 7.10: Sequence of tuples R_5, $a_{\text{ord}} \rightsquigarrow a_{\text{imp}}$

nodes matched to a_{imp} (whose ordering is inferred). This requirement is related to the notion of functional dependencies, which are discussed in detail by Arenas et al. [14, 15]. In the case of condition 1, this requirement is implicit, since for any Dewey ID of length greater than l there is exactly one prefix with length l.

Figure 7.10 shows a sequence that illustrates this scenario. As in the previous examples, the sequence is ordered by attribute a_{ord}. For each tuple, the Dewey ID of the node assigned to attribute a_{imp} consists of the Dewey ID of the node assigned to attribute a_{ord} plus an additional suffix. As required, tuples that have the same Dewey ID for the node assigned to attribute a_{ord} receive the same suffix in the Dewey ID of the node assigned to attribute a_{imp}. As can be seen, the sequence is also ordered by a_{imp}.

Proof 7.2 shows that, in the general case, by adding the uniqueness requirement, ordering can be inferred for attributes matching nodes whose ID contains an additional suffix (as is stated in Lemma 7.1, condition 2).

Proof 7.2 (Lemma 7.1, condition 2)**.** Let $a_{\text{ord}}, a_{\text{imp}} \in E$ such that condition 2 holds. Show that for any $t_i, t_j \in R$, $\text{id}(t_i[a_{\text{ord}}]) \leq \text{id}(t_j[a_{\text{ord}}]) \implies \text{id}(t_i[a_{\text{imp}}]) \leq \text{id}(t_j[a_{\text{imp}}])$. Assume the antecedent.

Case 1 $\text{id}(t_i[a_{\text{ord}}]) = \text{id}(t_j[a_{\text{ord}}])$:

By condition 2(c), $\text{id}(t_i[a_{\text{imp}}]) = \text{id}(t_j[a_{\text{imp}}])$.

Case 2 $\mathrm{id}(t_i[a_{\mathrm{ord}}]) < \mathrm{id}(t_j[a_{\mathrm{ord}}])$:

By condition 2(b), $\mathrm{length}(\mathrm{id}(t_i[a_{\mathrm{ord}}])) = \mathrm{length}(\mathrm{id}(t_j[a_{\mathrm{ord}}])) = l$.

Therefore, $\exists w$ with $0 < w \leq l$ such that for $c = 1, \ldots, w-1$, $\mathrm{id}(t_i[a_{\mathrm{ord}}])[c] = \mathrm{id}(t_j[a_{\mathrm{ord}}])[c]$ and $\mathrm{id}(t_i[a_{\mathrm{ord}}])[w] < \mathrm{id}(t_j[a_{\mathrm{ord}}])[w]$.

By conditions 2(a) and 2(b), for $c = 1, \ldots, l$, $\mathrm{id}(t_i[a_{\mathrm{imp}}])[c] = \mathrm{id}(t_i[a_{\mathrm{ord}}])[c]$ and $\mathrm{id}(t_j[a_{\mathrm{imp}}])[c] = \mathrm{id}(t_j[a_{\mathrm{ord}}])[c]$.

Thus, for $c = 1, \ldots, w-1$, $\mathrm{id}(t_i[a_{\mathrm{imp}}])[c] = \mathrm{id}(t_j[a_{\mathrm{imp}}])[c]$ and $\mathrm{id}(t_i[a_{\mathrm{imp}}])[w] < \mathrm{id}(t_j[a_{\mathrm{imp}}])[w]$.

Therefore, $\mathrm{id}(t_i[a_{\mathrm{imp}}]) < \mathrm{id}(t_j[a_{\mathrm{imp}}])$.

□

Building on Lemma 7.1, it is possible to specify how order implications can be detected and, in turn, how the order properties of a physical LQP can be inferred. This is done based on the local QTP corresponding to the physical LQP.

To reason about order implications in a local QTP, it is first necessary to define order implications among pattern nodes. This directly follows the definition of order implications among attributes of a physical LQP. As can be seen below, care has to be taken when reasoning about pattern nodes that are not extraction points. For these pattern nodes there is no corresponding attribute in the physical LQP. Thus, a modified pattern (denoted as $q_k^{u\prime}$) is considered, in which the pattern nodes of interest are designated as extraction points.

Definition 7.7. Let $q_k^u = \langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$ be a local QTP and let $n_{\mathrm{ord}}, n_{\mathrm{imp}} \in N$ be pattern nodes in q_k^u. Further let $q_k^{u\prime} = \langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \cup \{n_{\mathrm{ord}}, n_{\mathrm{imp}}\} \rangle$. Then the order of n_{ord} *implies* the order of n_{imp} (denoted as $n_{\mathrm{ord}} \rightsquigarrow n_{\mathrm{imp}}$) if for any physical LQP ${}^a p_k^{u\prime}$ corresponding to $q_k^{u\prime}$, $a_{\mathrm{ord}} \rightsquigarrow a_{\mathrm{imp}}$, where a_{ord} is the attribute corresponding to extraction point n_{ord} and a_{imp} is the attribute corresponding to extraction point n_{imp}. ■

Based on Definition 7.7, it is possible to detect order implications by inspecting the local QTP corresponding to an LQP. For example if pattern node n_{imp} occurs as a child of pattern node n_{ord} then for any tuple returned by the LQP, the Dewey ID of the node assigned to attribute a_{ord} is a prefix of the Dewey ID of the node assigned to attribute a_{imp}. Additionally, by inspecting the schema, it is possible to infer that nodes matching a given node test only occur at a certain depth in the collection and therefore have Dewey IDs of fixed length. Lemma 7.2 exploits this and formalizes how order dependencies between pattern nodes can be inferred from local QTP and schema.

Lemma 7.2. Let $q^u_k = \langle N, L, r, E, \nu, c, \varepsilon, \lambda, T \rangle$ be a local QTP corresponding to fragment f. Then $n_{\text{ord}} \rightsquigarrow n_{\text{imp}}$ if $\nu(n_{\text{ord}}) \neq *$, $\nu(n_{\text{imp}}) \neq *$, and one of the following conditions holds:

1. (a) n_{imp} occurs as an ancestor of n_{ord} in q^u_k, and

   (b) any path in the schema of f from a root (i.e., the root node type ρ or a node type reachable via an incoming edge from another fragment) to the node type $\nu(n_{\text{imp}})$ has the same number of steps, or

2. (a) n_{imp} occurs as a descendant of n_{ord} in q^u_k,

   (b) any path in the schema of f from a root (i.e., the root node type ρ or a node type reachable via an incoming edge from another fragment) to the node type $\nu(n_{\text{ord}})$ must have the same number of steps, and

   (c) there is exactly one path in the schema from node type $\nu(n_{\text{ord}})$ to node type $\nu(n_{\text{imp}})$ and this path consists solely of edges with the cardinality ONCE or OPT.

■

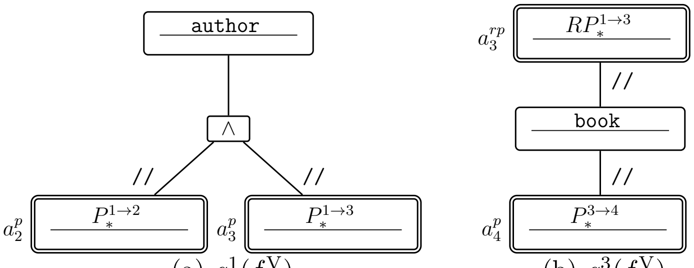
Proof 7.3 (Lemma 7.2)**.** Condition 1 in Lemma 7.2 corresponds directly to condition 1 in Lemma 7.1. If n_{imp} occurs as an ancestor of n_{ord} (condition 1(a)), then within the same pattern match the ID of the collection node matched to n_{imp} will be a prefix of the ID of the collection node matched to n_{ord}. This is because the XQ query model only supports downward axes. If any path from a root to a collection node matched to n_{imp} has the same length (condition 1(b)), then the depth at which collection nodes matched to n_{imp} occur is fixed.

Similarly, conditions 2(a) and 2(b) in Lemma 7.2 correspond to conditions 2(a) and 2(b) in Lemma 7.1, respectively. By requiring a unique path in the schema from $\nu(n_{\text{ord}})$ to $\nu(n_{\text{imp}})$ without any node types with multiple occurrences (condition 2(c)), it is ensured that for each collection node matched to n_{ord}, there is at most one collection node matched to n_{imp}. □

Once an order implication $n_{\text{ord}} \rightsquigarrow n_{\text{imp}}$ has been found for a local QTP q_k^u, Definition 7.7 makes it possible to immediately conclude that for any physical LQP ${}^{a}p_k^u$ corresponding to this local QTP, $a_{\text{ord}} \rightsquigarrow a_{\text{imp}}$. By repeatedly applying this inference, this makes it possible to infer the entire set of order properties for a given physical LQP. For a given physical LQP ${}^{a}p_k^u$ with ordering attribute a, the set of order properties encompasses all those attributes in $A\left(p_k^u\right)$ whose order can be (directly or indirectly) inferred from a:

$$O\left({}^{a}p_k^u\right) = \{a\} \cup \left\{a' \in A\left(p_k^u\right) \mid a \rightsquigarrow a'\right\}$$

For example, consider the local QTP q_8^1, shown in Figure 7.11(a). Assuming that the physical LQP ${}^{a_2^p}p_8^1$ is chosen, the result is explicitly ordered by a_2^p. Inspecting the schema (shown in Figure 7.12) reveals that the order of the `author` node is implied by the order of a_2^p, since the `author` node occurs as an ancestor of a_2^p in the pattern and at a fixed depth in the schema of fragment f_1^{V}. The order of the `author` node then in turn implies the order of a_3^p since there is a unique path in the schema from the `author` node type to the edge to fragment f_3^{V}. Thus, $a_3^p \in O\left({}^{a_2^p}p_8^1\right)$.



(a) $q_8^1(f_1^{\text{V}})$ (b) $q_8^3(f_3^{\text{V}})$

Figure 7.11: Two local QTPs

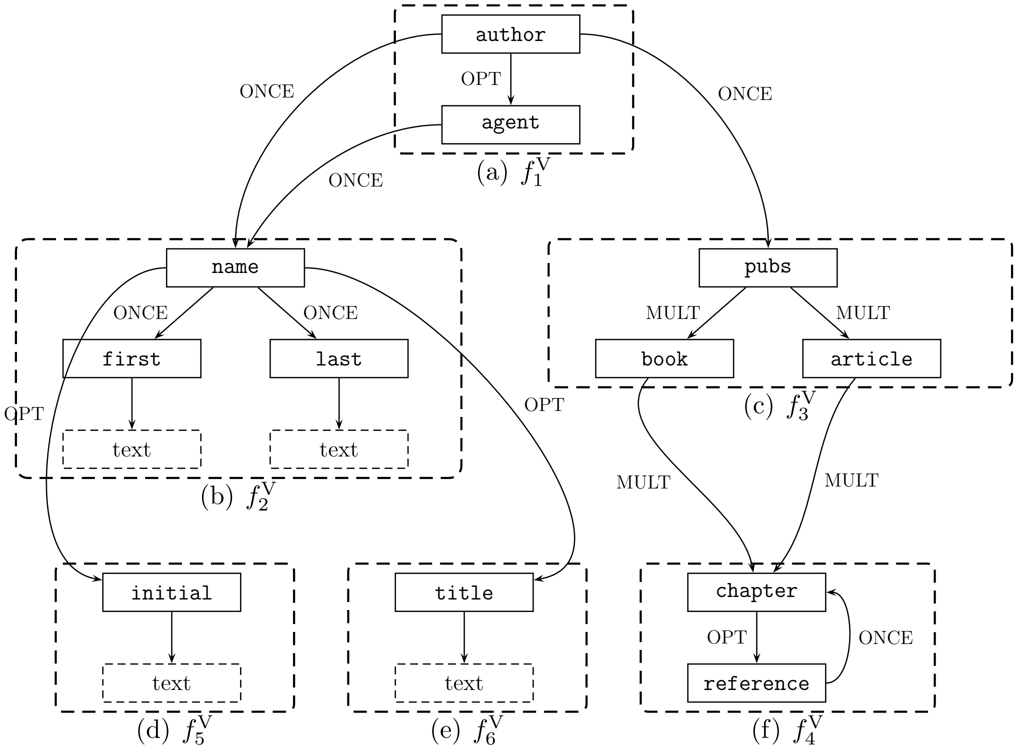


Figure 7.12: A vertical fragmentation schema

When starting with ${}^{a_3^p}p_8^1$, explicitly ordering by a_3^p, it can again be inferred that `author` is also ordered. However, in this case, it is not possible to infer the ordering of a_2^p since there is no unique path from `author` to an edge to fragment f_2^V. Therefore, $a_2^p \notin O\left({}^{a_3^p}p_8^1\right)$.

One interesting property of order implications as defined here is that the order of the attribute corresponding to the root proxy extraction point in a local QTP is implied by any other attribute. Thus, regardless of which attribute is chosen as the ordering attribute, the attribute corresponding the root proxy extraction point is always part of the order properties. This is because a root proxy extraction point always matches nodes at the root of a sub-tree, which means that these matching nodes occur at a fixed depth. In addition, the root proxy extraction point is an ancestor of any pattern node. Thus, for example, for the local QTP q_8^3, shown in Figure 7.11(b), $a_3^{rp} \in O\left({}^{a_4^p}p_8^3\right)$.

Being able to infer the ordering of the root proxy extraction point is highly advantageous. For DEPs containing only “linear” local QTPs (i.e., local QTPs with a single root proxy extraction point and a single proxy extraction point), this makes it possible to use a highly efficient merge join for all cross-fragment joins. Even in cases of more complex local QTPs, it is common that the ordering of all extraction points can be inferred, allowing for the use of a merge join even for these queries.

7.3.2.3 Comparing Alternative Physical LQPs

For a given logical LQP, it is not generally possible to identify a single, best physical LQP corresponding to it. This is because, when comparing the physical LQPs corresponding to a local QTP, multiple properties must be taken into account. For example, while one physical LQP might lead to the lowest total cost, another physical LQP might have a lower cost to first tuple, or a larger set of order properties. Only the distributed optimizer at the query dispatcher can decide which of these physical LQPs leads to the best overall DEP.

This section describes how the local optimizer at a given fragment can compare the possible physical LQPs for a given local QTP. Based on this, the local optimizer can immediately discard all physical LQPs whose performance is dominated by that of another physical LQP (by being worse in at least one of the properties and no better in any of them). The remaining physical LQPs (the “best LQPs”) that are not dominated by another LQP are then considered for inclusion in the physical DEP and, thus, their properties are reported to the distributed optimizer at the query dispatcher.

To determine the best physical LQPs for a local QTP q_k^u, each attribute in the result of p_k^u is considered as the ordering attribute. Thus for each $a \in A\left(p_k^u\right)$, the physical LQP ${}^{a}p_k^u$ with the lowest response time cost(${}^{a}p_k^u$) is determined by the local query optimizer. Additionally, the physical LQP with the lowest cost to first tuple cost-first(${}^{a}p_k^{u\prime}$) and the physical LQP with the lowest cost per sub-tree (subtcost(${}^{a}p_k^u$)) are determined. Together, this results in a set of physical LQPs corresponding to local QTP q_k^u. For each physical LQP ${}^{a}p_k^u$ in this set, the set of order properties ($O\left({}^{a}p_k^u\right)$) is then determined using the techniques presented in the previous section.

Next, the physical LQPs are compared based on their properties. Inferior physical LQPs (i.e., physical LQPs whose performance is dominated by another, better physical

LQP) are discarded. The following definition formalizes how this is done.

Definition 7.8. Let ${}^{a_1}p^u_k$ and ${}^{a_2}p^u_k$ be two physical LQPs corresponding to the same logical LQP. Then ${}^{a_1}p^u_k$ is *inferior* to ${}^{a_2}p^u_k$ if all of the following hold:

- $\text{cost}({}^{a_1}p^u_k) \geq \text{cost}({}^{a_2}p^u_k)$, and
- $\text{cost-first}({}^{a_1}p^u_k) \geq \text{cost-first}({}^{a_2}p^u_k)$, and
- $\text{subtcost}({}^{a_1}p^u_k) \geq \text{subtcost}({}^{a_2}p^u_k)$, and
- $O\left({}^{a_1}p^u_k\right) \subseteq O\left({}^{a_2}p^u_k\right)$.

and at least one of the following holds:

- $\text{cost}({}^{a_1}p^u_k) > \text{cost}({}^{a_2}p^u_k)$, and
- $\text{cost-first}({}^{a_1}p^u_k) > \text{cost-first}({}^{a_2}p^u_k)$, and
- $\text{subtcost}({}^{a_1}p^u_k) > \text{subtcost}({}^{a_2}p^u_k)$, and
- $O\left({}^{a_1}p^u_k\right) \subset O\left({}^{a_2}p^u_k\right)$.

■

Logical Properties	
card(p^u_k)	estimated by centralized cardinality model
$A\left(p^u_k\right)$	set of extraction points in local QTP
nsubt(p^u_k)	fragment statistics
Physical Properties	
cost(${}^{a}p^u_k$)	estimated by centralized cost model
cost-first(${}^{a}p^u_k$)	estimated by centralized cost model
$O\left({}^{a}p^u_k\right)$	inferred from a
subtcost(${}^{a}p^u_k$)	estimated by centralized cost model

Table 7.1: LQP properties

After all of the inferior physical LQPs have been eliminated, the properties of the remaining physical LQPs are reported to the distributed optimizer at the query dispatcher. The distributed optimizer then uses these properties to construct the best physical DEP.

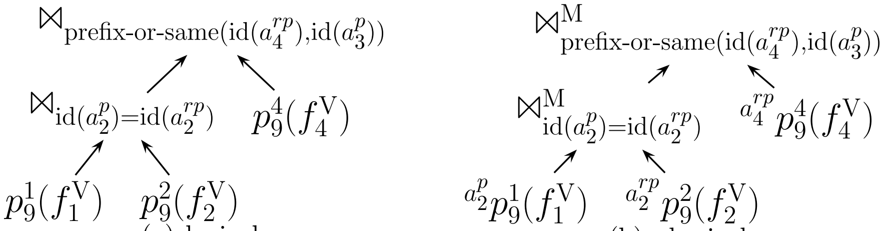
Table 7.1 shows an overview of how the logical and physical properties of LQPs are obtained.

7.4 Obtaining DEP Properties

After the properties of the physical LQPs have been received by the query dispatcher, distributed query optimization can begin. During this phase, the candidate DEPs for a given query are enumerated. The candidate DEPs differ in the order in which the results of the individual LQPs are combined and in the logical and physical operators used to do this.

To determine how the various DEP alternatives compare, the distributed optimizer needs to be able to infer the properties defined in Section 7.2 for a given candidate DEP. As described there, for logical DEPs, two logical properties are determined: the cardinality of the result of the DEP and the set of attributes of the tuples contained in this result. For physical DEPs, total response time cost, time to first tuple and order properties are inferred.

To determine these properties, the candidate DEP is traversed in a bottom up fashion such that each sub-plan visited is rooted at one of the operators in the DEP. This section describes how the properties of each of these sub-plans can be inferred based on the operator


$\bowtie_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^{p}))}$
$\bowtie_{\text{id}(a_2^{p})=\text{id}(a_2^{rp})}$
$p_9^4(f_4^{\text{V}})$
$p_9^1(f_1^{\text{V}})$
$p_9^2(f_2^{\text{V}})$
$\bowtie^{\text{M}}_{\text{prefix-or-same}(\text{id}(a_4^{rp}),\text{id}(a_3^{p}))}$
$\bowtie^{\text{M}}_{\text{id}(a_2^{p})=\text{id}(a_2^{rp})}$
$^{a_4^{rp}}p_9^4(f_4^{\text{V}})$
$^{a_2^{p}}p_9^1(f_1^{\text{V}})$
$^{a_2^{rp}}p_9^2(f_2^{\text{V}})$

(a) logical (b) physical

Figure 7.13: DEPs for query q_9

Logical		Physical	
LQP	A	LQP	cost
p_9^1	$\{a_2^p, a_3^p\}$	${}^{a_2^p}p_9^1$	10
p_9^2	$\{a_2^{rp}\}$	${}^{a_2^{rp}}p_9^2$	30
p_9^4	$\{a_4^{rp}, a_1^e\}$	${}^{a_4^{rp}}p_9^4$	20

Table 7.2: Properties of LQPs for query q_9

at the root of the sub-plan and the properties of the inputs to this operator. Once all sub-plans of a DEP have been visited, the properties of the overall DEP have been determined, including its estimated response time cost, which can be used to compare this DEP to other candidate DEPs.

To illustrate how DEP properties are obtained, consider the logical DEP shown in Figure 7.13(a). Table 7.2 shows the sets of attributes (A) returned by each of the three logical LQPs contained in this logical DEP. To infer the set of attributes returned by the overall logical DEP, the logical DEP is traversed bottom-up. First, the sub-plan rooted at the join between p_9^1 and p_9^2 is visited. To determine the attributes returned by this plan the set of attributes returned by each of the inputs to the join operator is examined. Assuming that the attributes used in the join predicate (a_2^p and a_2^{rp}) are discarded through an implicit projection, the set of attributes returned by this sub-plan can be determined as follows:

$$A\left(p_9^1 \bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} p_9^2\right) = \left(A(p_9^1) \cup A(p_9^2)\right) \setminus \{a_2^p, a_2^{rp}\} = \{a_3^p\}$$

Now, the next sub-plan is visited. In this case, this corresponds to the plan rooted at the remaining join operator, and thus to the entire logical DEP. To determine the set of attributes of the entire logical plan, the same strategy is employed:

$$\begin{aligned} A\left(\left(p_9^1 \bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} p_9^2\right) \bowtie_{\text{prefix-or-same}(\mathrm{id}(a_4^{rp}),\mathrm{id}(a_3^p))} p_9^4\right) = \\ \left(A\left(p_9^1 \bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} p_9^2\right) \cup A(p_9^4)\right) \setminus \{a_3^p, a_4^{rp}\} = \{a_1^e\} \end{aligned}$$

Figure 7.13(b) shows an example of a physical DEP corresponding to the logical DEP in Figure 7.13(a). To determine the overall cost of this physical DEP, the same bottom-up traversal can be employed. To determine the cost of the sub-plan rooted at the join between the physical LQPs ${}^{a_2^p}p_9^1$ and ${}^{a_2^{rp}}p_9^2$, the costs of these physical LQPs (shown in Table 7.2) are examined. Since the physical join operator used to join these LQPs is a merge join (denoted as $\bowtie^{\mathrm{M}}$), the cost of this sub-plan is simply the maximum of the costs of the inputs to the join:

$$\mathrm{cost}\left({}^{a_2^p}p_9^1 \bowtie^{\mathrm{M}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} {}^{a_2^{rp}}p_9^2\right) = \max\left\{\mathrm{cost}({}^{a_2^p}p_9^1), \mathrm{cost}({}^{a_2^{rp}}p_9^2)\right\} = 30$$

This is then repeated for the next sub-plan (i.e., the entire physical DEP):

$$\begin{aligned}\mathrm{cost}\left(\left({}^{a_2^p}p_9^1 \bowtie^{\mathrm{M}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} {}^{a_2^{rp}}p_9^2\right) \bowtie^{\mathrm{M}}_{\text{prefix-or-same}(\mathrm{id}(a_4^{rp}),\mathrm{id}(a_3^p))} {}^{a_4^{rp}}p_9^4\right) = \\ \max\left\{\mathrm{cost}\left({}^{a_2^p}p_9^1 \bowtie^{\mathrm{M}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} {}^{a_2^{rp}}p_9^2\right), \mathrm{cost}({}^{a_4^{rp}}p_9^4)\right\} = 30\end{aligned}$$

The reminder of this section describes how the properties of logical and physical sub-plans can be determined, for each of the operators used in DEPs. For each logical operator, formulas for the logical properties are given. Then the different physical implementations of this logical operator are listed, and for each of these implementations, formulas for inferring physical properties are proposed.

7.4.1 Merge Operator

Logical merge operators (denoted as $\odot$) are encountered in DEPs for horizontally fragmented collections and in DEPs for vertical fragmentation if a QTP has been split due to disjunction or negation.

The logical properties of a merge operator can easily be inferred from the logical properties of the operands of the merge. Since a merge operator returns exactly those tuples that it receives from its operands and the partitions are disjoint, the cardinality of the output of a merge is simply the sum of the cardinalities of its operands:

$$\operatorname{card}\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\left(G_{P_x}\right)\right)=\sum_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\left(\operatorname{card}\left(G_{P_x}\right)\right)$$

Merge operators, as used in this work, assume that all operands produce tuples with the same attributes. The output of the merge, in turn, consists of the same attributes:

$$A\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\right)=A\left(G_{P_u}\right)=A\left(G_{P_v}\right)=\ldots$$

As described in Section 6.1.2, there are four physical implementations of the logical merge operator that differ in how their results are ordered: Merging with full interleaving ($\odot^{\mathrm{FI}}$) returns tuples as soon as they are received. Merging with document-wise interleaving ($\odot^{\mathrm{DI}}$) returns the tuples from a given document once all tuples from that document have been received. Concatenation-based merging ($\odot^{\mathrm{C}}$) returns the tuples from a given operand once all tuples from that operand have been received. Merging with stable concatenation ($\odot^{\mathrm{SC}}$) also returns tuples one operand at a time but additionally enforces a stable order across operands.

While response time cost is a physical property, for merge operators it is independent of the specific implementation. Since none of the merge implementations described in Section 6.1.2 perform processing beyond buffering and passing on tuples received, the overall cost of a merge operator is directly determined by the cost of its most expensive operand[3]. Thus the cost of a merge operator can be determined using the following formula, regardless of which physical implementation of this operator is used (note that $\odot^*$ denotes any physical merge operator):

[3] In cases where all operands have approximately the same cost, a merge operator might incur a small overhead above the maximum operand cost. A similar case might happen with an ordered merge operator (such as $\odot^{\mathrm{SC}}$) when the sub-plan whose tuples need to be returned first is the slowest. This would make it necessary to emit the tuples from all other sub-plans once the slowest sub-plan is finished. Since this overhead is expected to be small, it is not considered in this cost model.

$$\text{cost}\left(\bigodot^{*}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)\approx\max_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\left(\text{cost}\left(G_{P_x}\right)\right)$$

While all physical merge operators yield the same overall cost, time to first tuple and order properties vary depending on which physical operator is chosen. In the following, formulas for determining these physical properties are provided for each of the four physical merge operators.

7.4.1.1 Physical Merge Operator With Full Interleaving

The physical merge operator with full interleaving (denoted as $\odot^{\text{FI}}$) returns its first result tuple as soon as one tuple has been received from any operand. Thus, the overall time to the first tuple is the minimum of the first-tuple delays of the operands of the merge operator:

$$\text{cost-first}\left(\bigodot^{\text{FI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)\approx\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\left(\text{cost-first}\left(G_{P_x}\right)\right)$$

Since the merge operator with full interleaving arbitrarily interleaves the sequences of tuples received from its operands, any order properties present in these sequences are potentially destroyed. Thus, no order properties can be inferred for this operator (i.e., the order properties are the empty set):

$$O\left(\bigodot^{\text{FI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)=\emptyset$$

7.4.1.2 Physical Merge Operator With Document-Wise Interleaving

As described in Section 6.1.2, the physical merge operator with document-wise interleaving (denoted as $\odot^{\text{DI}}$) is used in the context of a horizontal fragmentation, when relaxed order semantics (i.e., ignoring the requirement that there be a stable order of the documents in a collection) are traded off for potentially improved performance. When merging with document-wise interleaving, the first tuple is returned once one of the operands of the merge operator has produced all the tuples derived from one of the documents in the collection.

Due to these semantics, the merge operator with document-wise interleaving is never used when the horizontal fragmentation step corresponding to this operator is nested within a vertical fragmentation. Thus, the horizontal fragmentation step is known to be the outermost fragmentation step and for each operand G_{P_u}, there are two scenarios to consider: Either G_{P_u} consists of a single LQP p_k^u (i.e., there are no further fragmentation steps nested within this horizontal fragmentation) or G_{P_u} consists of multiple LQPs (corresponding to a scenario where a vertical fragmentation step is nested within this horizontal fragmentation). In the latter case, there must be some LQP in $p_k^u \in P_u$ (where P_u denotes the set of LQPs accessed by G_{P_u}) that is the root LQP of G_{P_u}. In both cases, the number of documents accessed by G_{P_u} corresponds to the number of sub-trees accessed by p_k^u, $\mathrm{nsubt}(p_k^u)$. Since, overall, G_{P_u} returns $\mathrm{card}(G_{P_u})$ tuples, the (average) number of tuples per document accessed by G_{P_u} can be estimated as $\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)}$.

Now, consider the time that elapses until G_{P_u} produces $\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)}$ tuples. To estimate this, two components need to be considered. The first component is the time to the first tuple returned by G_{P_u}. This is estimated as $\text{cost-first}\,(G_{P_u})$. The second component uses the assumption that after the first tuple, tuples are returned at a steady rate (cf. Section 7.1). After the first tuple, an additional $\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)} - 1$ tuples are needed. The time to produce these tuples can be estimated as $\left(\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)} - 1\right) \frac{\mathrm{cost}(G_{P_u}) - \text{cost-first}(G_{P_u})}{\mathrm{card}(G_{P_u}) - 1}$. However, it is necessary to consider the case that $\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)} \leq 1$. In this case, it is only necessary to wait for the first tuple produced by G_{P_u}. Based on this, the time required until G_{P_u} returns all tuples for one document (denoted as $\mathrm{docdelay}(G_{P_u})$) can be estimated as follows:

$$\mathrm{docdelay}(G_{P_u}) \approx \text{cost-first}\,(G_{P_u}) + \max\left\{\left(\frac{\mathrm{card}(G_{P_u})}{\mathrm{nsubt}(p_k^u)} - 1\right) \frac{\mathrm{cost}\,(G_{P_u}) - \text{cost-first}\,(G_{P_u})}{\mathrm{card}(G_{P_u}) - 1}, 0\right\}$$

The merge operator with document-wise interleaving produces its first tuple once it has received all tuples corresponding to one document, regardless of which operand has yielded these tuples. Thus, the overall time to first tuple for this operator can be estimated as follows:

$$\text{cost-first}\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{DI}}(G_{P_x})\right)\approx\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{docdelay}(G_{P_x}))$$

As with full interleaving, document-wise interleaving potentially destroys the ordering of its inputs. It does, however preserve any ordering present in all inputs within the context of a single document/sub-tree, which makes this strategy appealing when evaluating queries over horizontally fragmented collections if one is willing to somewhat relax the semantics of order between nodes in different documents. This trade-off is described in more detail in Section 6.1.2. Assuming relaxed order semantics, the order properties of the operator can be described as follows:

$$O\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{DI}}(G_{P_x})\right)=\bigcap_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}O\left(G_{P_x}\right)$$

7.4.1.3 Physical Merge Operator Based on Concatenation

The concatenation-based physical merge operator (denoted as $\odot^{\text{C}}$) returns the first tuple once all tuples have been received from at least one operand. Thus, the delay to the first tuple corresponds to the total response time of the least expensive operand:

$$\text{cost-first}\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{C}}(G_{P_x})\right)\approx\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{cost}\left(G_{P_x}\right))$$

Since concatenation may change the relative order of tuples derived from different subplans, none of the order properties present in the inputs are preserved. Thus, as in the case of the merge operator with full interleaving, no order properties can be inferred for this operator:

$$O\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{C}}\left(G_{P_x}\right)\right)=\emptyset$$

7.4.1.4 Physical Merge Operator Based on Stable Concatenation

With the physical merge operator based on stable concatenation (denoted as $\odot^{\text{SC}}$), the first tuple is returned once all tuples have been received from the leftmost operand. Thus, the time to first tuple can be estimated as the cost of this operand:

$$\text{cost-first}\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{SC}}\left(G_{P_x}\right)\right)\approx\text{cost}\left(G_{P_u}\right)$$

This is the only strategy that preserves order properties. However, this is only the case for order properties present in all of the operands of the merge operator and only if the horizontal fragmentation is not nested within a vertical fragmentation:

$$O\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}^{\text{SC}}\left(G_{P_x}\right)\right)=\bigcap_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}O\left(G_{P_x}\right)$$

Table 7.3 shows an overview of the logical and physical properties of the merge operator.

7.4.2 Cross-Fragment Join Operator

Cross-fragment joins are used to combine the results of LQPs evaluated over vertical fragments. Based on the independence assumption stated in Section 7.1, the cardinality of a cross-fragment join can be estimated. How this is done depends on whether the cross-fragment join has an equality predicate (e.g., $G_{P_u} \bowtie_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}$, as seen in cases without pipelining) or a prefix predicate (e.g., $G_{P_u} \bowtie_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} G_{P_v}$, as seen in cases where pipelining is used).

Logical Properties		
$\text{card}\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$=$	$\sum_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{card}\,(G_{P_x}))$
$A\left(\bigodot_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}\right)$	$=$	$A\,(G_{P_u}) = A\,(G_{P_v}) = \ldots$
Physical Properties – All		
$\text{cost}\left(\bigodot^{*}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$\approx$	$\max_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{cost}\,(G_{P_x}))$
Physical Properties – Full Interleaving		
$\text{cost-first}\left(\bigodot^{\text{FI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$\approx$	$\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{cost-first}\,(G_{P_x}))$
$O\left(\bigodot^{\text{FI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$=$	$\emptyset$
Physical Properties – Document-Wise Interleaving		
$\text{cost-first}\left(\bigodot^{\text{DI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$\approx$	$\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{docdelay}(G_{P_x}))$
$O\left(\bigodot^{\text{DI}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$=$	$\bigcap_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}} O\,(G_{P_x})$ (with relaxed semantics)
Physical Properties – Concatenation		
$\text{cost-first}\left(\bigodot^{\text{C}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$\approx$	$\min_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(\text{cost}\,(G_{P_x}))$
$O\left(\bigodot^{\text{C}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$=$	$\emptyset$
Physical Properties – Stable Concatenation		
$\text{cost-first}\left(\bigodot^{\text{SC}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$\approx$	$\text{cost}\,(G_{P_u})$
$O\left(\bigodot^{\text{SC}}_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}}(G_{P_x})\right)$	$=$	$\bigcap_{G_{P_x}\in\{G_{P_u},G_{P_v},\ldots\}} O\,(G_{P_x})$ (for top-level horizontal fragmentation)

Table 7.3: Merge operator properties

Cardinality of cross-fragment join with equality predicate Let $p_k^v \in P_v$ be the LQP containing the root proxy pattern node used in the join predicate. Since p_k^v accesses $\text{nsubt}(p_k^v)$ sub-trees in its corresponding fragment, the average number of tuples returned by G_{P_v} for each such sub-tree can be estimated as $\frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^v)}$. G_{P_u}, on the other hand, returns $\text{card}(G_{P_u})$ tuples. Since the cross-fragment join has an equality predicate, no intermediate fragments have been skipped. Thus, each proxy node $P_b^{i\to j}$ matched to a_v^p corresponds to exactly one sub-tree accessed by p_k^v. Therefore, the overall cardinality of the cross-fragment join can be estimated as follows:

$$\text{card}\left(G_{P_u} \ltimes_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right) \approx \text{card}(G_{P_u}) \frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^v)}$$

Cardinality of cross-fragment join with prefix predicate When fragments have been pruned from a DEP using the technique described in Section 6.2.1, prefix-based join predicates are used, resulting in a slightly more complicated situation. In this case, for each proxy node $P_a^{g\to i}$ matched to a_u^p by G_{P_u}, there may be multiple corresponding root proxy nodes $RP_b^{i\to j}$, each of which corresponds to a sub-tree that is considered by p_k^v. Assume, for example, that the proxy node $P_a^{g\to i}$ has the Dewey ID 1.2. Then any root proxy node $RP_b^{i\to j}$ with a Dewey ID with the prefix 1.2 (e.g., 1.2.1.1, 1.2.1.2, etc.) is matched by the cross-fragment join.

To address this, it is necessary to insert an additional scaling factor into the cardinality estimation formula. This factor needs to capture how many sub-trees are considered by p_k^v for each proxy node $P_a^{i\to j}$.

$$\text{factor} = \frac{\#\text{ of } P_a^{g\to i} \text{ in } f_g^{\text{V}}}{\#\text{ of } RP_b^{i\to j} \text{ in } f_j^{\text{V}}}$$

Due to the one-to-one correspondence between proxy nodes and root proxy nodes, the number of proxy nodes $P_a^{g\to i}$ in f_g^{V} is the same as the number of root proxy nodes $RP_a^{g\to i}$ in f_i^{V}. This number corresponds to $\text{nsubt}(p_k^u)$ where p_k^u is the skipped LQP corresponding to fragment f_i^{V}. Similarly, the number of root proxy nodes $RP_b^{i\to j}$ in f_j^{V} is $\text{nsubt}(p_k^v)$. Thus, the factor can be determined as follows:

$$\text{factor} = \frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}$$

Including this factor in the formula for estimating the cardinality of the cross-fragment join yields the following estimate:

$$\begin{aligned}\text{card}\left(G_{P_u} \bowtie_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} G_{P_v}\right) &\approx \text{card}(G_{P_u})\frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}\frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^v)} \\ &= \text{card}(G_{P_u})\frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^u)}\end{aligned}$$

Attribute set of cross-fragment join To determine the set of attributes, it is assumed that the attributes used in the join predicate are projected away immediately after the cross-fragment join. This leads to the following formula, where θ is the comparison used in the join predicate (i.e., equality or prefix)[4] :

$$A\left(G_{P_u} \bowtie_{\text{id}\left(a_u^p\right)\,\theta\,\text{id}\left(a_v^{rp}\right)} G_{P_v}\right) = \left(A\left(G_{P_u}\right) \cup A\left(G_{P_v}\right)\right) \setminus \{a_u^p, a_v^{rp}\}$$

Depending on the ordering of the operands of the cross-fragment join, several different physical join operators are considered. If both operands are ordered by the attributes used in the join predicate (either because of the order properties of the LQPs in the operand, or because a separate sort operator was inserted), a *merge join* (denoted as $\bowtie^{\text{M}}$) can be used. This physical operator supports full pipelining on both of its operands, allowing for maximum parallelism. Alternatively, a one-sided hash join (denoted as $\bowtie^{\text{H}}$) can be employed. In this case it is necessary to materialize one of the operands before the join can be performed. A symmetric hash join (denoted as $\bowtie^{\text{SH}}$) represents a third alternative. Like the one-sided hash join, this physical operator does not require that its operands be ordered by the attributes used in the join predicate. However, unlike the one-sided hash join, the symmetric hash join produces join tuples as soon as tuples are received from

[4] For simplicity, in the following $\text{id}(a_1)\,\text{prefix-or-same}\,\text{id}(a_2)$ is defined to be equivalent to $\text{prefix-or-same}(\text{id}(a_1), \text{id}(a_2))$.

the operands. This is achieved by using two hash tables and obviates the need to fully materialize one of the operands. Pushing the cross-fragment join into an LQP represents a fourth option. In this case it is possible to use an index join (denoted as $\bowtie^{\mathrm{I}}$).

7.4.2.1 Physical Merge Join Operator

The merge join operator (denoted as $\bowtie^{\mathrm{M}}$) is a highly efficient physical join operator that relies on both of its operands being ordered by the attributes referenced in the join predicate. Therefore, to be able to use a merge join to evaluate the join $G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}$, the result of G_{P_u} has to be ordered by a_u^p and the result of G_{P_v} has to be ordered by a_v^{rp}. Formally, these requirements can be expressed as $a_u^p \in O\left(G_{P_u}\right)$ and $a_v^{rp} \in O\left(G_{P_v}\right)$.

As described in [25, 105], a merge join works by synchronously iterating over the tuples generated by both operands. This way, it is possible to process the join without materializing either operand, which allows for full pipelining on both operands. Due to this characteristic, the response time of performing a merge join is dominated by the cost of evaluating the operands of the join and the overall cost can be estimated as the maximum operand cost:

$$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \max\left\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\right\}$$

To estimate the time until the first join tuple is produced, consider the average number of tuples from each operand of the join that are needed to produce one join tuple. $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$ denotes the number of tuples needed from operand G_{P_u} so that $G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}$ can produce one join tuple. This quantity can be estimated by dividing the cardinality of the operand by the cardinality of the join result:

$$\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \frac{\mathrm{card}(G_{P_u})}{\mathrm{card}\left(G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)}$$

Now consider the time it takes for G_{P_u} to produce $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$ tuples. This can be estimated similar to how $\mathrm{docdelay}(G_{P_u})$ is estimated in Section

7.4.1.2 by considering the time to the first tuple of G_{P_u} and then assuming that the remaining tupfirst $\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) - 1$ tuples are produced at a steady rate. This yields the following estimate of the time elapsed before G_{P_u} produces tupfirst(G_{P_u}, $G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}$) tuples (denoted as tupdelay $\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$):

$$\begin{aligned}&\text{tupdelay}\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \text{cost-first}(G_{P_u}) + \\ &\quad \max\left\{\frac{\text{cost}(G_{P_u}) - \text{cost-first}(G_{P_u})}{\text{card}(G_{P_u}) - 1}\left(\text{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) - 1\right), 0\right\}\end{aligned}$$

Based on this, the time until the fist join tuple is produced can be estimated as the time until a sufficient number of input tuples have been received from both operands:

$$\begin{aligned}&\text{cost-first}\left(G_{P_u} \ltimes^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \\ &\quad \max\left\{\text{tupdelay}\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right), \text{tupdelay}\left(G_{P_v}, G_{P_u} \ltimes_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)\right\}\end{aligned}$$

To determine the order properties of the physical merge join operator, it is helpful to take a closer look at how this operator proceeds. In particular, it is important to consider what happens when there are duplicate values of the join attribute in one or both of the operands. The merge join handles this by taking a tuple from the outer (left) operand of the join and matching it to each tuple from the inner (right) operand of the join for which the join predicate is satisfied. This results in the tuples from the inner operand being iterated over multiple times if there are duplicate attribute values in the tuples from the outer operand.

Consider, for example, the sub-plans G_{P_u} and G_{P_v}, which yield the tuples shown in Figures 7.14(a) and 7.14(b), respectively. As can be seen, $R(G_{P_u})$ is ordered by a_1^e and a_v^p, and $R(G_{P_v})$ is ordered by a_v^{rp} and a_n^p.

When performing the join $G_{P_u} \ltimes^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}$, G_{P_u} is on the left and thereby the outer side of the join. Thus, for each tuple in $R(G_{P_u})$, the join iterates over the matching tuples in $R(G_{P_v})$. This yields the tuples shown in Figure 7.15(a). As can be seen, for tuple

$t_{a,1}$ with $\mathrm{id}(a_v^p) = 10$, there are two matching tuples ($t_{b,1}$ and $t_{b,2}$) in $R(G_{P_v})$. Therefore, the tuples $t_{a,1} \cdot t_{b,1}$ and $t_{a,1} \cdot t_{b,2}$ are produced (denoting $t_{a,1}$ concatenated with $t_{b,1}$, and $t_{a,1}$ concatenated with $t_{b,2}$, respectively). Since $t_{a,1} \cdot t_{b,1}$ and $t_{a,1} \cdot t_{b,2}$ are produced in the same order as the original tuples in $R(G_{P_v})$ and $R(G_{P_v})$, the join preserves the order properties a_1^e and a_n^p in this case.

For $\mathrm{id}(a_v^p) = \mathrm{id}(a_v^{rp}) = 11$, there are two tuples in $R(G_{P_u})$ ($t_{a,2}$ and $t_{a,3}$) and two tuples in $R(G_{P_v})$ ($t_{b,3}$ and $t_{b,4}$). Since G_{P_u} is the outer input, $t_{a,2}$ is processed first and matched with $t_{b,3}$ and $t_{b,4}$, yielding $t_{a,2} \cdot t_{b,3}$ and $t_{a,2} \cdot t_{b,4}$. Then $t_{a,3}$ is matched with $t_{b,3}$ and $t_{b,4}$, yielding $t_{a,3} \cdot t_{b,3}$ and $t_{a,3} \cdot t_{b,4}$. As can be seen, while this preserves the ordering of the attribute on the outer side of the join (a_1^e), it does not preserve the order of the attribute on the inner side (a_n^p).

For $\mathrm{id}(a_v^p) = \mathrm{id}(a_v^{rp}) = 12$, a third scenario is encountered. This time, there are two tuples on the outer side ($t_{a,4}$ and $t_{a,5}$), but only one tuple on the inner side ($t_{b,5}$). In this case the order of all attributes is preserved, as is shown in the result tuples $t_{a,4} \cdot t_{b,5}$ and $t_{a,5} \cdot t_{b,5}$.

Figure 7.15(b) shows the result of performing the join $G_{P_v} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_u}$. This time, G_{P_v} is the outer input and G_{P_u} is the inner input. As before, for values of $\mathrm{id}(a_v^p)$ and $\mathrm{id}(a_v^{rp})$ where at least one side is duplicate-free, the ordering of all attributes is preserved. For $\mathrm{id}(a_v^p) = \mathrm{id}(a_v^{rp}) = 11$, however, where there are two tuples in $R(G_{P_v})$ and two tuples in $R(G_{P_u})$, only the order of the attributes from the outer side (i.e., G_{P_v}) is preserved.

$$\begin{aligned}
t_{a,1} =& [a_1^e = \mathtt{name}_4, a_v^p = P_{10}^{i \to j}] \\
t_{a,2} =& [a_1^e = \mathtt{name}_5, a_v^p = P_{11}^{i \to j}] \\
t_{a,3} =& [a_1^e = \mathtt{name}_6, a_v^p = P_{11}^{i \to j}] \\
t_{a,4} =& [a_1^e = \mathtt{name}_7, a_v^p = P_{12}^{i \to j}] \\
t_{a,5} =& [a_1^e = \mathtt{name}_8, a_v^p = P_{12}^{i \to j}]
\end{aligned}$$

(a) $R(G_{P_u})$

$$\begin{aligned}
t_{b,1} =& [a_v^{rp} = RP_{10}^{i \to j}, a_n^p = P_{21}^{j \to k}] \\
t_{b,2} =& [a_v^{rp} = RP_{10}^{i \to j}, a_n^p = P_{22}^{j \to k}] \\
t_{b,3} =& [a_v^{rp} = RP_{11}^{i \to j}, a_n^p = P_{23}^{j \to k}] \\
t_{b,4} =& [a_v^{rp} = RP_{11}^{i \to j}, a_n^p = P_{24}^{j \to k}] \\
t_{b,5} =& [a_v^{rp} = RP_{12}^{i \to j}, a_n^p = P_{25}^{j \to k}]
\end{aligned}$$

(b) $R(G_{P_v})$

Figure 7.14: Tuples produced by sub-plans G_{P_u} and G_{P_v}

$$
\begin{aligned}
t_{a,1} \cdot t_{b,1} =& [a_1^e = \mathtt{name}_4, a_n^p = P_{21}^{j \to k}] \\
t_{a,1} \cdot t_{b,2} =& [a_1^e = \mathtt{name}_4, a_n^p = P_{22}^{j \to k}] \\
t_{a,2} \cdot t_{b,3} =& [a_1^e = \mathtt{name}_5, a_n^p = P_{23}^{j \to k}] \\
t_{a,2} \cdot t_{b,4} =& [a_1^e = \mathtt{name}_5, a_n^p = P_{24}^{j \to k}] \\
t_{a,3} \cdot t_{b,3} =& [a_1^e = \mathtt{name}_6, a_n^p = P_{23}^{j \to k}] \\
t_{a,3} \cdot t_{b,4} =& [a_1^e = \mathtt{name}_6, a_n^p = P_{24}^{j \to k}] \\
t_{a,4} \cdot t_{b,5} =& [a_1^e = \mathtt{name}_7, a_n^p = P_{25}^{j \to k}] \\
t_{a,5} \cdot t_{b,5} =& [a_1^e = \mathtt{name}_8, a_n^p = P_{25}^{j \to k}]
\end{aligned}
$$

(a) $G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}$

$$
\begin{aligned}
t_{a,1} \cdot t_{b,1} =& [a_1^e = \mathtt{name}_4, a_n^p = P_{21}^{j \to k}] \\
t_{a,1} \cdot t_{b,2} =& [a_1^e = \mathtt{name}_4, a_n^p = P_{22}^{j \to k}] \\
t_{a,2} \cdot t_{b,3} =& [a_1^e = \mathtt{name}_5, a_n^p = P_{23}^{j \to k}] \\
t_{a,3} \cdot t_{b,3} =& [a_1^e = \mathtt{name}_6, a_n^p = P_{23}^{j \to k}] \\
t_{a,2} \cdot t_{b,4} =& [a_1^e = \mathtt{name}_5, a_n^p = P_{24}^{j \to k}] \\
t_{a,3} \cdot t_{b,4} =& [a_1^e = \mathtt{name}_6, a_n^p = P_{24}^{j \to k}] \\
t_{a,4} \cdot t_{b,5} =& [a_1^e = \mathtt{name}_7, a_n^p = P_{25}^{j \to k}] \\
t_{a,5} \cdot t_{b,5} =& [a_1^e = \mathtt{name}_8, a_n^p = P_{25}^{j \to k}]
\end{aligned}
$$

(b) $G_{P_v} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_u}$

Figure 7.15: Tuples produced by merge joins

As is illustrated in this example, the set of attributes that are ordered in the result of a merge join is the set of attributes that are ordered in the outer (i.e., left-hand side) input of the join.

Additionally, the requirement that G_{P_u} and G_{P_v} be ordered by the join attributes implies that the result of the join is ordered by a_v^{rp}. Hence, the join result is additionally ordered by all those attributes in $A\left(G_{P_v}\right)$ whose order is implied by a_v^{rp}.

If it can be shown that either the result of G_{P_u} or the result of G_{P_v} is free of duplicate values of the join attribute, then the ordering of all attributes is preserved. This is the case, for example, when G_{P_v} consists of a single LQP p_k^v and the QTP corresponding to p_k^v, q_k^v has only a single extraction point.

In summary, the order properties of a merge join can be determined as follows:

$$
O\left(G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) =
\begin{cases}
O\left(G_{P_u}\right) \cup O\left(G_{P_v}\right) & \text{if } a_u^p \text{ or } a_v^{rp} \text{ dupl. free} \\
O\left(G_{P_u}\right) \cup \left\{a_i \in A\left(G_{P_v}\right), a_v^{rp} \rightsquigarrow a_i\right\} & \text{otherwise}
\end{cases}
$$

If at least one of the operands of a cross-fragment join is not ordered by the join predicate, it is not possible to use the merge join operator directly. To address this, it is possible to insert a sort operator, which is discussed further in Section 7.4.3. Alternatively, other physical join operators such as a one-sided or two-sided hash join can be considered.

7.4.2.2 Physical One-Sided Hash Join Operator

A one-sided hash join (denoted as $\bowtie^{\mathrm{H}}$) is a physical join operator that does not require its operands to be ordered in any specific way. There exist many different variants of this physical operator in the literature, for an overview the reader is referred to Graefe's survey [59].

This hash join operator proceeds by populating a hash table with the tuples received from the inner (right) operand. Once this hash table is in place, it is probed for each tuple from the outer (left) operand and for each match found in the hash table an output tuple is produced.

When estimating the cost of the hash join $G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}$, two scenarios need to be considered. If the response time cost of G_{P_u} is sufficiently high, then this cost dominates the overall response time cost of the join. If however, the cost of G_{P_v} dominates, then after all the tuples in the result of G_{P_v} have been inserted into the hash table, the hash table still has to be probed for each tuple in the result of G_{P_u}. The cost of this probing phase depends on the cost of an individual probe operation on the hash table (denoted as probecost, which can easily be determined experimentally) and the number of times this operation is performed (once for each tuple received from G_{P_u}, i.e., $\mathrm{card}(G_{P_u})$ times). Based on this, the overall cost of the hash join can be estimated as follows:

$$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \max\left\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v}) + \mathrm{card}(G_{P_u})\,\mathrm{probecost}\right\}$$

The hash join operator always materializes its inner operand before returning the first tuple and this fact has to be taken into account when estimating the response time elapsed until the first join tuple is produced. Once the hash table is built, it has to be probed with (on average) tupfirst $\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$ tuples from the outer operand. Thus, there are again two cases to consider: If the response time cost of producing the first tupfirst $\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$ input tuples from G_{P_u} dominates, this cost (tupdelay $\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$) dominates the time until the first join tuple is produced. Otherwise, the time to the first tuple corresponds to the time until the hash table is built (cost(G_{P_v})) plus the cost of probing the hash table for tupfirst $\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$ tuples from G_{P_u}. This yields the following estimate:

$$
\begin{aligned}
&\text{cost-first}\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \approx \\
&\qquad \max\left\{ \begin{aligned} &\text{tupdelay}\left(G_{P_u}, G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right), \\ &\text{cost}(G_{P_v}) + \text{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) \text{probecost} \end{aligned} \right\}
\end{aligned}
$$

Since the hash join operator considers one tuple from the outer operand at a time, the order of all attributes from the outer operand is preserved. For the attributes from the inner operand, which are accessed randomly via the hash table, there is no such guarantee. The sole exception to this occurs in the case where G_{P_u} is ordered by the proxy attribute a_u^p. If the hash operator is implemented such that probing the hash table with a proxy ID yields the matching root proxy nodes in document order, it is possible to infer that the join result will be ordered by all those attributes in G_{P_v} whose order is implied by a_v^{rp}. Thus, the order properties of a hash join can be determined as follows:

$$
O\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) = \begin{cases} O\left(G_{P_u}\right) \cup \left\{a_i \in A\left(G_{P_v}\right), a_v^{rp} \rightsquigarrow a_i\right\} & \text{if } a_u^p \in O\left(G_{P_u}\right) \\ O\left(G_{P_u}\right) & \text{otherwise} \end{cases}
$$

7.4.2.3 Physical Symmetric Hash Join Operator

Another alternative that combines some of the benefits of the merge join operator and the one-sided hash join operator is the symmetric hash join operator (denoted as $\bowtie^{\mathrm{SH}}$). This operator, first proposed by Wilschut and Apers [134], relies on two separate hash tables, one for each operand. Whenever an input tuple is received from one of the operands, the hash table of the other operand is probed and output tuples are generated as in the probing phase of the one-sided hash join. Then, the received input tuple is inserted into the hash table corresponding to its operand, where it is available for probing once further tuples from the other operand are received. By using this strategy, the symmetric hash can begin producing join tuples before either operand has been fully materialized, obviating the need for a separate probing phase. Assuming that enough processing capacity is available for the symmetric hash join to keep up with the rate at which input tuples are received, the overall cost of this operator can thus be estimated as follows:

$$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right) \approx \max\left\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\right\}$$

As can be seen, this is the same overall cost as in the case of the merge join operator.

When considering time to first tuple, the characteristics of the symmetric hash join operator are similarly advantageous. Since neither operand needs to be materialized fully, the first join tuple is produced once $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right)$ tuples have been received from G_{P_u} and $\mathrm{tupfirst}\left(G_{P_v}, G_{P_u} \bowtie_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right)$ have been received from G_{P_v}. In analogy to the formula used to estimate the cost the first tuple produced by the merge join operator, the cost to first tuple is thus:

$$\begin{aligned}&\text{cost-first}\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right) \approx \\ &\quad \max\left\{\mathrm{tupdelay}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right), \mathrm{tupdelay}\left(G_{P_v}, G_{P_u} \bowtie_{\mathrm{id}(a^p_u)\,\theta\,\mathrm{id}(a^{rp}_v)} G_{P_v}\right)\right\}\end{aligned}$$

When comparing total cost and cost to first tuple of merge join, one-sided hash join, and symmetric hash join, the latter comes out ahead, especially when considering that, unlike the merge join operator, it does not require its operands to be ordered. However, this flexibility of the symmetric hash join operator comes at a price. For each input tuple received by this operator, some join tuples may be produced as soon as the input tuple is received. Then, additional join tuples may be produced whenever further matching tuples are received from the other operand. This has the effect of destroying any order properties that may have been present in the join operands, which represents a key disadvantage of this operator.

$$O\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right) = \emptyset$$

7.4.2.4 Pushed Cross-Fragment Joins

As discussed in Section 6.2.2.3, for cross-fragment joins that have been pushed into an LQP, it is possible to pre-compute an index over the root proxy nodes to enable efficient retrieval of the relevant sub-trees. Using this index, it is possible to use an index join operator (denoted as $\bowtie^{\mathrm{I}}$), i.e., a hash join for which the hash table is pre-computed. For each proxy ID pipelined into an LQP, the index join looks up the relevant sub-tree in the index, and the remainder LQP (denoted as $\bar{p}_k^v$) is then evaluated over this sub-tree.

When evaluating a pushed cross-fragment join, the remainder LQP $\bar{p}_k^v$ is evaluated for each pipelined tuple from G_{P_u}. Thus, when estimating the cost of this join, there are two scenarios to consider: If the outer (left) operand (G_{P_u}) is the performance limiting factor, then the overall cost of the join is simply the cost of G_{P_u} plus the cost of evaluating $\bar{p}_k^v$ over the last tuple ($\mathrm{subtcost}(p_k^v)$). If the inner (right) operand is slower, the total cost consists of the time spent waiting for G_{P_u}'s first tuple ($\text{cost-first}(G_{P_u})$) plus the response time cost of evaluating $\bar{p}_k^v$ over each tuple returned by G_{P_u} ($\mathrm{card}(G_{P_u})\,\mathrm{subtcost}(p_k^v)$). This yields the following cost estimation formula:

$$\begin{aligned}\mathrm{cost}\left(\bar{p}_k^v\left(G_{P_u} \bowtie^{\mathrm{I}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right) \approx& \\ \max\left\{\mathrm{cost}(G_{P_u}) + \mathrm{subtcost}(p_k^v), \text{cost-first}(G_{P_u}) + \mathrm{card}(G_{P_u})\,\mathrm{subtcost}(p_k^v)\right\}&\end{aligned}$$

When the cross-fragment join is used in combination with a skipped LQP, each tuple pipelined into the LQP might result in the remainder plan being evaluated over multiple sub-trees, all of which have a root proxy ID starting with the same prefix. In this case, it is necessary to consider the additional sub-trees in the cost estimate. Assuming that p_k^u is the first LQP skipped, the number of sub-trees accessed by p_k^v for each sub-tree accessed by p_k^u is $\frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}$ (as explained in Section 7.4.2). Considering this factor in the cost estimate yields the following formula:

$$\begin{aligned}\text{cost}\left(\bar{p}_k^v\left(G_{P_u} \ltimes_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} \text{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right) \approx \\ \max\left\{\text{cost}(G_{P_u}) + \frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}\,\text{subtcost}(p_k^v),\right. \\ \left.\text{cost-first}(G_{P_u}) + \text{card}(G_{P_u})\frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}\,\text{subtcost}(p_k^v)\right\}\end{aligned}$$

To produce the first join tuple, $\text{tupfirst}(G_{P_u}, G_{P_u} \ltimes_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} p_k^v)$ tuples are needed from G_{P_u}. To estimate the time to the first join tuple, two cases need to be considered: If the outer operand (G_{P_u}) is the limiting factor, then the time to the first tuple can be estimated as the time elapsed to get a sufficient number of tuples from G_{P_u} plus the time needed to evaluate the remainder LQP for the last of these tuples. If, on the other hand, the remainder LQP is the limiting factor, then the time to the first join tuple consists of the time spent waiting for the first tuple from G_{P_u}, plus the cost of evaluating the remainder LQP for each of the $\text{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} p_k^v\right)$ tuples from G_{P_u} necessary to produce the first join tuple. Together, this yields the following formula:

$$\begin{aligned}\text{cost-first}\left(\bar{p}_k^v\left(G_{P_u} \ltimes^{\text{I}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} \text{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right) \approx \\ \max\left\{\text{tupdelay}\left(G_{P_u}, G_{P_u} \ltimes_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} p_k^v\right) + \text{subtcost}(p_k^v),\right. \\ \left.\text{cost-first}(G_{P_u}) + \text{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} p_k^v\right)\,\text{subtcost}(p_k^v)\right\}\end{aligned}$$

In the presence of skipped LQPs, it is necessary to consider that for each proxy node contained in a tuple from G_{P_u}, there may be multiple matching sub-trees, over which the remainder LQP needs to be executed. On average, there are $\frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}$ sub-trees for each for each proxy node from G_{P_u}. Thus, the delay to the first join tuple can be estimated by the following formula:

$$\begin{aligned}&\text{cost-first}\left(\bar{p}_k^v\left(G_{P_u} \ltimes_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} \text{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right) \approx \\ &\quad \max\left\{\text{tupdelay}\left(G_{P_u}, G_{P_u} \ltimes_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} p_k^v\right) + \frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}\text{subtcost}(p_k^v),\right. \\ &\left.\text{cost-first}(G_{P_u}) + \text{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\text{prefix-or-same}(\text{id}(a_u^p),\text{id}(a_v^{rp}))} p_k^v\right) \frac{\text{nsubt}(p_k^v)}{\text{nsubt}(p_k^u)}\text{subtcost}(p_k^v)\right\}\end{aligned}$$

The index join operator processes tuples from the outer operand one at a time. Therefore, the order properties of the outer operand are preserved. Since the sub-trees accessed by the remainder LQP $\bar{p}_k^v$ are accessed randomly, in general no ordering can be inferred for the attributes generated by $\bar{p}_k^v$. As in the case of the one-sided hash join operator, there is one exception to this rule. If $R(G_{P_u})$ is ordered by the join attribute a_u^p then the output of the join is ordered by all attributes whose ordering can be inferred from a_v^{rp}. This corresponds directly to $O\left({}^{a_v^{rp}}p_k^v\right)$, i.e., the ordered attributes of ${}^{a_v^{rp}}p_k^v$, the physical LQP with a_v^{rp} designated as the ordering extraction point. Assuming that multiple sub-trees whose root proxy IDs start with the same prefix are accessed in order (which is easily achieved by using an order-preserving index structure), the same holds in the scenario where LQPs are skipped.

$$O\left(\bar{p}_k^v\left(G_{P_u} \ltimes^{\text{I}}_{\text{id}(a_u^p)\,\theta\,\text{id}(a_v^{rp})} \text{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right) = \begin{cases} O\left(G_{P_u}\right) \cup O\left({}^{a_v^{rp}}p_k^v\right) & \text{if } a_u^p \in O\left(G_{P_u}\right) \\ O\left(G_{P_u}\right) & \text{otherwise} \end{cases}$$

7.4.2.5 Example

To illustrate the properties of the various physical join operators, consider the simple, logical DEP shown in Figure 7.16. Assuming that the LQPs used in this logical DEP have the properties shown in Table 7.4, the cardinality of this DEP can be determined as follows:

$$\text{card}\left(p_1^1 \ltimes_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} p_1^2\right) \approx \text{card}(p_1^1)\frac{\text{card}(p_1^2)}{\text{nsubt}(p_1^2)} = 8\frac{10}{5} = 16$$

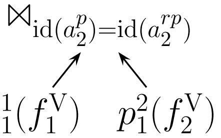

$\bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
$p_1^1(f_1^{\mathrm{V}})$
$p_1^2(f_2^{\mathrm{V}})$

Figure 7.16: A logical DEP with a single cross-fragment join

Logical				Physical			
LQP	card	subt	A	LQP	cost	cost-first	O
p_1^1	8	4	$\{a_2^p, a_1^e\}$	${}^{a_2^p}p_1^1$	16	2	$\{a_2^p, a_1^e\}$
p_1^2	10	5	$\{a_2^{rp}\}$	${}^{a_2^{rp}}p_1^2$	30	3	$\{a_2^{rp}\}$

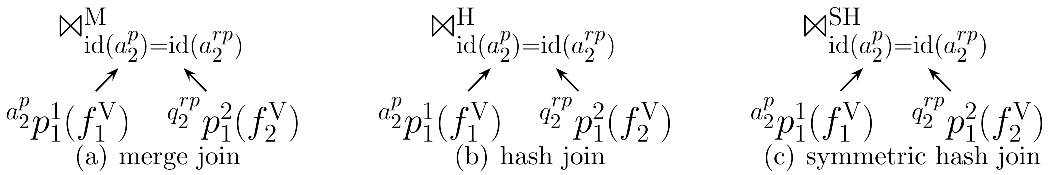
Table 7.4: Properties of LQPs

Similarly, the set of attributes of the tuples returned by this DEP can be determined based on the attributes returned by p_1^1 and p_1^2:

$$A\left(p_9^1 \bowtie_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})} p_9^2\right) = \left(A(p_9^1) \cup A(p_9^2)\right) \setminus \{a_2^p, a_2^{rp}\} = (\{a_2^p, a_1^e\} \cup \{a_2^{rp}\}) \setminus \{a_2^p, a_2^{rp}\} = \{a_1^e\}$$

Figure 7.17 shows three physical DEPs corresponding to this logical DEP that differ in the physical join operator they use. The physical DEP shown in Figure 7.17(a) uses a merge join, the physical DEP in Figure 7.17(b) uses a one-sided hash join, and the physical DEP in Figure 7.17(c) uses a symmetric hash join. Based on the properties of the physical LQPs used in these physical DEPs (shown in Table 7.4), the physical properties of the DEPs can be inferred.

Cost The cost of the DEP with the merge join and the cost of the DEP with the symmetric hash join are both determined by the maximum LQP cost:


$\bowtie^{\mathrm{M}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
${}^{a_2^p}p_1^1(f_1^{\mathrm{V}})$
${}^{q_2^{rp}}p_1^2(f_2^{\mathrm{V}})$
$\bowtie^{\mathrm{H}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
${}^{a_2^p}p_1^1(f_1^{\mathrm{V}})$
${}^{q_2^{rp}}p_1^2(f_2^{\mathrm{V}})$
$\bowtie^{\mathrm{SH}}_{\mathrm{id}(a_2^p)=\mathrm{id}(a_2^{rp})}$
${}^{a_2^p}p_1^1(f_1^{\mathrm{V}})$
${}^{q_2^{rp}}p_1^2(f_2^{\mathrm{V}})$

(a) merge join (b) hash join (c) symmetric hash join

Figure 7.17: Physical DEPs with a single cross-fragment join

$$\begin{aligned}\text{cost}\left({}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{a_2^{rp}}p_1^2\right) \approx \text{cost}\left({}^{a_2^p}p_1^1 \bowtie^{\text{SH}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{a_2^{rp}}p_1^2\right) \approx \\ \max\left\{\text{cost}({}^{a_2^p}p_1^1), \text{cost}({}^{a_2^{rp}}p_1^2)\right\} = \max\{16, 30\} = 30\end{aligned}$$

For the DEP with the one-sided hash join, the cost of the probing phase needs to be taken into account. Assuming that the cost of a single probe operation is 0.2, the cost of this DEP can be estimated as follows:

$$\begin{aligned}&\text{cost}\left({}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{a_2^{rp}}p_1^2\right) \approx \\ &\max\left\{\text{cost}({}^{a_2^p}p_1^1), \text{cost}({}^{a_2^{rp}}p_1^2)\right\} + \text{card}(p_1^1)\,\text{probecost} = \max\{16, 30 + 8 \cdot 0.2\} = 31.6\end{aligned}$$

Cost to first tuple For both merge join and symmetric hash join, cost to first tuple can be determined based on the time that elapses before a sufficient number of tuples have been received by from each operand (denoted as tupdelay()):

$$\begin{aligned}\text{cost-first}\left({}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right) \approx \text{cost-first}\left({}^{a_2^p}p_1^1 \bowtie^{\text{SH}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right) \approx \\ \max\left\{\text{tupdelay}\left({}^{a_2^p}p_1^1, {}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right),\right. \\ \left.\text{tupdelay}\left({}^{a_2^{rp}}p_1^2, {}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right)\right\} =\end{aligned}$$

$$\max\Bigg\{\text{cost-first}({}^{a_2^p}p_1^1)+$$

$$\max\Big\{\frac{\text{cost}({}^{a_2^p}p_1^1)-\text{cost-first}({}^{a_2^p}p_1^1)}{\text{card}({}^{a_2^p}p_1^1)-1}\Big(\text{tupfirst}\Big({}^{a_2^p}p_1^1, {}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\Big)-1\Big),$$

$$0\Big\}, \text{cost-first}({}^{a_2^{rp}}p_1^2)+$$

$$\max\Big\{\frac{\text{cost}({}^{a_2^{rp}}p_1^2)-\text{cost-first}({}^{a_2^{rp}}p_1^2)}{\text{card}({}^{a_2^{rp}}p_1^2)-1}\Big(\text{tupfirst}\Big({}^{a_2^{rp}}p_1^2, {}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\Big)-1\Big),$$

$$0\Big\}\Bigg\} =$$

$$\max\Bigg\{\text{cost-first}({}^{a_2^p}p_1^1)+$$

$$\max\left\{\frac{\text{cost}({}^{a_2^p}p_1^1)-\text{cost-first}({}^{a_2^p}p_1^1)}{\text{card}({}^{a_2^p}p_1^1)-1}\left(\frac{\text{card}(p_1^1)}{\text{card}\left(p_1^1 \bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} p_1^2\right)}-1\right), 0\right\},$$

$$\text{cost-first}({}^{a_2^{rp}}p_1^2)+$$

$$\max\left\{\frac{\text{cost}({}^{a_2^{rp}}p_1^2)-\text{cost-first}({}^{a_2^{rp}}p_1^2)}{\text{card}({}^{a_2^{rp}}p_1^2)-1}\left(\frac{\text{card}(p_1^2)}{\text{card}\left(p_1^1 \bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} p_1^2\right)}-1\right), 0\right\}\Bigg\} =$$

$$\max\left\{2+\max\left\{\frac{16-2}{8-1}(\frac{8}{16}-1), 0\right\}, 3+\max\left\{\frac{30-3}{10-1}(\frac{10}{16}-1), 0\right\}\right\} = 3$$

For the DEP with a one-sided hash join, on the other hand, the first tuple cannot be produced until the entire hash table has been initialized. Thus, cost to first tuple can be estimated as follows:

$$\text{cost-first}\left({}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right) \approx$$

$$\max\Bigg\{\text{tupdelay}\left({}^{a_2^p}p_1^1, {}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right),$$

$$\text{cost}({}^{q_2^{rp}}p_1^2) + \text{tupfirst}\left({}^{a_2^{rp}}p_1^2, {}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right)\ \text{probecost}\Bigg\} =$$

$$
\begin{aligned}
&\max\Bigg\{\text{cost-first}({}^{a_2^p}p_1^1)+\\
&\max\left\{\frac{(\text{cost}({}^{a_2^p}p_1^1)-\text{cost-first}({}^{a_2^p}p_1^1)}{\text{card}({}^{a_2^p}p_1^1)-1}\left(\text{tupfirst}\left({}^{a_2^p}p_1^1, {}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right)-1\right),\right.\\
&\qquad\qquad\qquad\qquad\left.0\right\},\\
&\qquad\qquad\text{cost}({}^{q_2^{rp}}p_1^2)+\text{tupfirst}\left({}^{a_2^{rp}}p_1^2, {}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right)\text{probecost}\Bigg\}=
\end{aligned}
$$

$$
\begin{aligned}
&\max\Bigg\{\text{cost-first}({}^{a_2^p}p_1^1)+\\
&\quad\max\left\{\frac{\text{cost}({}^{a_2^p}p_1^1)-\text{cost-first}({}^{a_2^p}p_1^1)}{\text{card}({}^{a_2^p}p_1^1)-1}\left(\frac{\text{card}(p_1^1)}{\text{card}\left(p_1^1 \bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} p_1^2\right)}-1\right),0\right\},\\
&\qquad\qquad\text{cost}({}^{q_2^{rp}}p_1^2)+\frac{\text{card}(p_1^2)}{\text{card}\left(p_1^1 \bowtie_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} p_1^2\right)}\text{probecost}\Bigg\}=\\
&\qquad\qquad\quad\max\left\{2+\max\left\{\frac{16-2}{8-1}(\frac{8}{16}-1),0\right\},30+\frac{10}{16}\cdot 0.2\right\}=30.125
\end{aligned}
$$

Order properties Now, the order properties of the physical DEPs can be determined. Assuming that the join attributes a_2^P and a_2^{rp} are not known to be duplicate free, both the DEP with the merge join and the DEP with the one-sided hash join share the same order properties:

$$
\begin{aligned}
O\left({}^{a_2^p}p_1^1 \bowtie^{\text{M}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right) = O\left({}^{a_2^p}p_1^1 \bowtie^{\text{H}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right) =\\
O({}^{a_2^p}p_1^1)\cup\{a_i\in A(p_1^2), a_2^{rp}\rightsquigarrow a_i\}=\{a_1^e\}
\end{aligned}
$$

The DEP with the symmetric hash join, in contrast, provides no order properties:

$$
O\left({}^{a_2^p}p_1^1 \bowtie^{\text{SH}}_{\text{id}(a_2^p)=\text{id}(a_2^{rp})} {}^{q_2^{rp}}p_1^2\right)=\emptyset
$$

Tables 7.5 and 7.6 show an overview of the properties of the cross-fragment join operators.

Logical Properties		
$\mathrm{card}\left(G_{P_u} \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\mathrm{card}(G_{P_u})\frac{\mathrm{card}(G_{P_v})}{\mathrm{nsubt}(p_k^v)}$
$\mathrm{card}\left(G_{P_u} \bowtie_{\text{p-o-s}(\mathrm{id}(a_u^p),\mathrm{id}(a_v^{rp}))} G_{P_v}\right)$	$\approx$	$\mathrm{card}(G_{P_u})\frac{\mathrm{card}(G_{P_v})}{\mathrm{nsubt}(p_k^u)}$
$A\left(G_{P_u} \bowtie_{\mathrm{id}\left(a_u^p\right)\theta\,\mathrm{id}\left(a_v^{rp}\right)} G_{P_v}\right)$	$=$	$A\left(G_{P_u}\right) \cup A\left(G_{P_v}\right) \setminus \{a_u^p, a_v^{rp}\}$
Physical Properties – Merge Join		
$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\left\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\right\}$
$\text{cost-first}\left(G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\big\{\mathrm{tupdelay}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right),$ $\mathrm{tupdelay}\left(G_{P_v}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)\big\}$
$O\left(G_{P_u} \bowtie^{\mathrm{M}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$=$	$\begin{cases} O\left(G_{P_u}\right) \cup O\left(G_{P_v}\right) & \text{if } a_u^p \text{ or } a_v^{rp} \text{ dupl. free} \\ O\left(G_{P_u}\right) \cup \{a_i \in A\left(G_{P_v}\right), a_v^{rp} \rightsquigarrow a_i\} & \text{otherwise} \end{cases}$
Physical Properties – Hash Join		
$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})+$ $\mathrm{card}(G_{P_u})\,\mathrm{probecost}\}$
$\text{cost-first}\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\big\{\mathrm{tupdelay}\left(G_{P_u}, G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right),$ $\mathrm{cost}(G_{P_v})+$ $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)\,\mathrm{probecost}\big\}$
$O\left(G_{P_u} \bowtie^{\mathrm{H}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$=$	$\begin{cases} O\left(G_{P_u}\right) \cup \{a_i \in A\left(G_{P_v}\right), a_v^{rp} \rightsquigarrow a_i\} & \text{if } a_u^p \in O\left(G_{P_u}\right) \\ O\left(G_{P_u}\right) & \text{otherwise} \end{cases}$
Physical Properties – Symmetric Hash Join		
$\mathrm{cost}\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\left\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\right\}$
$\text{cost-first}\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$\approx$	$\max\big\{\mathrm{tupdelay}\left(G_{P_u}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right),$ $\mathrm{tupdelay}\left(G_{P_v}, G_{P_u} \bowtie_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)\big\}$
$O\left(G_{P_u} \bowtie^{\mathrm{SH}}_{\mathrm{id}(a_u^p)\,\theta\,\mathrm{id}(a_v^{rp})} G_{P_v}\right)$	$=$	$\emptyset$

Table 7.5: Cross-fragment join operator properties

Physical Properties – Pushed Index Join		
$\mathrm{cost}\left(\bar{p}_k^v\left(G_{P_u} \bowtie^{\mathrm{I}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right)$	$\approx$	$\max\{\mathrm{cost}(G_{P_u}) + \mathrm{subtcost}(p_k^v),$ $\mathrm{cost\text{-}first}(G_{P_u}) + \mathrm{card}(G_{P_u})\,\mathrm{subtcost}(p_k^v)\}$
$\mathrm{cost}\left(\bar{p}_k^v\left(G_{P_u} \ltimes_{\text{p-o-s}(\mathrm{id}(a_u^p),\mathrm{id}(a_v^{rp}))} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right)$	$\approx$	$\max\{\mathrm{cost}(G_{P_u})+$ $\frac{\mathrm{nsubt}(p_k^v)}{\mathrm{nsubt}(p_k^u)}\,\mathrm{subtcost}(p_k^v),$ $\mathrm{cost\text{-}first}(G_{P_u})+$ $\mathrm{card}(G_{P_u})\frac{\mathrm{nsubt}(p_k^v)}{\mathrm{nsubt}(p_k^u)}\,\mathrm{subtcost}(p_k^v)\Big\}$
$\mathrm{cost\text{-}first}\left(\bar{p}_k^v\left(G_{P_u} \bowtie^{\mathrm{I}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right)$	$\approx$	$\max\{\mathrm{tupdelay}\,(G_{P_u},$ $G_{P_u} \ltimes_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} p_k^v\big) +$ $\mathrm{subtcost}(p_k^v),$ $\mathrm{cost\text{-}first}(G_{P_u})+$ $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} p_k^v\right)$ $\mathrm{subtcost}(p_k^v)\}$
$\mathrm{cost\text{-}first}\left(\bar{p}_k^v\left(G_{P_u} \ltimes_{\text{p-or-s}(\mathrm{id}(a_u^p),\mathrm{id}(a_v^{rp}))} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right)$	$\approx$	$\max\{\mathrm{tupdelay}\,(G_{P_u},$ $G_{P_u} \ltimes_{\text{p-or-s}(\mathrm{id}(a_u^p),\mathrm{id}(a_v^{rp}))} p_k^v\big) +$ $\frac{\mathrm{nsubt}(p_k^v)}{\mathrm{nsubt}(p_k^u)}\,\mathrm{subtcost}(p_k^v),$ $\mathrm{cost\text{-}first}(G_{P_u})+$ $\mathrm{tupfirst}\left(G_{P_u}, G_{P_u} \ltimes_{\text{p-or-s}(\mathrm{id}(a_u^p),\mathrm{id}(a_v^{rp}))} p_k^v\right)$ $\frac{\mathrm{nsubt}(p_k^v)}{\mathrm{nsubt}(p_k^u)}\,\mathrm{subtcost}(p_k^v)\Big\}$
$O\left(\bar{p}_k^v\left(G_{P_u} \bowtie^{\mathrm{I}}_{\mathrm{id}(a_v^p)\,\theta\,\mathrm{id}(a_v^{rp})} \mathrm{scan}_{a_v^{rp}:RP_*^{i\to j}}\right)\right)$	$=$	$\begin{cases} O\left(G_{P_u}\right) \cup O\left({}^{a_v^{rp}}p_k^v\right) & \text{if } a_v^p \in O\left(G_{P_u}\right) \\ O\left(G_{P_u}\right) & \text{otherwise} \end{cases}$

Table 7.6: Cross-fragment join operator properties (cont'd)

7.4.3 Sort Operator

During plan enumeration, a sort operator (denoted as $\mathbb{S}$) may be inserted into a DEP to ensure that an intermediate result has a particular order property. This may be needed to allow the use of certain physical operators (e.g., the merge join operator requires its operands to be ordered by the attributes used in the join predicate) or to ensure that the nodes matched to the extraction point of the overall query are returned in document order as required by the XPath specification [24].

Since sorting does not introduce or remove any tuples, cardinality is unaffected by this operator:

$$\text{card}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})) = \text{card}(G_{P_u})$$

Similarly, sorting does not change the attributes in the tuples it processes. Thus, the attributes returned by the sort operator are the same attributes as those in the operand of the sort operator:

$$A\left(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})\right) = A\left(G_{P_u}\right)$$

While there are many different physical implementations of sorting (see Graefe's survey [59] for an overview), ranging from in-memory approaches for shorter sequences to disk-based merge approaches for larger volumes of data, in general, sorting requires that the sequence of tuples that is being sorted is fully materialized. Thus, the cost of a sort operator can be estimated as follows:

$$\text{cost}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})) \approx \text{cost}(G_{P_u}) + \text{sortcost}(\text{card}(G_{P_u}))$$

In this formula, sortcost(n) refers to the time consumed by sorting a sequence of n tuples. Since this cost depends solely on the implementation of the sort operator (the intricacies of which are outside the scope of this thesis), it is assumed that estimates for sortcost(n) are available (e.g., obtained experimentally).

Logical Properties		
$\text{card}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$	$=$	$\text{card}(G_{P_u})$
$A\left(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})\right)$	$=$	$A\left(G_{P_u}\right)$
Physical Properties		
$\text{cost}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$	$\approx$	$\text{cost}(G_{P_u}) + \text{sortcost}(\text{card}(G_{P_u}))$
$\text{cost-first}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$	$\approx$	$\text{cost}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$
$O\left(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})\right)$	$=$	$\{a_1\} \cup \{a_i \in A\left(G_{P_u}\right), a_1 \rightsquigarrow a_i\}$

Table 7.7: Sort operator properties

Since sorting requires all tuples to be fully materialized, in general, the first tuple is only returned once all tuples have been sorted[5]. Therefore, the time that elapses before the first tuple is returned can be estimated as follows:

$$\text{cost-first}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})) \approx \text{cost}(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$$

The semantics of $\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u}))$ are as follows: the sequence is sorted by attribute a_1, tuples having the same value for a_1 are then sorted by a_2, and so forth. However, as mentioned before, order properties as used in this work only include attributes by which a result is fully ordered (and not secondary order properties in a hierarchy). Thus, overall, the result of the sorting operation is ordered by a_1 and by all attributes whose order is implied by a_1:

$$O\left(\mathbb{S}_{[a_1,a_2,\ldots]}(G_{P_u})\right) = \{a_1\} \cup \{a_i \in A\left(G_{P_u}\right), a_1 \rightsquigarrow a_i\}$$

Table 7.7 shows an overview of the properties of the sort operator.

[5] Note that with some sort algorithms (such as selection sort), the first tuple may be returned before the full sequence of tuples has been sorted. Nevertheless, the sequence to be sorted has to be fully materialized first.

7.4.4 Outer Join, Grouping and Selection Operators

As described in Section 5.2.6.1, when a negation is folded into a cross-fragment join, a combination of three operators is used: a left outer join (⟕), a grouping ($\mathbb{G}$) with aggregation ($\mathbb{A}$) and finally a selection (σ):

$$\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u}\text{⟕}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})}G_{P_v}\right)\right)$$

Since, in a DEP, these operators are always used together, for the purposes of this cost model they are treated as a single operator. This section describes how the properties of this cluster of operators can be determined.

To allow for an efficient implementation of this cluster of operators, the operands of the outer join are required to be ordered by the attributes referenced in the join predicate, i.e., $a_v^p \in O\left(G_{P_u}\right)$ and $a_v^{rp} \in O\left(G_{P_v}\right)$. This can easily be assured by choosing sub-plans for G_{P_u} and G_{P_v} that provide these order properties, or by inserting a sort operator.

Definition 5.4 on page 99 requires that G_{P_v} has only a single extraction point a_v^{rp} (i.e., $A\left(G_{P_v}\right) = \{a_v^{rp}\}$). Now, assume that the result of G_{P_v} is free of duplicates. Since G_{P_v} is ordered by a_v^{rp}, duplicates can be eliminated with negligible overhead if necessary. This implies that the probability that there is a match for a given root proxy ID is approximately $\frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^v)}$, where p_k^v is the LQP that yields attribute a_v^{rp}. Conversely, the probability that there is no match (denoted as nullprob(G_{P_v})) can be estimated as follows:

$$\text{nullprob}(G_{P_v}) \approx 1 - \frac{\text{card}(G_{P_v})}{\text{nsubt}(p_k^v)}$$

Since an outer join is used, nullprob(G_{P_v}) corresponds to the probability that a null value is supplied for a given proxy node matched to a_v^p.

Let p_k^u be the LQP in P_u that produces the attribute a_v^p. Now consider a modified LQP $\hat{p}_k^u$, corresponding to a local QTP $\hat{q}_k^u$ that is identical to q_k^u except that it does not designate a_v^p as an extraction point. Both $\hat{p}_k^u$ and p_k^u, as LQPs, are required to produce duplicate-free results. Thus, the average number of tuples in $R(p_k^u)$ that share the same values for $A\left(p_k^u\right) \setminus \{a_v^p\}$ and differ only in their value of a_v^p (denoted as samegroup(p_k^u, a_v^p)) can be estimated as follows:

$$\text{samegroup}(p_k^u, a_v^p) \approx \frac{\text{card}(p_k^u)}{\text{card}(\hat{p}_k^u)}$$

For each tuple in $R(p_k^u)$ the number of corresponding tuples in $R(G_{P_u})$ can be estimated as $\frac{\text{card}(G_{P_u})}{\text{card}(p_k^u)}$. Thus, the average number of tuples in $R(G_{P_u})$ that share the same values for the attributes in $A\left(G_{P_u}\right) \setminus \{a_v^p\}$ (denoted as samegroup(G_{P_u}, a_v^p)) can be estimated by the following formula:

$$\text{samegroup}(G_{P_u}, a_v^p) \approx \frac{\text{card}(G_{P_u})}{\text{card}(p_k^u)} \frac{\text{card}(p_k^u)}{\text{card}(\hat{p}_k^u)} = \frac{\text{card}(G_{P_u})}{\text{card}(\hat{p}_k^u)}$$

Since the result of the outer join has the same cardinality as G_{P_u}, the number of groups produced by the grouping operator can be estimated as $\frac{\text{card}(G_{P_u})}{\text{samegroup}(G_{P_u}, a_v^p)}$.

Since each group, on average, contains samegroup(p_k^u, a_v^p) different values of a_v^p and since, in order to pass through the selection, none of these values of a_v^p must result in a matching a_v^{rp}, the probability that a given group satisfies the selection can be estimated as nullprob$(G_{P_v})^{\text{samegroup}(p_k^u, a_v^p)}$.

This leads to the following estimate for the overall cardinality of this operator cluster:

$$\begin{aligned}\text{card}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right) \approx \\ \frac{\text{card}(G_{P_u})}{\text{samegroup}(G_{P_u})}\ \text{nullprob}(G_{P_v})^{\text{samegroup}(p_k^u, a_v^p)}\end{aligned}$$

To estimate the response time cost of the operator cluster, it is helpful to look at each operator separately. Since G_{P_u} is ordered by a_v^p and G_{P_v} is ordered by a_v^{rp}, the outer join can be evaluated using a merge join with outer join semantics ($⟕^{\text{M}}$). As described in Section 7.4.2.1, the cost of this operator can be estimated as:

$$\text{cost}\left(G_{P_u} ⟕^{\text{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right) \approx \max\{\text{cost}(G_{P_u}), \text{cost}(G_{P_v})\}$$

If $R(G_{P_u})$ (and therefore the result of the outer join) is ordered by $A\left(G_{P_u}\right) \setminus \{a_v^p\}$, grouping can be done with minimal overhead on a pipelined basis. Otherwise, the join result (which is of size $\mathrm{card}(G_{P_u})$) needs to be materialized and sorted first. Thus, the cost of grouping can be estimated as follows:

$$\begin{aligned}
&\mathrm{cost}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\mathrm{count}(a_v^{rp})}\left(G_{P_u} ⟕^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}\right)\right) \approx \\
&\qquad\begin{cases} \max\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\} & \text{if } A\left(G_{P_u}\right) \setminus \{a_v^p\} \subseteq O\left(G_{P_u}\right) \\ \max\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\} + \mathrm{sortcost}\left(\mathrm{card}(G_{P_u})\right) & \text{otherwise} \end{cases}
\end{aligned}$$

The selection adds minimal overhead and therefore the overall cost estimate is as follows:

$$\begin{aligned}
&\mathrm{cost}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\mathrm{count}(a_v^{rp})}\left(G_{P_u} ⟕^{\mathrm{M}}_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp})} G_{P_v}\right)\right)\right) \approx \\
&\qquad\begin{cases} \max\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\} & \text{if } A\left(G_{P_u}\right) \setminus \{a_v^p\} \subseteq O\left(G_{P_u}\right) \\ \max\{\mathrm{cost}(G_{P_u}), \mathrm{cost}(G_{P_v})\} + \mathrm{sortcost}\left(\mathrm{card}(G_{P_u})\right) & \text{otherwise} \end{cases}
\end{aligned}$$

To estimate time to first tuple, it is again necessary to distinguish between the case where G_{P_u} is ordered by all the attributes on which grouping is performed and the case where it is not. If $R(G_{P_u})$ is ordered by all required attributes, it is possible to fully pipeline the operator cluster. Since each group consists of $\mathrm{samegroup}(G_{P_u}, a_v^p)$ tuples, on average the same number of input tuples from G_{P_u} are required before the first output tuple can be produced. Using a formula that is analogous to the formula for tupdelay() in Section 7.4.2.1, the delay until a sufficient number of tuples have been produced by G_{P_u} (denoted as $\mathrm{groupdelay}(G_{P_u}, a_v^p)$) can be estimated as follows:

$$\begin{aligned}
&\mathrm{groupdelay}(G_{P_u}, a_v^p) \approx \\
&\qquad \text{cost-first}(G_{P_u}) + \max\left\{\frac{\mathrm{cost}(G_{P_u}) - \text{cost-first}(G_{P_u})}{\mathrm{card}(G_{P_u}) - 1}\left(\mathrm{samegroup}(G_{P_u}, a_v^p) - 1\right), 0\right\}
\end{aligned}$$

To estimate the number of tuples required from G_{P_v}, the formula from Section 7.4.2.1 can be applied directly. If $R(G_{P_u})$ is not ordered by all required attributes, no pipelining is possible, and the result of the outer join needs to be fully materialized. Together, this leads to the following estimate:

$$\text{cost-first}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} \leftouterjoin^{\mathrm{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right) \approx \begin{cases} \max\left\{\text{groupdelay}(G_{P_u}, a_v^p), \text{tupdelay}(G_{P_v}, G_{P_V} \bowtie_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_u})\right\} \\ \qquad\qquad \text{if } A\left(G_{P_u}\right) \setminus \{a_v^p\} \subseteq O\left(G_{P_u}\right) \\ \max\left\{\text{cost}(G_{P_u}), \text{cost}(G_{P_v})\right\} + \text{sortcost}\left(\text{card}(G_{P_u})\right) \\ \qquad\qquad \text{otherwise} \end{cases}$$

Now, consider the order properties of the operator cluster. If $A\left(G_{P_u}\right)\setminus\{a_v^p\} \subseteq O\left(G_{P_u}\right)$, then pipelined execution can be used and the order on these attributes is preserved. Otherwise, grouping implicitly sorts the result of the outer join by $A(G_{P_u}) \setminus \{a_v^p\}$, resulting in a set of order properties consisting of some $a_1 \in A(G_{P_u})\setminus\{a_v^p\}$ and all attributes whose order is implied by a_1. Therefore, the order properties of the operator cluster can be described as follows:

$$O\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} \leftouterjoin^{\mathrm{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right) = \begin{cases} A\left(G_{P_u}\right) \setminus \{a_v^p\} & \text{if } A\left(G_{P_u}\right) \setminus \{a_v^p\} \subseteq O\left(G_{P_u}\right) \\ \{a_1\} \cup \{a_i \in \left(A\left(G_{P_u}\right) \setminus \{a_v^p\}\right) \mid a_1 \rightsquigarrow a_i\} & \text{otherwise} \end{cases}$$

Table 7.8 shows an overview of the properties of the operator cluster consisting of outer join, grouping with aggregation, and selection.

Logical Properties		
$\text{card}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$	$\approx$	$\frac{\text{card}(G_{P_u})}{\text{samegroup}(G_{P_u})}\ \text{nullprob}(G_{P_v})^{\text{samegroup}(p_k^u, a_v^p)}$
$A\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$	$=$	$A\left(G_{P_u}\right)\{a_v^n, a_v^p\}$
Physical Properties		
$\text{cost}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕^{\text{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$	$\approx$	$\begin{cases}\max\{\text{cost}(G_{P_u}), \text{cost}(G_{P_v})\} & \text{if } A\left(G_{P_u}\right)\setminus\{a_v^p\}\subseteq O\left(G_{P_u}\right)\\ \max\{\text{cost}(G_{P_u}), \text{cost}(G_{P_v})\} & \\ +\text{sortcost}\left(\text{card}(G_{P_u})\right) & \text{otherwise}\end{cases}$
$\text{cost-first}\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕^{\text{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$	$\approx$	$\begin{cases}\max\{\text{groupdelay}(G_{P_u}, a_v^p), & \\ \text{tupdelay}(G_{P_v}, & \\ G_{P_V} \bowtie_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_u})\} & \text{if } A\left(G_{P_u}\right)\setminus\{a_v^p\}\subseteq O\left(G_{P_u}\right)\\ \max\{\text{cost}(G_{P_u}), \text{cost}(G_{P_v})\} & \\ +\text{sortcost}\left(\text{card}(G_{P_u})\right) & \text{otherwise}\end{cases}$
$O\left(\sigma_{a_v^{rp}=0}\left(\mathbb{G}_{A(G_{P_u})\setminus\{a_v^p\}}\mathbb{A}_{\text{count}(a_v^{rp})}\left(G_{P_u} ⟕^{\text{M}}_{\text{id}(a_v^p)=\text{id}(a_v^{rp})} G_{P_v}\right)\right)\right)$	$=$	$\begin{cases}A\left(G_{P_u}\right)\setminus\{a_v^p\} & \text{if } A\left(G_{P_u}\right)\setminus\{a_v^p\}\subseteq O\left(G_{P_u}\right)\\ \{a_1\} & \\ \cup\{a_i\in\left(A\left(G_{P_u}\right)\setminus\{a_v^p\}\right) & \\ \mid a_1 \rightsquigarrow a_i\} & \text{otherwise}\end{cases}$

Table 7.8: Outer join, grouping and selection operator properties

7.5 Enumerating DEP Alternatives

Techniques for enumerating plan alternatives have received significant attention in the literature, and many different approaches exist to solve this problem (cf. Section 3.3.3). These approaches can be broadly categorized into techniques that find the optimal plan by fully enumerating the entire search space (while possibly pruning plans that can be shown not to lead to the optimal result, e.g., [87, 121]) and techniques that rely on randomization or heuristics to find a good, but not necessarily optimal plan more quickly (e.g., [61, 92]). The cost estimation techniques described in the previous sections can be used in combination with either approach.

In traditional distributed query processing, where the number of plan alternatives is very large due to many different options for placing operators, randomized strategies have often been favoured because they tend to cope better with extremely large search spaces [61]. When optimizing DEPs as described here, the search space is comparatively small. This results from the fact that LQPs are treated as black boxes and only a small set of physical LQPs are considered during distributed query optimization (those that are better than their alternatives in total cost, time to first tuple, or order properties). Pruning further reduces the number of LQPs considered. Therefore, DEPs consist of relatively few large atoms, which keeps the size of the search space manageable, making exhaustive enumeration (e.g., via dynamic programming) of the space of DEPs a feasible alternative in many cases.

Since the enumeration of plan alternatives is a well understood problem, this section does not provide a detailed description of any one technique. Instead, requirements are laid out that need to be satisfied by any plan enumeration technique used in this context. For an overview of existing enumeration techniques that meet these requirements refer to Section 3.3.3.

7.5.1 DEP Shapes

To find the optimal DEP, it is necessary to consider bushy plans. Restricting plan enumeration to left-deep plans may yield sub-optimal results since they may limit parallelism. Many of the plan enumeration techniques discussed in Section 3.3.3 satisfy this requirement.

7.5.2 Comparing Sub-Plans

Bottom-up approaches for plan enumeration operate by comparing sub-plans consisting of a subset of the LQPs needed to answer a query. At this stage, candidate sub-plans are usually discarded when their estimated cost is higher than that of other sub-plans covering the same subset. Once a sub-plan is discarded, it is no longer considered as a building block for the overall DEP.

While the best overall DEP is chosen based on response time, when comparing sub-plans, it is also necessary to take into account time to first tuple and the set of order properties. This is because, for example, a sub-plan G'_{P_u} with sub-optimal response time cost may lead to a better DEP than a sub-plan G_{P_u} with lower response time cost if G'_{P_u}

has additional order properties that increase the efficiency of other operators in the DEP (e.g., by allowing for the use of a more efficient physical join operator or by avoiding an explicit sorting step).

Thus, a sub-plan G'_{P_u} can only be discarded as inferior to another sub-plan covering the same LQPs G_{P_u} if G'_{P_u} is no better than G_{P_u} in any of the three metrics. The following definition formalizes this in a way that is is analogous to Definition 7.8 for LQPs:

Definition 7.9. Let $P_u \subset P$ be a subset of the LQPs P needed to answer query q_k. Let G_{P_u} and G'_{P_u} be sub-plans consisting of the LQPs in P_u. Then G'_{P_u} is *inferior* to G_{P_u} and, when building a DEP for q_k, G'_{P_u} can be discarded in favour of G_{P_u} if all of the following hold:

- $\text{cost}(G'_{P_u}) \geq \text{cost}(G_{P_u})$, and
- $\text{cost-first}(G'_{P_u}) \geq \text{cost-first}(G_{P_u})$, and
- $O\left(G'_{P_u}\right) \subseteq O\left(G_{P_u}\right)$.

and at least one of the following holds:

- $\text{cost}(G'_{P_u}) > \text{cost}(G_{P_u})$, and
- $\text{cost-first}(G'_{P_u}) > \text{cost-first}(G_{P_u})$, and
- $O\left(G'_{P_u}\right) \subset O\left(G_{P_u}\right)$.

■

If there are two sub-plans for the same set of LQPs P_u and none of them dominates the other, then both must be considered during the enumeration of possible DEPs.

In contrast to this, the final DEP is chosen purely based on cost and the one required order property (ensuring that the result is in document order). Thus, at this point, the other properties no longer matter.

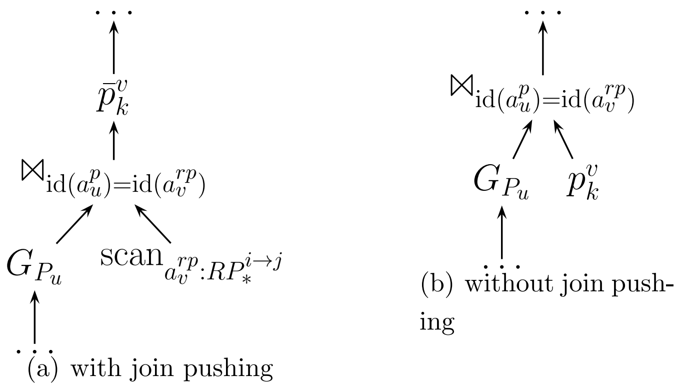
7.5.3 Execution Order Constraints

While cross-fragment joins can be re-ordered arbitrarily (as long as both operands of the join contain the necessary join attributes), the same is not true for merge operators and outer join/group/select clusters. For merge operators, this is because all sub-plans that are merged are required to return tuples consisting of the same set of attributes (as specified in Definition 5.3). For outer join/group/select clusters, the right-hand side input is required to return tuples consisting of a single attribute, the join attribute (as specified in Definition 5.4). In both cases, these constraints limit the order in which operators can be executed by a DEP. Therefore, plan enumeration needs to take these constraints into account.

7.6 Dynamic DEP Adaptation

So far, the focus in this chapter has been on determining the best DEP for a given query by performing static optimization before query evaluation begins. While this works well when cost estimates are close to actual costs, in practice, this may not always hold. Inaccurate cost estimates may be caused by data skew (invalidating the independence assumption made by this cost model) or resource contention resulting from the evaluation of multiple queries at the same time. This section describes how this problem can be addressed by dynamically adapting a DEP while it is being executed. This makes the performance of distributed query execution more robust with regard to cost estimation errors.

One of the key decisions that is made when optimizing a DEP is whether a given cross-fragment join $G_{P_u} \bowtie_{\mathrm{id}(a_v^p)=\mathrm{id}(a_v^{rp}))} p_k^v$ should be pushed into the LQP p_k^v. Pushing the join into the LQP has the advantage of potentially skipping large portions of the fragment corresponding to p_k^v and thereby reducing the overall response time. On the negative side, pushing the join into the LQP stalls execution of $\bar{p}_k^v$ (the remainder LQP corresponding to p_k^v) until the first tuple has been received from G_{P_u}. If this tuple arrives after a long delay, this can mean that pushing the join into the LQP leads to an overall response time that is significantly higher than the response time of a DEP in which the join is not pushed into p_k^v. If the time to the first tuple of G_{P_u} was underestimated, this can result in a sub-optimal DEP.



$\bar{p}_k^v$
$\bowtie_{\mathrm{id}(a_u^p)=\mathrm{id}(a_v^{rp})}$
G_{P_u}
$\mathrm{scan}_{a_v^{rp}:RP_*^{i \to j}}$
$\bowtie_{\mathrm{id}(a_u^p)=\mathrm{id}(a_v^{rp})}$
G_{P_u}
p_k^v

(a) with join pushing

(b) without join pushing

Figure 7.18: Dynamic adaptation of DEP

To avoid this problem, it is possible to keep track of the delay with which the pushed cross-fragment join (shown in Figure 7.18(a)) receives its first tuple from G_{P_u}. If the delay exceeds the estimate of cost-first(G_{P_u}) to an extent that renders pushing the join into p_k^v sub-optimal, the corresponding portion of the DEP can be modified by switching from the DEP in Figure 7.18(a) to the DEP in Figure 7.18(b). Note that this is possible because, as long as no tuples have been received from G_{P_u}, $\bar{p}_k^v$ has not been evaluated over any sub-trees in its corresponding fragment and thus no tuples have been returned from the DEP fragment shown in Figure 7.18(a).

Since $\bar{p}_k^v$ is not executed until the first tuple is received from G_{P_u} and assuming that the site to which this LQP is assigned is otherwise idle, it is even possible to start evaluating p_k^v as soon as query execution starts. If the first tuple is received from G_{P_u} within the estimated time, p_k^v is aborted and $\bar{p}_k^v$ is executed in its stead as shown in Figure 7.18(a). If the first tuple from G_{P_u} does not arrive in a timely manner, query execution can proceed based on the DEP fragment shown in Figure 7.18(b), in which case execution of p_k^v will already be well underway.

7.7 Summary

This chapter has introduced a cost model that makes it possible to accurately predict the performance of DEPs constructed based on the query evaluation techniques from Chapters

5 and 6. This is achieved by traversing the DEP in a bottom-up fashion, starting with the LQPs contained in the DEP. For each operator in the DEP, a set of properties is determined using the formulas presented in this chapter until finally the cost of the entire DEP can be inferred. By enumerating the set of candidate DEPs for a given query and comparing them based on their estimated cost, the best DEP for a given query and distributed collection can be chosen.

Chapter 8

Cost-Based Fragmentation of XML Collections

The cost-based optimization technique presented in Chapter 7 makes it possible to determine the best plan for evaluating a query over a given distributed collection. While this is useful for improving the performance and scalability of query evaluation over a collection with a fixed fragmentation schema, additional performance gains can be obtained by tailoring the fragmentation schema for the query workload that is being evaluated. Assuming that this query workload is known ahead of time (as is frequently the case), the characteristics of the queries in this workload can be taken into account when determining how the collection should be fragmented and distributed. This yields a fragmentation schema that is specialized for the query workload. When evaluating the anticipated queries over a collection fragmented according to this specialized fragmentation schema, performance and scalability can be improved significantly in many cases.

A naïve strategy for determining the best fragmentation schema for a given query workload is to exhaustively enumerate all possible fragmentation schemas. For each of these schemas, the cost of evaluating the query workload can then be determined using the cost estimation techniques presented in Chapter 7. Finally, the fragmentation schema that yields the lowest cost for the queries in the workload can be chosen. While this technique is guaranteed to lead to the optimal fragmentation (i.e., the fragmentation schema that minimizes the estimated cost of evaluating the query workload), the search space of possible fragmentation schemas that need to be enumerated is very large. Even when only vertical fragmentation steps are considered, the number of ways in which a schema with $|\Sigma|$ node types can be partitioned is $B_{|\Sigma|}$, the $|\Sigma|$th Bell number [23], which grows exponentially in $|\Sigma|$. Considering hybrid fragmentation schemas consisting of both vertical and horizontal fragmentation steps further increases the size of the search space. Thus, complete enumeration of all possible fragmentation schemas is generally infeasible for all but the smallest schemas.

To obtain a feasible fragmentation technique, a heuristic strategy is proposed in this chapter. Since the cost of a DEP is dominated by the cost of evaluating the individual LQPs used in this DEP (cf. Chapter 7), the heuristics employed in this strategy are based on repeatedly reducing the cost of the most expensive LQP and thereby reducing the cost of the overall DEP.

While the fragmentation schema obtained using this strategy is not guaranteed to be optimal, the heuristics ensure that it allows for the efficient evaluation of the queries in the workload using the techniques presented in this work. As will be shown in Chapter 9, this leads to good performance results in practice.

The heuristic fragmentation technique presented here can be characterized as a greedy strategy. It begins with an initial, vertical fragmentation schema, consisting of a large number of small fragments. Then, the query workload is localized and LQPs corresponding to the fragments in the initial fragmentation are obtained. To improve the fragmentation, the greedy strategy attempts to decrease the cost of evaluating the LQP with the highest estimated cost. This is done either by merging the fragment corresponding to the most expensive LQP (if this reduces the cost of the most expensive LQP) or by horizontally splitting this fragment. This is repeated until the cost of the most expensive LQP can no longer be reduced. At this point, the fragmentation strategy terminates, and a fragmentation schema that is tailored to the query workload has been obtained. The collection is then fragmented according to this fragmentation schema, and each fragment is placed at a separate site in the system.

The rationale behind this approach is as follows: The cost-based optimization strategy described in Chapter 7 aims to obtain a DEP in which the LQPs are evaluated in parallel. Thus, query performance is limited by the most expensive LQP contained in a DEP and focusing on reducing the cost of this LQP is a reasonable strategy to obtain a fragmentation schema that improves query performance.

The remainder of this chapter is organized as follows: In Section 8.1, the initial fragmentation schema used by the greedy strategy is described. Then, Section 8.2 describes how the initial fragmentation is refined by repeatedly modifying the fragment corresponding to the most expensive LQP until no further improvement is possible.

8.1 Initial Fragmentation Schema

The starting point of the greedy fragmentation strategy described in this chapter is a fragmentation schema consisting of many small vertical fragments. To obtain the initial fragmentation schema, in general, each node type is placed in its own fragment. However, there are two exceptions to this rule:

Since fragmentation schemas are required to be acyclic (cf. Section 4.2), special care needs to be taken for schema graphs that contain cycles. In this case, node types that are part of a cycle are placed together into the same fragment, resulting in one fragment for each such cycle. For an example of this, consider the schema graph shown in Figure

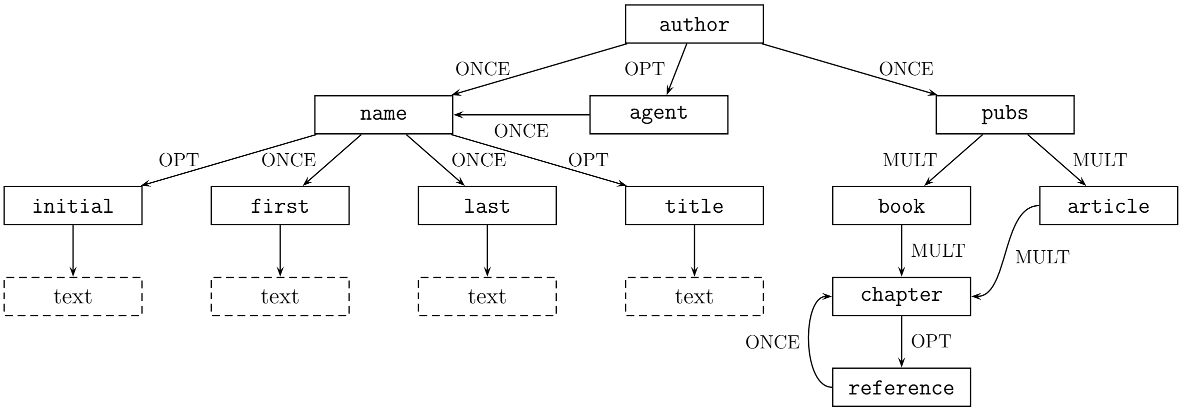


Figure 8.1: An XML schema graph

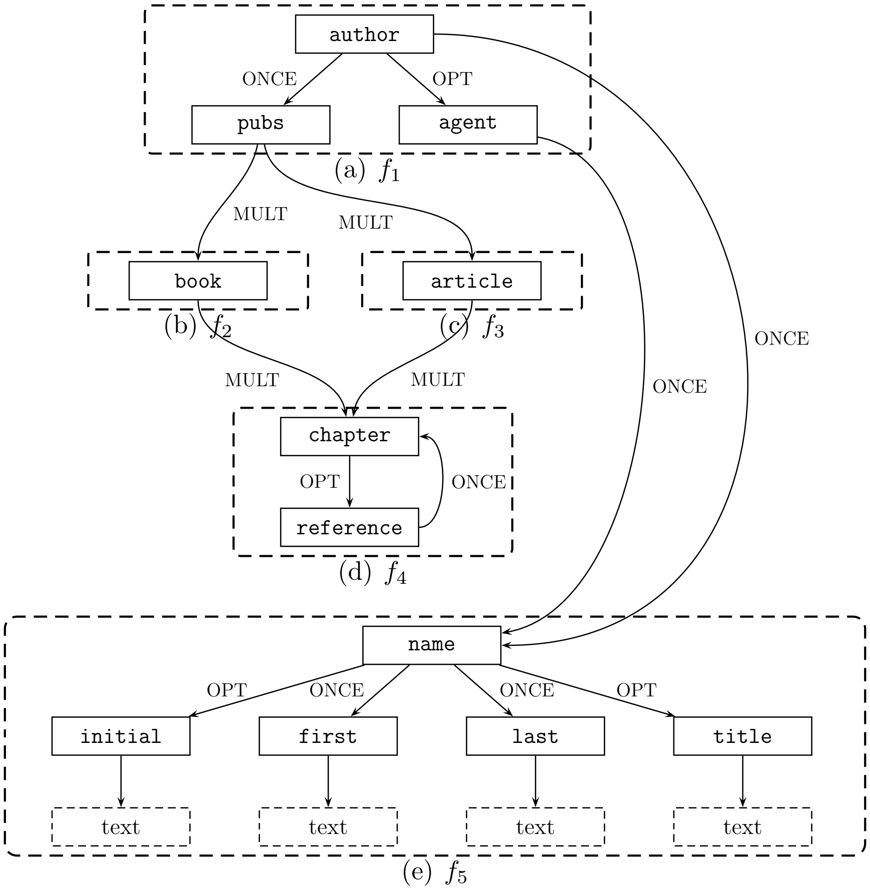


Figure 8.2: Initial fragmentation schema

8.1. Since this schema graph contains a cycle consisting of the node types `chapter` and `reference`, both of these node types are placed into the same fragment (f_4) in the initial fragmentation schema.

The second exception is concerned with node types that only have a single incoming edge in the schema, which must have a cardinality of ONCE or OPT. Conceptually, these node types represent mandatory (in the case of ONCE) or optional (in the case of OPT) attributes of the node type from which the incoming edge originates. Thus, in the initial fragmentation schema, they are placed in the same fragment.

In the schema graph shown in Figure 8.1, there are several instances of this. The node type pubs is only reachable via an edge from the node type author and this edge has the cardinality ONCE. Thus, author and pubs are placed in the same fragment (f_1) in the initial fragmentation schema shown in Figure 8.2. The node type agent is similarly reachable from the node type author via a unique incoming edge with cardinality OPT and is thus included in the same fragment. The node types initial, first, last, and title are all reachable via a unique edge from node type name and are thus included in fragment f_5. Note that, as in the schema graph, the dashed boxes in this fragment do not refer to node types. Instead they denote the text content of nodes of the types initial, first, last, and title, respectively.

8.2 Improving the Fragmentation

The initial fragmentation schema is then improved by repeatedly modifying the fragment corresponding to the most expensive LQP (thereby reducing the cost of this LQP) until no further improvement can be made. This is done by the recursive function fragalg(F, Q, Q_{todo}) shown in Algorithm 6, which is initially called with the following arguments:

- the initial fragmentation schema, represented as a set of fragments F,
- the set of queries in the workload Q_{all}, and
- the set of queries for which the fragmentation schema has not yet been optimized (Q_{todo}), which is initially equal to the set of all queries in the workload (Q_{all}).

To improve the fragmentation schema F, the queries in Q_{todo} are localized based on this fragmentation schema, and the LQP with the highest estimated cost (denoted as $p^{\text{todo}}_{\text{max}}$) is identified (line 2). Then, the fragment corresponding to this LQP (denoted as $f^{\text{todo}}_{\text{max}}$) is determined (line 4). The algorithm then attempts to improve the fragmentation schema by modifying $f^{\text{todo}}_{\text{max}}$ in one of the following three ways:

- The first possible way of improving the fragmentation schema is to merge $f^{\text{todo}}_{\text{max}}$ with one of its ancestor fragments (lines 7–17). By doing this, it may be possible to avoid

Algorithm 6: $\mathrm{fragalg}(F, Q_{\mathrm{all}}, Q_{\mathrm{todo}})$ *improves fragmentation F for queries Q_{todo}*

```
input  : set of fragments $F$, query workload $Q_{\mathrm{all}}$, queries not yet optimized $Q_{\mathrm{todo}}$ (initial equal to $Q_{\mathrm{all}}$)
output : improved set of fragments
$p_{\mathrm{max}}^{\mathrm{all}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{all}}$ over fragments $F$
$p_{\mathrm{max}}^{\mathrm{todo}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{todo}}$ over fragments $F$
$q_{\mathrm{max}}^{\mathrm{todo}} \leftarrow$ query corresponding to LQP $p_{\mathrm{max}}^{\mathrm{todo}}$
$f_{\mathrm{max}}^{\mathrm{todo}} \leftarrow$ fragment corresponding to LQP $p_{\mathrm{max}}^{\mathrm{todo}}$
$F_{\mathrm{modified}} \leftarrow F$
$p_{\mathrm{max}}^{\mathrm{modified}} \leftarrow p_{\mathrm{max}}^{\mathrm{todo}}$
for $p_{\mathrm{ancestor}} \in P$ such that $p_{\mathrm{ancestor}}$ is an ancestor of $p_{\mathrm{max}}^{\mathrm{todo}}$ do
    $f_{\mathrm{ancestor}} \leftarrow$ fragment corresponding to LQP $p_{\mathrm{ancestor}}$
    $F_{\mathrm{intermediate}} \leftarrow \{f \in F \mid f$ is reachable from $f_{\mathrm{ancestor}}$ and $f_{\mathrm{max}}^{\mathrm{todo}}$ is reachable from $f\}$
    $F_{\mathrm{aftermerge}} \leftarrow \big(F \setminus \big(F_{\mathrm{intermediate}} \cup \{f_{\mathrm{max}}^{\mathrm{todo}}, f_{\mathrm{ancestor}}\}\big)\big) \cup \{(\bigcup_{f \in F_{\mathrm{intermediate}}} f) \cup f_{\mathrm{max}}^{\mathrm{todo}} \cup f_{\mathrm{ancestor}}\}$
    $p_{\mathrm{max}}^{\mathrm{aftermerge,\ todo}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{todo}}$ over fragments $F_{\mathrm{aftermerge}}$
    $p_{\mathrm{max}}^{\mathrm{aftermerge,\ all}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{all}}$ over fragments $F_{\mathrm{aftermerge}}$
    if $\mathrm{cost}(p_{\mathrm{max}}^{\mathrm{aftermerge,\ todo}}) < \mathrm{cost}(p_{\mathrm{max}}^{\mathrm{modified}}) \wedge \forall q_i \in Q_{\mathrm{all}}, \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{aftermerge,\ all}}) \leq \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{all}})$ then
        $F_{\mathrm{modified}} \leftarrow F_{\mathrm{aftermerge}}$
        $p_{\mathrm{max}}^{\mathrm{modified}} \leftarrow p_{\mathrm{max}}^{\mathrm{aftermerge,\ todo}}$
if $F_{\mathrm{modified}} \neq F$ then
    return $\mathrm{fragalg}(F_{\mathrm{modified}}, Q_{\mathrm{all}}, Q_{\mathrm{todo}})$
if some sub-trees in $f_{\mathrm{max}}^{\mathrm{todo}}$ have node type paths that are incompatible with query $q_{\mathrm{max}}^{\mathrm{todo}}$ then
    $f_{\mathrm{max}}^{\mathrm{compat}} \leftarrow \{s \in f_{\mathrm{max}}^{\mathrm{todo}} \mid \mathrm{ntpath}(s)$ compatible with $q_{\mathrm{max}}^{\mathrm{todo}}\}$
    $f_{\mathrm{max}}^{\mathrm{incompat}} \leftarrow \{s \in f_{\mathrm{max}}^{\mathrm{todo}} \mid \mathrm{ntpath}(s)$ incompatible with $q_{\mathrm{max}}^{\mathrm{todo}}\}$
    $F_{\mathrm{aftersplit}} \leftarrow (F \setminus \{f_{\mathrm{max}}^{\mathrm{todo}}\}) \cup \{f_{\mathrm{max}}^{\mathrm{compat}} \cup f_{\mathrm{max}}^{\mathrm{incompat}}\}$
    $p_{\mathrm{max}}^{\mathrm{aftersplit,\ todo}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{todo}}$ over fragments $F_{\mathrm{aftersplit}}$
    $p_{\mathrm{max}}^{\mathrm{aftersplit,\ all}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{all}}$ over fragments $F_{\mathrm{aftersplit}}$
    if $\mathrm{cost}(p_{\mathrm{max}}^{\mathrm{aftersplit,\ todo}}) < \mathrm{cost}(p_{\mathrm{max}}^{\mathrm{modified}}) \wedge \forall q_i \in Q_{\mathrm{all}}, \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{aftersplit,\ all}}) \leq \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{all}})$ then
        return $\mathrm{fragalg}(F_{\mathrm{aftersplit}}, Q_{\mathrm{all}}, Q_{\mathrm{todo}})$
for value constraint $c$ in $p_{\mathrm{max}}^{\mathrm{todo}}$ do
    $F_{\mathrm{partition}} \leftarrow$ partitioning of $f_{\mathrm{max}}^{\mathrm{todo}}$ based on value constraint $c$
    $F_{\mathrm{aftersplit}} \leftarrow (F \setminus \{f_{\mathrm{max}}^{\mathrm{todo}}\}) \cup F_{\mathrm{partition}}$
    $p_{\mathrm{max}}^{\mathrm{aftersplit,\ todo}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{todo}}$ over fragments $F_{\mathrm{aftersplit}}$
    $p_{\mathrm{max}}^{\mathrm{aftersplit,\ all}} \leftarrow$ most expensive LQP when evaluating queries in $Q_{\mathrm{all}}$ over fragments $F_{\mathrm{aftersplit}}$
    if $\mathrm{cost}(p_{\mathrm{max}}^{\mathrm{aftersplit,\ todo}}) < \mathrm{cost}(p_{\mathrm{max}}^{\mathrm{modified}}) \wedge \forall q_i \in Q_{\mathrm{all}}, \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{aftersplit,\ all}}) \leq \mathrm{cost}(p_{i,\mathrm{max}}^{\mathrm{all}})$ then
        $F_{\mathrm{modified}} \leftarrow F_{\mathrm{aftersplit}}$
        $p_{\mathrm{max}}^{\mathrm{modified}} \leftarrow p_{\mathrm{max}}^{\mathrm{aftersplit,\ todo}}$
if $F_{\mathrm{modified}} \neq F$ then
    return $\mathrm{fragalg}(F_{\mathrm{modified}}, Q_{\mathrm{all}}, Q_{\mathrm{todo}})$
if $Q_{\mathrm{todo}} \setminus \{q_{\mathrm{max}}\} \neq \emptyset$ then
    return $\mathrm{fragalg}(F, Q_{\mathrm{all}}, Q_{\mathrm{todo}} \setminus \{q_{\mathrm{max}}\})$
return $F$
```

accessing a portion of the data contained in this fragment, reducing the cost of the most expensive LQP.

- Alternatively, it may be possible to split $f_{\text{max}}^{\text{todo}}$ horizontally based on the node type paths associated with the root proxy nodes at which the sub-trees in this fragment are rooted (lines 18–25). This is only possible if $f_{\text{max}}^{\text{todo}}$ is not at the root of the fragmentation schema (otherwise, sub-trees in this fragment would be rooted at ordinary nodes from the collection rather than at root proxy nodes).

- As a third alternative, value or structural constraints in $p_{\text{max}}^{\text{todo}}$ can be exploited to split $f_{\text{max}}^{\text{todo}}$ horizontally (lines 26–35).

If one of these possible improvements decreases the cost of the most expensive LQP, the fragmentation is modified and fragalg() is called recursively with the modified fragmentation (lines 17, 25, and 35).

If none of the improvement steps reduces the cost of the most expensive LQP, the algorithm assumes that no further optimization of the fragmentation schema is possible for the query corresponding to LQP $p_{\text{max}}^{\text{todo}}$ (denoted as $q_{\text{max}}^{\text{todo}}$). Thus, $q_{\text{max}}^{\text{todo}}$ is removed from the set of queries considered when determining the most expensive LQP (Q_{todo}) and fragalg() is called recursively (line 37). During the next execution, fragalg() then no longer considers LQPs corresponding to $q_{\text{max}}^{\text{todo}}$ when determining the most expensive LQP $p_{\text{max}}^{\text{todo}}$. However, when verifying whether a potential modification to the fragmentation schema is beneficial, the algorithm still considers the LQPs corresponding to all queries to ensure that improving the fragmentation for one query does not make it worse for another. This is verified by checking that the cost of the most expensive LQP (denoted as $p_{i,\text{max}}^{\text{all}}$) does not increase for any query $q_i \in Q_{\text{all}}$ (as seen in the second part of the if clauses in lines 13, 24, and 31). Once all queries have been eliminated from consideration, the fragmentation algorithm terminates, and the improved fragmentation is returned (line 38).

In the remainder of this section, each individual improvement considered by Algorithm 6 is discussed in detail.

8.2.1 Merging Fragments

To reduce the cost of LQP $p^{\text{todo}}_{\max}$, it is possible to merge the fragment corresponding to this LQP (denoted as $f^{\text{todo}}_{\max}$) with one of its ancestor fragments. While merging two fragments results in a fragment that is larger in size than each of the original fragments, it may nevertheless reduce the cost of the most expensive LQP. This is the case, for example, when constraints placed on nodes in one of the original fragments make it possible to skip some of the data in the other fragment.

To merge fragments, the algorithm proceeds as follows: For each ancestor LQP of $p^{\text{todo}}_{\max}$ (denoted as p_{ancestor}), the corresponding fragment (denoted as f_{ancestor}) is determined (lines 7–8). Then, $f^{\text{todo}}_{\max}$ is merged with f_{ancestor}, resulting in the modified fragmentation schema $F_{\text{aftermerge}}$ (line 10). Next, the algorithm determines whether this decreases the cost of the most expensive LQP corresponding to one of the queries in Q_{todo} without increasing the cost of the most expensive LQP of any query in Q_{all} (lines 11–14).

Special care has to be taken if the fragmentation schema contains intermediate fragments between $f^{\text{todo}}_{\max}$ and f_{ancestor}. In this case, when merging these fragments, the intermediate fragments (determined in line 9 of Algorithm 6 and denoted as $F_{\text{intermediate}}$) have to be included in the merged fragment (line 10).

This procedure is repeated for all ancestor LQPs of $p^{\text{todo}}_{\max}$, and the merge that results in the largest reduction in the cost of the most expensive LQP corresponding to a query in Q_{todo} without making the fragmentation worse for any query in Q_{all} is chosen (lines 13-15).

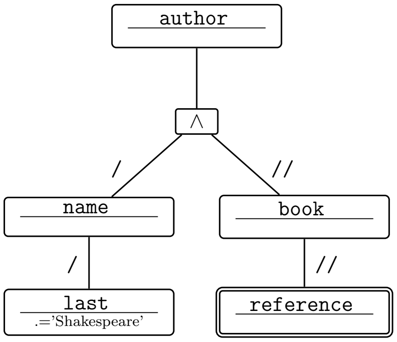


Figure 8.3: QTP representation of query q_{11}

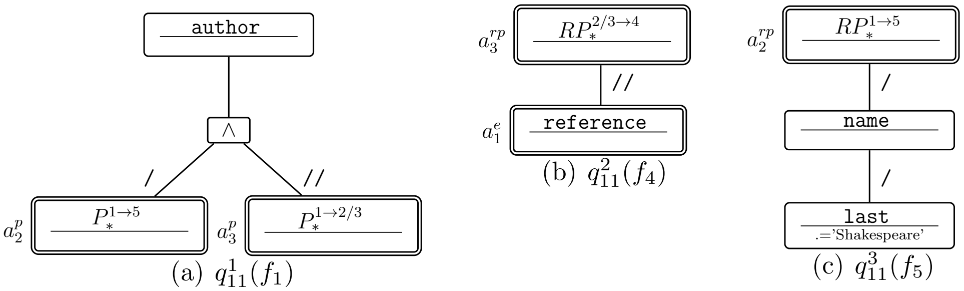


Figure 8.4: Local QTPs corresponding to query q_{11}

If there are no ancestor fragments of $f_{\text{max}}^{\text{todo}}$ for which merging decreases the cost of the most expensive LQP, the algorithm attempts to improve the fragmentation by splitting $f_{\text{max}}^{\text{todo}}$ horizontally, which is described in the next two sections.

To illustrate how the algorithm merges fragments and how this might decrease the cost of the most expensive LQP, consider a query workload that consists of a single query, q_{11} (shown in Figure 8.3). Localizing this query based on the initial fragmentation schema shown in Figure 8.2 and pruning irrelevant fragments yields the local QTPs q_{11}^1, q_{11}^2, and q_{11}^3 (shown in Figure 8.4 and corresponding to fragments f_1, f_4, and f_5, respectively).

Now assume that p_{11}^1, p_{11}^2, and p_{11}^3 are the LQPs corresponding to local QTPs q_{11}^1, q_{11}^2, and q_{11}^3, respectively. Further assume that the estimated costs of these LQPs are as follows:

$$\begin{aligned}\text{cost}(p_{11}^1(f_1)) &= 50\\ \text{cost}(p_{11}^2(f_4)) &= 60\\ \text{cost}(p_{11}^3(f_5)) &= 70\end{aligned}$$

As can be seen, p_{11}^3 is the most expensive LQP. Therefore, for this example, the fragmentation strategy begins by attempting to merge fragment f_5 (the fragment corresponding to this LQP) with its ancestor fragments. Because p_{11}^3 has only a single ancestor plan (p_{11}^1), which corresponds to fragment f_1, the only merge considered is between fragments f_1 and f_5. Since f_1 and f_5 are directly connected in the fragmentation schema, there are no intermediate fragments to include.

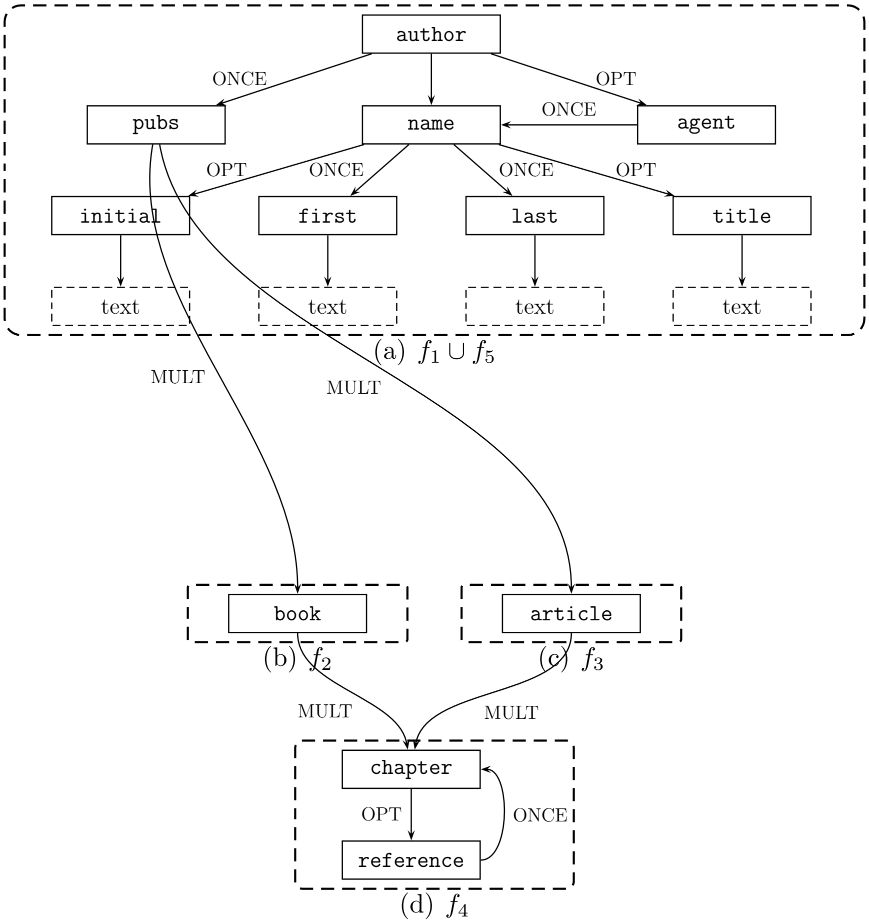


Figure 8.5: Fragmentation schema after merging f_1 and f_5

To see why it might be beneficial to merge these fragments, observe that fragment f_5 consists of sub-trees corresponding to `name` nodes that occur as a direct child of an `author` node and of `name` nodes that occur as the child of an `agent` node. If a navigational strategy is used to evaluate the LQP from Figure 8.6 over the merged fragment, access to the latter type of `name` nodes can be avoided. If a structural join-based strategy is used, the size of intermediate results can be reduced. Thus, regardless of the centralized query evaluation strategy used, merging fragments may result in a performance gain.

Merging fragments f_1 and f_5 results in the fragmentation schema shown in Figure 8.5.

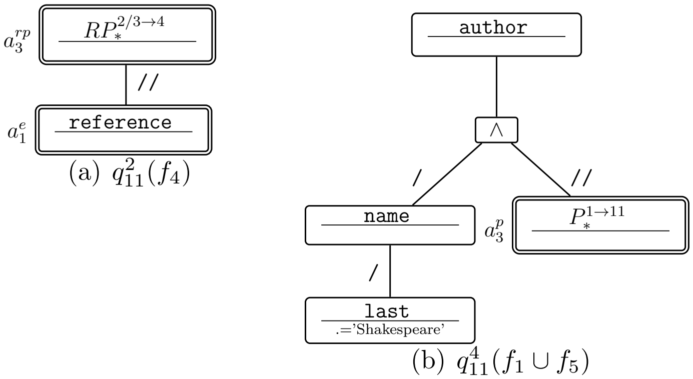


Figure 8.6: Local QTPs corresponding to query q_{11} after merging f_1 and f_5

Localizing query q_{11} results in two LQPs, q_{11}^2 (corresponding to fragment f_4 and unchanged) and q_{11}^4 (corresponding to the newly merged fragment $f_1 \cup f_5$), both of which are shown in Figure 8.6.

To decide whether it is beneficial to merge fragment f_1 with fragment f_5, the cost of each of these LQPs has to be determined. For this example, assume that cost estimates are as follows:

$$\text{cost}(p_{11}^2(f_4)) = 60$$
$$\text{cost}(p_{11}^4(f_1 \cup f_5)) = 55$$

As can be seen, the cost of the most expensive LQP (p_{11}^2) is now 60, and thereby less than the previous cost of 70. Thus, the merge between fragments f_1 and f_5 is beneficial.

8.2.2 Horizontal Fragmentation Based on Node Type Paths

When the cost of the LQP $p_{\max}^{\text{todo}}$ cannot be reduced by merging $f_{\max}^{\text{todo}}$ with one of its ancestor fragments, Algorithm 6 instead attempts to split $f_{\max}^{\text{todo}}$ horizontally based on the root proxy nodes at which sub-trees in this fragment are rooted (lines 18–25). As is described in Section 6.2.2.4, these root proxy nodes can be annotated with node type paths that can be used to filter out sub-trees that are known to be irrelevant for a given query. Algorithm 6 exploits this and splits $f_{\max}^{\text{todo}}$ into a portion that contains the sub-trees whose nodes type paths are compatible with the query ($f_{\max}^{\text{compat}}$, line 19) and a portion that contains the sub-trees whose node type paths are not compatible with the query ($f_{\max}^{\text{incompat}}$, line 20), resulting in a new fragmentation schema (denoted as $F_{\text{aftersplit}}$). As in the previous section, the algorithm then verifies whether this split reduces the cost of the most expensive LQP corresponding to one of the queries in Q_{todo} without making the fragmentation worse for any query in Q_{all} (lines 24–25).

In the example of query q_{11}, after merging fragments f_1 and f_5, LQP p_{11}^2 (corresponding to fragment f_4) is the most expensive LQP. As can be seen in the fragmentation schema shown in Figure 8.5, this fragment consists of sub-trees that occur as the descendants of a `book` node (and whose root proxies therefore contain `book` as part of their node type path) and of sub-trees that occur as the descendants of an `article` node (containing `article` as part of their node type path).

As can be determined using the technique proposed in Section 6.2.2.4, only the sub-trees that contain the node type `book` in their node type path need to be accessed when evaluating LQP p_{11}^2. Thus, it is possible to split fragment f_4 into f_4^{compat} (consisting of the sub-trees in f_4 that contain the node type `book` in their node type path) and f_4^{incompat} (consisting of the sub-trees that contain the node type `article` in their node type path). This yields the fragmentation schema shown in Figure 8.7[1].

Of the new fragments, only fragment f_4^{compat} needs to be accessed to evaluate query q_{11}. Thus, there are still two LQPs. Assume that their costs are as follows:

$$\text{cost}(p_{11}^2(f_4^{\text{compat}})) = 30$$
$$\text{cost}(p_{11}^4(f_1 \cup f_5)) = 55$$

As can be seen, the cost of the most expensive LQP is now 55 and therefore less than

[1] Due to space constraints, the schema of fragments f_4^{compat} and f_4^{incompat} is only shown once even though these fragments are entirely separate.

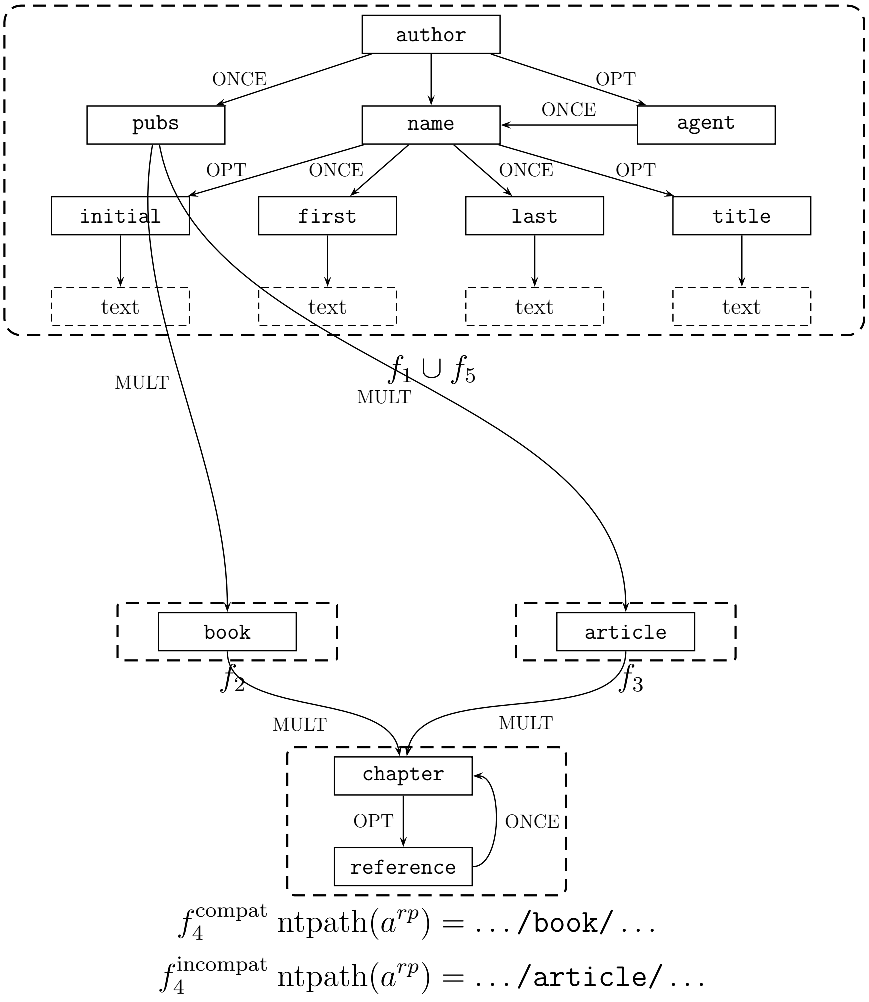
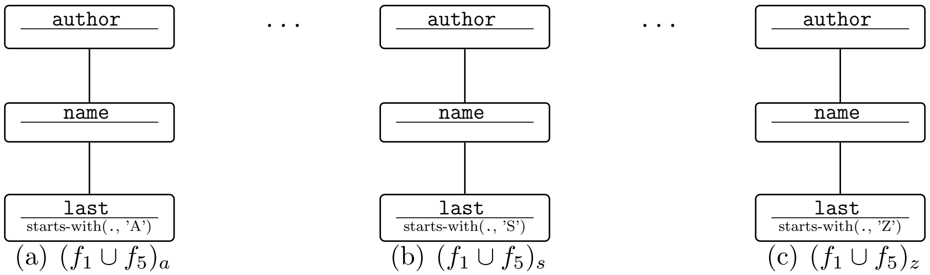

author
ONCE
OPT
pubs
name
ONCE
agent
OPT
ONCE
ONCE
OPT
initial
first
last
title
text
text
text
text
MULT
$f_1 \cup f_5$
MULT
book
article
f_2
f_3
MULT
MULT
chapter
OPT
ONCE
reference
f_4^{compat} $\text{ntpath}(a^{rp}) = \ldots/\texttt{book}/\ldots$
f_4^{incompat} $\text{ntpath}(a^{rp}) = \ldots/\texttt{article}/\ldots$

Figure 8.7: Fragmentation schema after horizontally splitting f_4

the previous cost of 60. Thus, horizontally splitting fragment f_4 into f_4^{compat} and f_4^{incompat} is beneficial.



(a) $(f_1 \cup f_5)_a$ (b) $(f_1 \cup f_5)_s$ (c) $(f_1 \cup f_5)_z$

Figure 8.8: Set of FTPs for horizontally fragmenting $f_1 \cup f_5$

8.2.3 Horizontal Fragmentation Based on Value or Structural Constraints

If neither merging $f^{\text{todo}}_{\text{max}}$ with one of its ancestor fragments nor horizontally splitting $f^{\text{todo}}_{\text{max}}$ based on the node type paths associated with the root proxy nodes in this fragment reduces the cost of the most expensive LQP, Algorithm 6 attempts to split $f^{\text{todo}}_{\text{max}}$ based on value or structural constraints in LQP $p^{\text{todo}}_{\text{max}}$ (lines 26–35).

If $p^{\text{todo}}_{\text{max}}$ places a value constraint on a node of type σ in $f^{\text{todo}}_{\text{max}}$, it is possible to split $f^{\text{todo}}_{\text{max}}$ by partitioning the domain of the content of nodes of type σ (line 27)[2]. For structural constraints that are based on the existence of a node of a particular type, it is similarly possible to partition fragment $f^{\text{todo}}_{\text{max}}$ into sub-trees that do or do not contain nodes of this type.

As in the case of the other two methods for improving a fragmentation schema, the algorithm then verifies whether this partitioning is beneficial (i.e., whether it reduces the cost of the most expensive LQP corresponding to a query in Q_{todo} without making the fragmentation worse for any query in Q_{all}, verified in lines 31–33).

For an example of this, consider the fragmentation schema that has resulted from the improvement steps presented in the previous two sections. LQP p^4_{11}, which corresponds to

[2] The details of how the domain of the content of nodes of a particular type should be partitioned are beyond the scope of this work. This problem is encountered in essentially the same fashion when horizontally fragmenting relational data. Possible solutions include, for example, partitioning based on minterm predicates extracted from the query (cf. [115]).

fragment $f_1 \cup f_5$, is the most expensive LQP. As can be seen in the QTP representation of this LQP shown in Figure 8.6, p_{11}^4 places a value constraint on nodes of the type `last` in fragment $f_1 \cup f_5$, checking whether the content of such nodes is equal to 'Shakespeare'. It is thus possible to partition fragment $f_1 \cup f_5$ based on the content of nodes of this type. A simple way of doing this divides the domain of the content of `last` nodes based on the first character of this content (which is assumed to be a character in the English alphabet). This can be expressed using the FTPs shown in Figure 8.8.

Partitioning fragment $f_1 \cup f_5$ in this fashion yields the fragmentation schema shown in Figure 8.9. Using the horizontal pruning strategy described in Section 6.1.1, it can be determined that only one of the fragments resulting from splitting $f_1 \cup f_5$ needs to be accessed to answer query q_{11} (namely $(f_1 \cup f_5)_s$). Thus, there are still two LQPs. Assume that their costs are estimated as follows:

$$\text{cost}(p_{11}^2(f_4^{\text{compat}})) = 30$$
$$\text{cost}(p_{11}^4((f_1 \cup f_5)_s)) = 20$$

As can be seen, the cost of the most expensive LQP is now 30 (and therefore less than the previous cost of 55). Thus, horizontally splitting $f_1 \cup f_5$ is beneficial.

The most expensive LQP is now p_{11}^2, which is evaluated over fragment f_4^{compat}. Assuming that the cost of this LQP cannot be reduced further, Algorithm 6 terminates and the fragmentation schema shown in Figure 8.9 is returned as the final fragmentation schema.

8.3 Losslessness of resulting fragmentation

An important consideration when decomposing a collection is that the resulting, fragmented collection must contain all the necessary information needed to reconstruct the original collection. In other words, the fragmentation strategy must be lossless This section shows why the fragmentation strategy described in this chapter has this property.

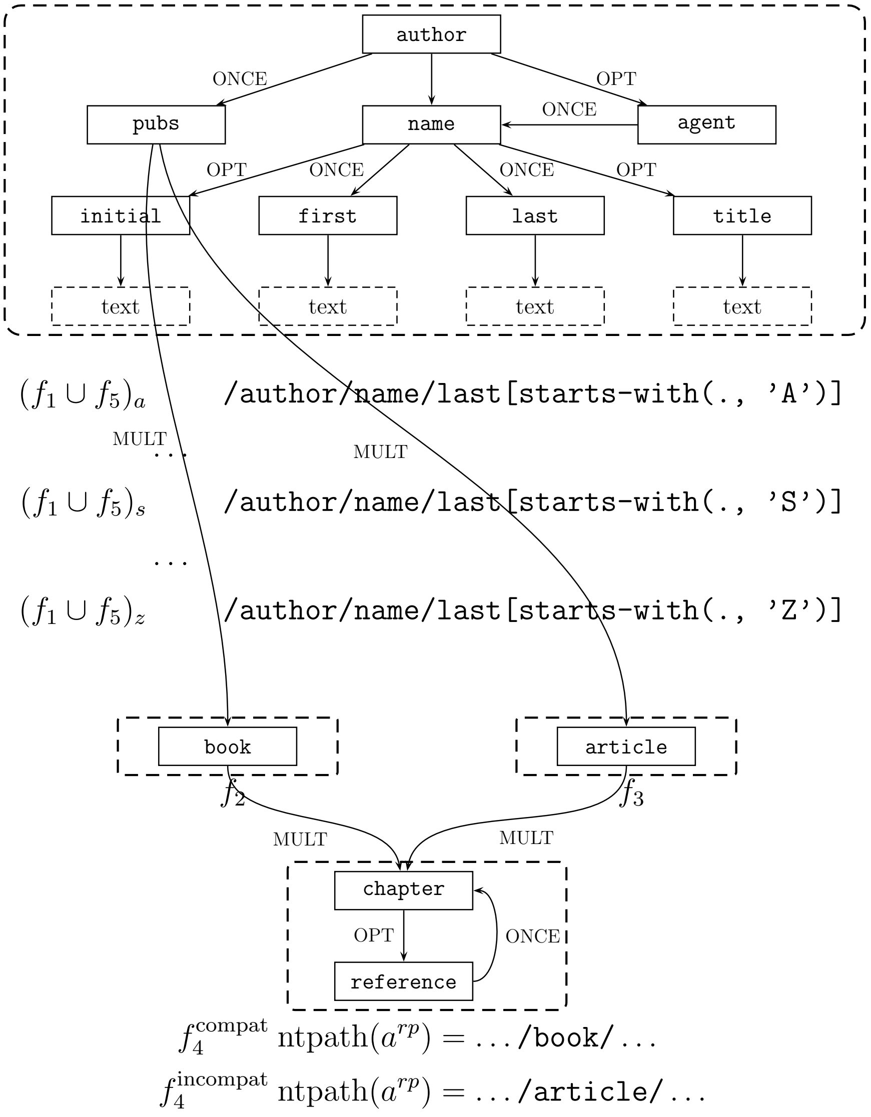

author
ONCE
OPT
pubs
name
ONCE
agent
OPT
ONCE
ONCE
OPT
initial
first
last
title
text
text
text
text
$(f_1 \cup f_5)_a$ /author/name/last[starts-with(., 'A')]
MULT
...
MULT
$(f_1 \cup f_5)_s$ /author/name/last[starts-with(., 'S')]
...
$(f_1 \cup f_5)_z$ /author/name/last[starts-with(., 'Z')]
book
article
f_2
f_3
MULT
MULT
chapter
OPT
ONCE
reference
f_4^{compat} ntpath$(a^{rp}) = \ldots$/book/$\ldots$
f_4^{incompat} ntpath$(a^{rp}) = \ldots$/article/$\ldots$

Figure 8.9: Final fragmentation schema

Fragmenting a collection based on the heuristic strategy described in this chapter results in a vertically fragmented collection (cf. Section 8.2.1), in which some vertical fragments may have been further fragmented horizontally based on node type paths (Section 8.2.2) or based on value or structural constrains (Section 8.2.3).

As is described in Section 4.2, when a collection is fragmented vertically, each edge that is bisected by a fragment boundary is replaced with a pair of special nodes: a proxy node in the originating fragment, and a root proxy node in the target fragment. Both the proxy and the root proxy node corresponding the same edge receive matching identifiers, and these identifiers are required to be unique within the entire collection. In addition, no nodes from the original collection are lost, since each node is required to correspond to exactly one node type in the schema, and each node type in the schema is assigned to exactly one vertical fragment.

To reconstruct the original, unfragmented collection, it is thus possible to find all pairs of matching proxy and root proxy nodes and replace them with a document edge from the parent of the proxy node to the child of the root proxy node (which, as discussed in Section 4.2, is required to be unique).

After vertically fragmenting the collection, the heuristic fragmentation strategy described in this chapter may further fragment some of the resulting vertical fragments horizontally. The first approach for doing this (described in Section 8.2.2), divides a fragment f into two partitions f^{compat} and f^{incompat}, where f^{compat} contains all those sub-trees in f whose node type paths are compatible with some query, and f^{incompat} contains the sub-trees whose node type paths are incompatible with the query. Since each sub-tree in f falls into exactly one of these categories (and is thus placed in exactly one of f^{compat} or f^{incompat}), it is possible to reconstruct f by simply merging f^{compat} and f^{incompat}.

In addition to horizontal fragmentation based on node type paths, the fragmentation strategy presented in this chapter also performs horizontal fragmentation based on structural or value constraints. While the specifics of how this is done are outside the scope of this thesis, the fragmentation strategy requires that each horizontal fragmentation step performed results in a partitioning of the set of sub-trees in a fragment. Relational fragmentation techniques, such as horizontal fragmentation based on minterm predicates, which can be applied in this case, have the property of generating a partitioning [115]. This ensures that the original fragment can always be reconstructed by merging the resulting partitions.

In summary, it can be observed that each of the fragmentation steps performed by the fragmentation strategy presented in this chapter can be reversed. From this follows that the original, unfragmented collection can be reconstructed and that, therefore, the fragmentation is lossless.

Chapter 9

Performance Evaluation

The previous chapters have introduced a suite of techniques that are designed to improve the scalability of XML query processing by distributing both the collection and the query processing workload across multiple sites in a distributed system. This chapter presents a thorough experimental evaluation that demonstrates that these techniques yield a significant improvement in scalability and performance. In addition, the various query evaluation techniques presented in the earlier chapters are examined individually, which provides valuable insight into each technique's individual contribution to the overall performance improvement and into the problem settings for which a given technique is best suited.

To allow extensive experimentation in a realistic scenario, the native XML database system Natix [49] was extended with the features that are necessary to support distributed query execution, including primitives for passing (sub-)queries between instances and operators for sending and receiving tuples over the network. Then, the query evaluation techniques presented in this thesis were implemented within this framework.

The distributed version of Natix was then deployed on virtualized Linux instances within Amazon's Elastic Compute Cloud (EC2) [1]. EC2 provides a variety of instance types with different levels of CPU, memory, and I/O capacity. For all of the experiments presented in this chapter, "small" instances were chosen, which provide 1.7 GB of memory, a single-core CPU, and 160 GB of local instance storage. In addition to instance-local storage, EC2 also provides network-attached storage that can be shared between instances. This feature was not used for the experiments presented here.

To ensure that the conditions of the experiments represented here reflect the conditions encountered in a data centre, all instances were allocated within the same availability zone. This yields low-latency, high-throughput communication between instances.

All of the experiments presented here rely on a collection of on-line auction data generated by the XMark benchmark [120], which is a standard, widely used benchmark for evaluating XML query performance. The XMark data generator can be configured to produce collections of any size, which has made it possible to study scalability across a range of collection sizes.

To obtain a realistic query workload that can be evaluated over the XMark collections, queries from the performance-oriented portion[1] of the XPathMark benchmark[2] [53] were selected. These queries represent realistic use cases for XPath and, when evaluated over the XMark data, make it possible to evaluate the distributed techniques presented in this thesis in a realistic use case. To augment the results obtained with the XPathMark queries, additional, synthetic queries were crafted that stress-test the behaviour of the various query evaluation techniques introduced in this thesis.

A detailed discussion of the performance evaluation is presented in the remainder of this chapter, organized as follows:

- The first set of experiments, presented in Section 9.1, focuses on examining the overall performance and scalability effect of a combination of all query evaluation techniques presented in this thesis. To this end, an XMark collection is fragmented based on the heuristic fragmentation technique presented in Chapter 8, resulting in a hybrid fragmentation that uses both vertical and horizontal fragmentation steps. Then, for

[1] In addition to queries that are designed to evaluate query performance, the XPathMark benchmark provides a second set of queries (referred to as the functional benchmark) that can be used to verify that a query processor supports a wide range of XPath features.

[2] Note that the XMark benchmark also provides a set of sample queries. However, unlike the queries in the XPathMark benchmark, these queries are expressed as FLWOR expressions and make use of features that cannot be expressed in the XQ query model.

each of the XPathMark queries, the best DEP is determined using the cost-based optimization technique described in Chapter 7. The performance obtained by this plan is then compared to the performance of centralized query execution for various collection sizes. Additionally, the performance of several existing distributed query evaluation techniques is examined for comparison.

- In Section 9.2, query evaluation over a horizontally distributed collection is examined more closely. The definition of horizontal fragmentation allows a collection to be partitioned into an arbitrary number of fragments. Thus, the experimental evaluation of horizontal distribution focuses on scalability, both in terms of throughput and query response time. The impact of data skew in the distribution is also examined. For all experiments in this section, particular attention is paid to the impact of the horizontal pruning techniques presented in Section 6.1.1.
- Section 9.3 similarly focuses on vertical distribution. In addition to pruning techniques, whose performance characteristics are examined in detail in Section 9.3.1, the impact of the various techniques for improving distributed execution plans (e.g., pushing of cross-fragment joins) is examined closely in Section 9.3.2.
- Finally, Section 9.4 presents an evaluation of the cost model introduced in Chapter 7. Here, attention is paid to how well the estimated cost of a DEP corresponds to the actual response time cost of this DEP and how effectively cost-based optimization identifies the plan with the lowest actual response time cost.

9.1 Full Suite of Techniques

This section examines the overall performance and scalability impact of applying the complete suite of distributed query evaluation techniques presented in this thesis. Table 9.1 shows the 10 XPathMark queries that are supported by the XQ query model. Each of these queries is evaluated over the collections generated by the XMark benchmark. These collections consist of multiple documents, each of which is approximately 40 MB in size. The total size of the collection (and thereby the number of documents contained in the

A1	`/site/closed_auctions/closed_auction/annotation/description/text/keyword`
A2	`//closed_auction//keyword`
A3	`/site/closed_auctions/closed_auction//keyword`
A4	`/site/closed_auctions/closed_auction[annotation/description/text/keyword] /date`
A5	`/site/closed_auctions/closed_auction[descendant::keyword]/date`
A6	`/site/people/person[profile/gender and profile/age]/name`
A7	`/site/people/person[phone or homepage]/name`
A8	`/site/people/person[address and (phone or homepage) and (creditcard or profile)]/name`
B7	`//person[profile/@income]/name`
C1	`/site/people/person[profile/age >= 18 and profile/@income < 10000 and address/city != 'Dallas']/name`

Table 9.1: XPathMark queries

collection) is scaled to three sizes: 120 MB (corresponding to a scale factor of 1), 1.2 GB (corresponding to scale factor 10), and 12 GB (corresponding to scale factor 100).

Before the distributed query evaluation techniques introduced in this thesis can be applied, it is first necessary to fragment and distribute the XMark collections. This is done by applying the heuristic fragmentation technique described in Chapter 8. This results in the hybrid fragmentation schema shown in Figure 9.1[3].

As can be seen, the heuristic fragmentation strategy first divides the XMark data into a set of vertical fragments. Three of the vertical fragments are then further fragmented horizontally. Two different types of horizontal fragmentation are employed. Two of the vertical fragments are horizontally fragmented based on structural constraints present in the queries. In the case of the vertical fragment consisting of the node types `annotation`, `description`, `parlist`, `listitem`, and `text`, three horizontal sub-fragments (labeled f_{11}, f_{12}, and f_{13}) are defined. These horizontal fragments correspond to the sub-trees that have

[3] To increase clarity, this representation is simplified slightly by omitting node types that are not relevant for the XPathMark queries.

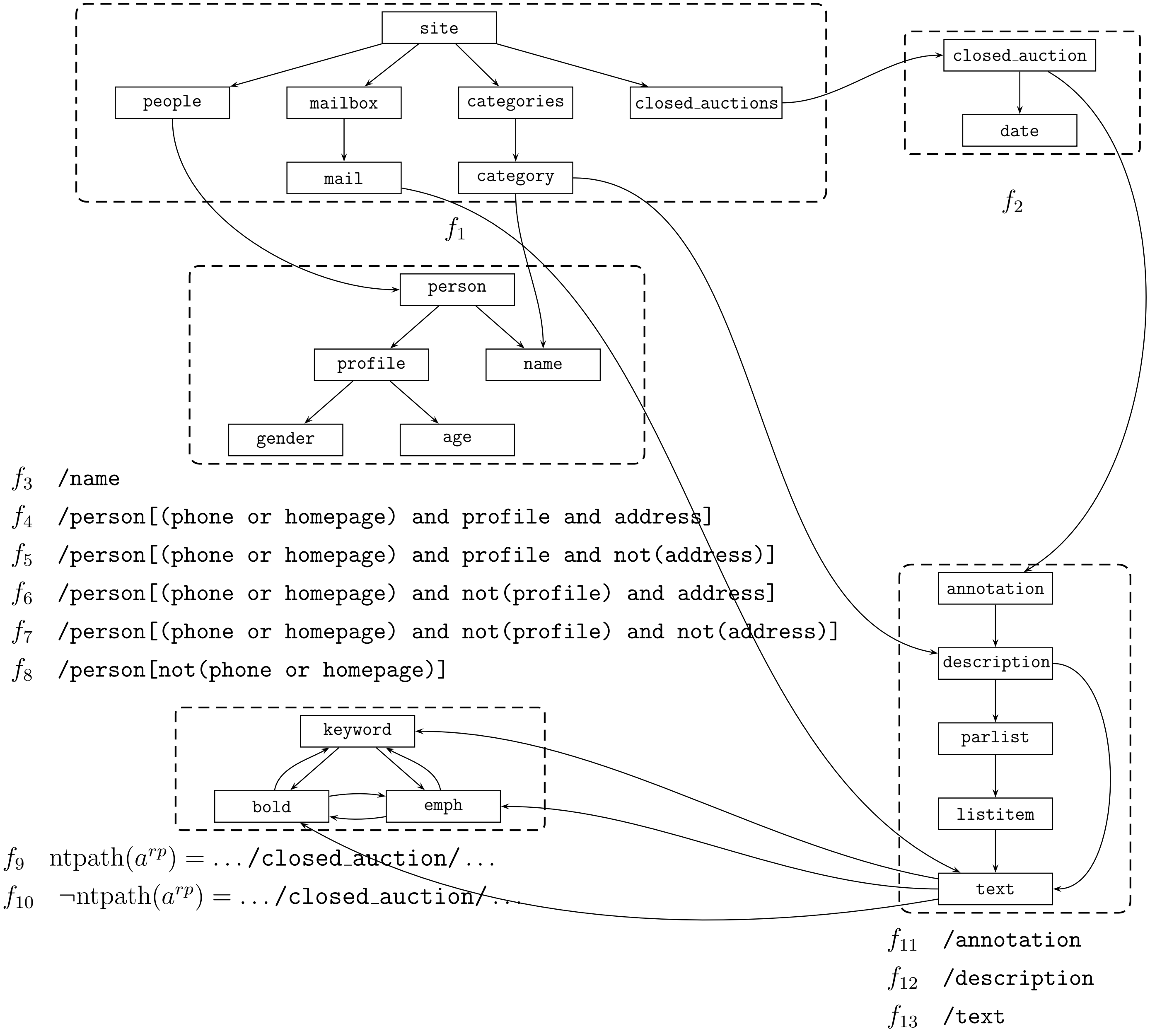

site
people
mailbox
categories
closed_auctions
mail
category
f_1
closed_auction
date
f_2
person
profile
name
gender
age
f_3 /name
f_4 /person[(phone or homepage) and profile and address]
f_5 /person[(phone or homepage) and profile and not(address)]
f_6 /person[(phone or homepage) and not(profile) and address]
f_7 /person[(phone or homepage) and not(profile) and not(address)]
f_8 /person[not(phone or homepage)]
keyword
bold
emph
f_9 ntpath(a^{rp}) = .../closed_auction/...
f_{10} ¬ntpath(a^{rp}) = .../closed_auction/...
annotation
description
parlist
listitem
text
f_{11} /annotation
f_{12} /description
f_{13} /text

Figure 9.1: Hybrid fragmentation schema obtained using heuristics

nodes of types `annotation`, `description`, or `text` as the immediate child of their root proxy nodes, respectively. Fragments f_9 and f_{10}, in contrast, result from the horizontal fragmentation of a vertical fragment based on the node type path associated with the root proxy node of each sub-tree. Fragment f_9 consists of sub-trees whose root proxy node has a node type path that contains the node type `closed_auction`, whereas fragment f_{10} contains the sub-trees whose root proxy node has a node type path that does not contain this node type.

As with all experiments presented in this chapter, each fragment is then loaded onto a separate EC2 instance, thus distributing the collection across the machines used in the experiment.

9.1.1 Scalability and Performance Impact

To evaluate the scalability and performance impact of the distributed query evaluation techniques presented in this thesis, the performance of centralized query evaluation collection is compared to the performance of distributed query evaluation. For this experiment, the collection is scaled to three different sizes: 120 MB (corresponding to three 40 MB documents, and an XMark scale factor of 1), 1.2 GB (corresponding to 30 such documents and a scale factor of 10) and 12 GB (corresponding to 300 documents and a scale factor of 100).

To measure the performance achieved by centralized query evaluation, the unfragmented collection is loaded onto a single EC2 instance. For distributed query evaluation, the collection is partitioned into 13 fragments according to the fragmentation schema shown in Figure 9.1 and each fragment is loaded onto a separate machine. The cost-based optimization technique from Chapter 7 is used to obtain a distributed execution plan for each query, and the resulting plan is then evaluated over all of the instances.

The results of this experiment are shown in Figures 9.2, 9.3, and 9.4. For each query and collection size, the end-to-end response time of centralized query evaluation over an unfragmented collection (denoted as "central") and the end-to-end response time of distributed query evaluation using the techniques presented in this thesis (denoted as "dist optimized") are shown. As with all results presented in this thesis, the response times reported in Figures 9.2, 9.3, and 9.4 represent averages over at least five runs. As can be seen, distributed query evaluation performs significantly better than centralized query evaluation for all queries and collection sizes.

Table 9.2 shows the speed-up factor achieved by distributed query evaluation over centralized query evaluation for each query and collection size. In addition, the average speed-up of all queries is shown for each collection size.

It can be seen that the performance advantage of the distributed technique increases with collection size for all queries. This can be explained by the fact that, at the smaller collection sizes, some of the performance benefit obtained by parallelizing query execution across multiple sites and evaluating LQPs over smaller portions of the overall collection is compensated for by the overhead of distribution. At the largest collection size, however, the positive impact of parallelism and limiting the scope of local query evaluation to a

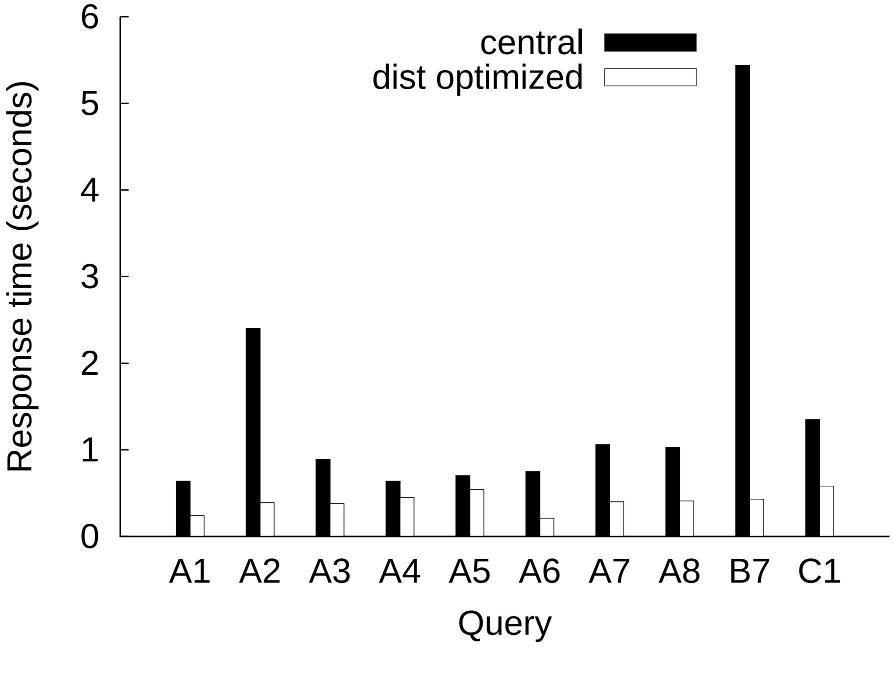


Figure 9.2: Centralized vs. distributed query evaluation, 120 MB

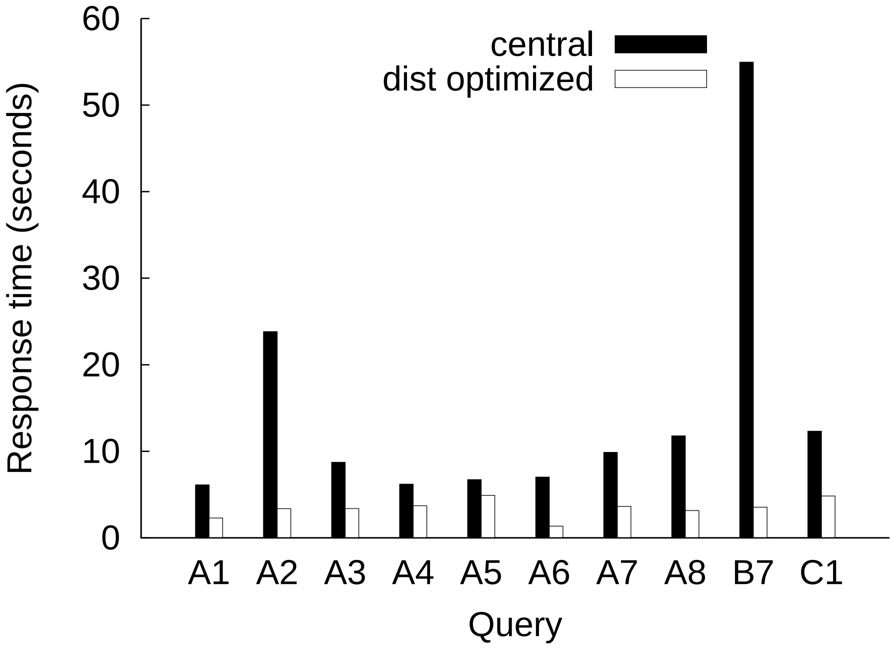


Figure 9.3: Centralized vs. distributed query evaluation, 1.2 GB

subset of the collection (resulting in reduced memory pressure) clearly dominates. This illustrates the superior scalability of distributed query evaluation.

For the largest collection, distributed query evaluation is more than 43 times faster in the best case (query A6). Even for the query with the least pronounced benefit of distributed evaluation (query A5), a more than 11-fold performance improvement is achieved.

Together, these results corroborate the usefulness of the distributed query evaluation

Collection size	Speed-up										
	A1	A2	A3	A4	A5	A6	A7	A8	B7	C1	Average
120 MB	2.62	6.24	2.32	1.44	1.30	3.61	2.61	2.53	12.61	2.35	3.76
1.2 GB	2.66	7.08	2.58	1.67	1.37	5.21	2.72	3.73	15.54	2.55	4.51
12 GB	24.18	21.954	17.33	15.27	11.45	43.42	16.17	19.18	29.85	12.50	21.09

Table 9.2: Speed-up factor of distributed query evaluation over centralized query evaluation

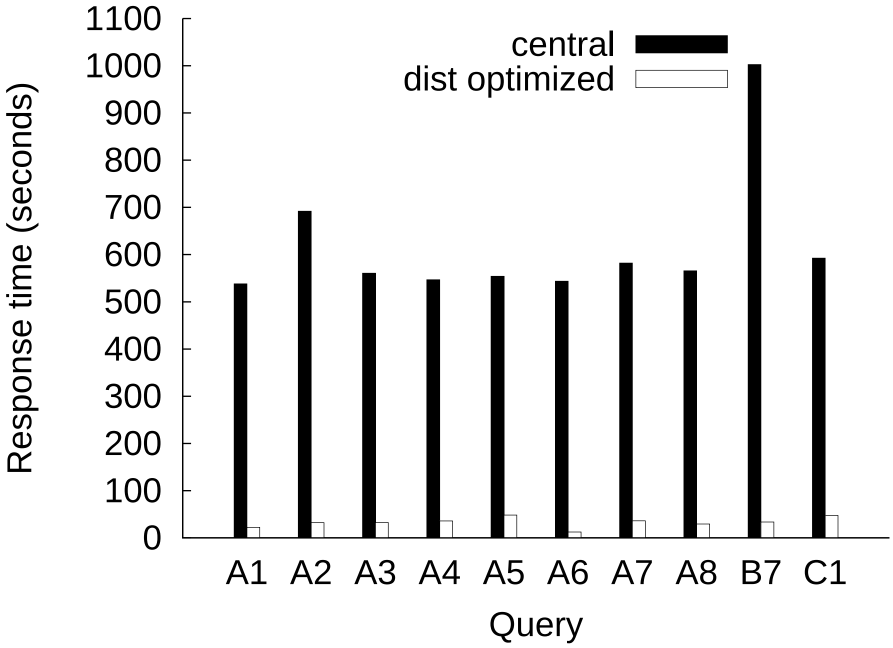


Figure 9.4: Centralized vs. distributed query evaluation, 12 GB

techniques presented in this thesis for improving query performance and scalability. They also show that the cost-based optimization technique presented in Chapter 7 is effective in determining a plan that yields a significant performance benefit. The validity of the heuristic fragmentation strategy from Chapter 8 is also supported by these results since fragmenting the collection based on this strategy has made it possible to apply the distributed query evaluation techniques and to obtain the performance benefit shown.

9.1.2 Comparison With Existing Distributed Query Evaluation Techniques

While much of the existing work in the area of distributed query evaluation over XML collections either focuses primarily on data integration (e.g., [6, 43, 2]) or relies heavily on a replicated index structure (e.g., [31]), there are two techniques that follow a performance motivation that is similar to the one followed in this thesis: Cong et al.'s technique for distributed query evaluation [39] and Suciu's query evaluation technique for semistructured data [124]. While both papers use a definition of performance that is somewhat different from the one used in this work (focusing primarily on communication cost rather than end-to-end response time), they are nevertheless the most appropriate candidates for a direct comparison.

Cong et al. present two multi-phase algorithms for distributed query evaluation, named PaX3 and PaX2. Both algorithms feature a phase during which all fragments are traversed in parallel (phase 2 in PaX3 and phase 1 in PaX2). Since all fragments are traversed in their entirety during this phase, it is reasonable to suspect that this phase dominates the overall response time of their technique. Therefore, for this comparison, the traversal phase has been implemented within Natix.

Figure 9.5 shows the response time of executing this traversal over those fragments of the 12 GB collection that remain after applying their simple pruning strategy (denoted as "PaX"). While this does not capture the total response time cost of evaluating PaX3 or PaX2, the traversal is a necessary step for either algorithm that cannot be avoided or parallelized with other phases. Therefore, the time consumed by this parallel traversal can serve as a lower bound on the overall response time of PaX3 and PaX2. Note that due to the slightly more restrictive query model, query C1 cannot be supported by these algorithms and, therefore, no response time is shown for this query.

For Suciu's distributed evaluation algorithm, a similar insight is exploited: while the paper does not give any experimental results, the response time cost of applying this technique appears to be dominated by the generation of partial results using an automaton that accepts the query. Unlike this work, Suciu's technique does not take advantage of a fragmentation specification. Therefore, the starting state of the automaton at a given root proxy node cannot be determined and all states have to be examined, increasing the processing cost of this phase.

The partial result generation phase of Suciu's algorithm was implemented within Natix and its response time is reported as "disteval" in Figure 9.5. As in the case of Cong et al.'s work, this phase is not parallelized with other phases of the algorithm, and it cannot

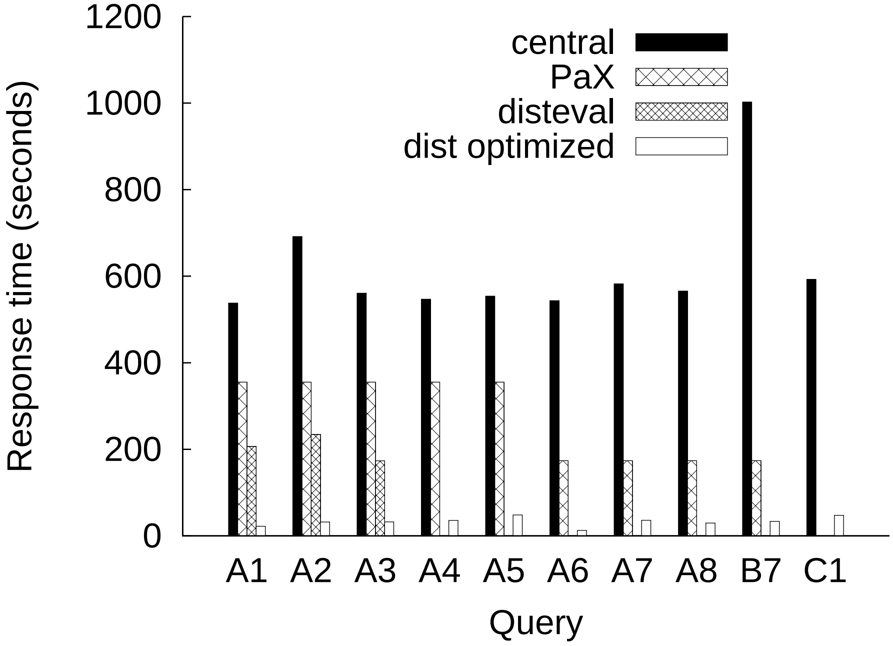


Figure 9.5: Comparison to existing distributed techniques, 12 GB

be avoided. Thus, its response time can serve as a lower bound on the total response time of Suciu's algorithm. The query model used in Suciu's paper is somewhat different from the XPath-based models seen in more recent work and appears to support only linear path queries. Therefore it can be applied only to the linear queries A1, A2, and A3, and only for these queries are response times shown in Figure 9.5.

Comparing the lower bounds on the cost of these existing techniques with the total cost of centralized query evaluation (denoted as "central") and distributed query evaluation using the techniques introduced in this thesis (denoted as "dist optimized") leads to the following observations:

- Both Cong et al.'s and Suciu's algorithms significantly improve response time compared to centralized query evaluation. For the queries supported by both techniques, the partial result generation phase of Suciu's algorithm is faster than the traversal phase of Cong et al.'s technique. This illustrates that even though these techniques are primarily designed to minimize communication cost (rather than overall response time), applying them may yield an overall performance benefit when compared to centralized evaluation[4].

- However, the results confirm that the techniques described in this thesis lead to significantly lower response times for all queries considered.

Overall, the experiment confirms that the techniques presented in this work successfully improve the scalability of distributed query evaluation. While both Cong's and Suciu's technique offer impressive guarantees with regard to communication cost, the results of the experiment show that when optimizing for end-to-end performance, the techniques presented in this thesis, which are specifically designed for this purpose, yield significantly better results.

9.2 Techniques for Horizontal Fragmentation

This section presents an in-depth evaluation of the query evaluation techniques that can be applied to horizontally fragmented collections. The goal of this evaluation is twofold: First, the usefulness of the horizontal fragmentation model for improving query performance is examined. Then, the impact of the pruning techniques presented in Section 6.1.1 is studied. Since the impact of pruning is primarily on query throughput (rather than on the response time of an individual query), both response time and throughput rates are measured.

Since the definition of horizontal fragmentation assumes a multiple-document collection, these experiments are conducted using an XMark collection that has been decomposed into multiple small documents by placing each `open_auction` node into its own document along with its descendants and document sub-trees referenced via ID/IDREF. This results

[4] Since lower bounds on the performance of Cong et al.'s and Suciu's algorithms are considered, a definite comparison between the performance of these algorithms and that of centralized query evaluation is not possible based on the results reported here (nor is such a comparison within the scope of this experimental evaluation).

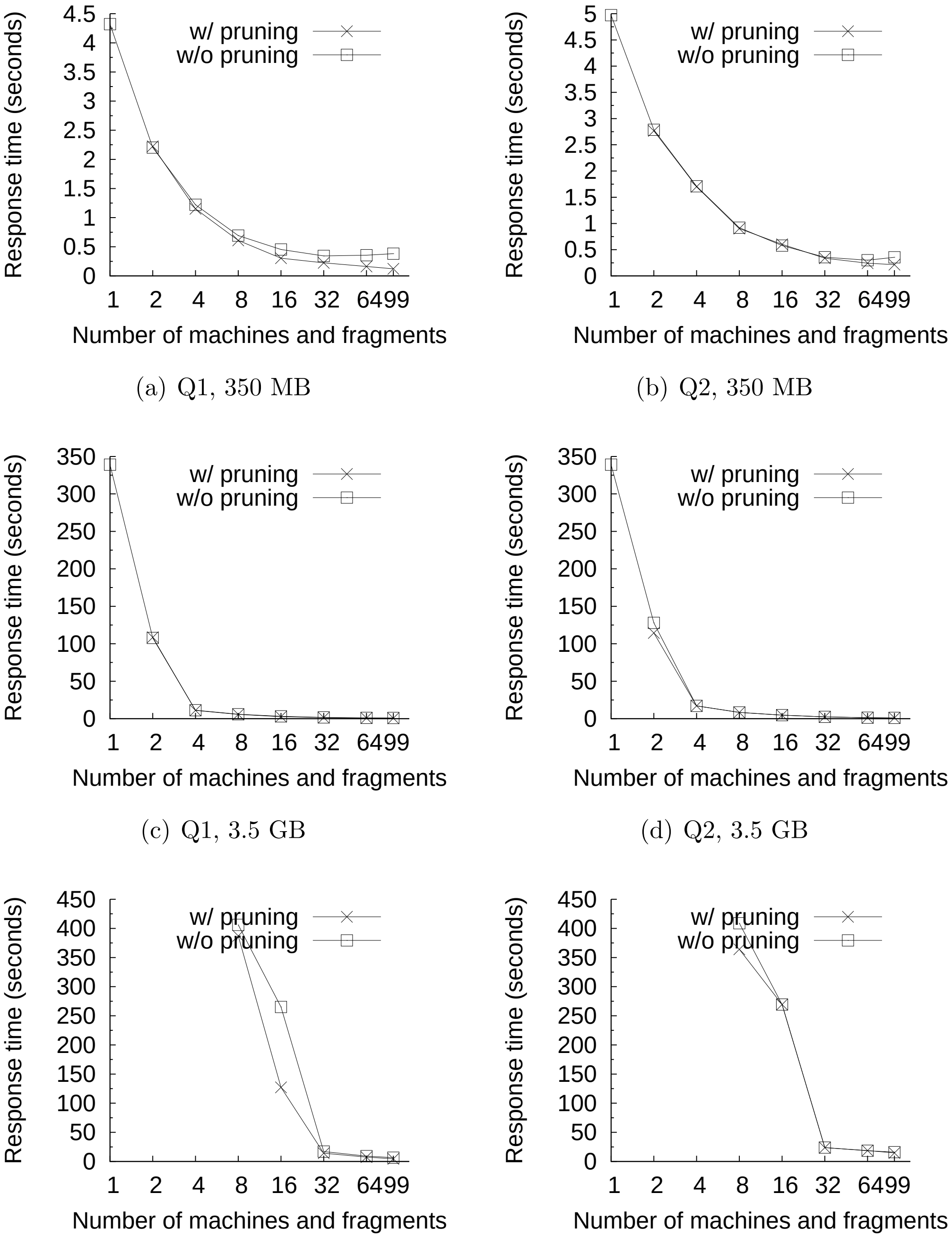
in documents of regular structure with an average size of approximately 30 KB. Three collection sizes are used: 350 MB, 3.5 GB, and 35 GB. Since the decomposition of the collection increases the size by a factor of about three (as some nodes are duplicated), the collections used in this experiment correspond to the same data as the collections used in the previous experiments.

9.2.1 Balanced Fragmentation

Each `open_auction` element generated by XMark contains an auction end date and these dates are uniformly distributed across the years 1998-2001. It is therefore possible to obtain a balanced horizontal fragmentation schema (i.e., a fragmentation schema in which all fragments are approximately the same size) by dividing this date range into non-overlapping periods of equal length, with each such period corresponding to one horizontal fragment. For this experiment, fragmentation schemas consisting of 1, 2, 4, 8, 16, 32, 64 and 99 horizontal fragments are used[5].

After fragmenting the collection in this fashion, five classes of queries are evaluated. These classes of queries have been chosen to illustrate the behaviour of the distributed query evaluation techniques in different scenarios. Q1 consists of queries that contain a point predicate on the auction end date, i.e., each query returns auctions that end on exactly one date within the 4 year period. Q2–Q5 represent range queries that cover 25%, 50%, 75%, and 100% of the four-year date range, respectively. These queries correspond to different scenarios for the horizontal pruning algorithm: whereas Q1 can be answered using a single fragment, Q2-Q5 need to access an increasingly large fraction of all fragments. Thus, Q1 is a good fit for this fragmentation and Q5 is an extremely poor fit. It is important to note that each time a query in one of these classes is evaluated, a date (in the case of class Q1) or a date range of the appropriate length within the 4-year range (in the case of queries Q2-Q5) is chosen randomly. For the purpose of illustration, Table 9.3 shows an example of a query in each class.

[5] The experimental set-up is limited to 100 EC2 instances running simultaneously. Since one such instance is needed for the query dispatcher, this means that at most 99 instances can be used to store fragments.



(a) Q1, 350 MB

(b) Q2, 350 MB

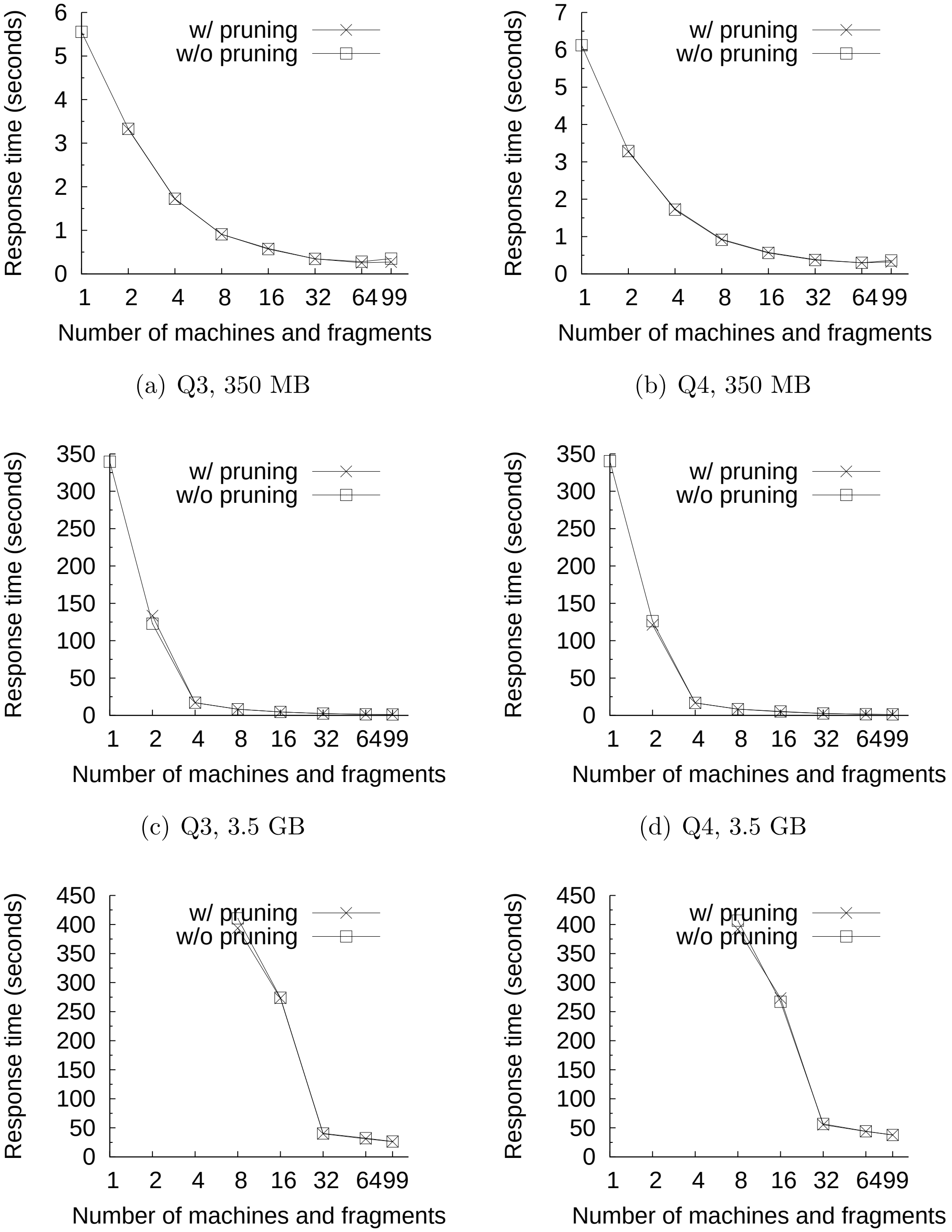
(c) Q1, 3.5 GB

(d) Q2, 3.5 GB

(e) Q1, 35 GB

(f) Q2, 35 GB

Figure 9.6: Response time, balanced horizontal fragmentation



(a) Q3, 350 MB

(b) Q4, 350 MB

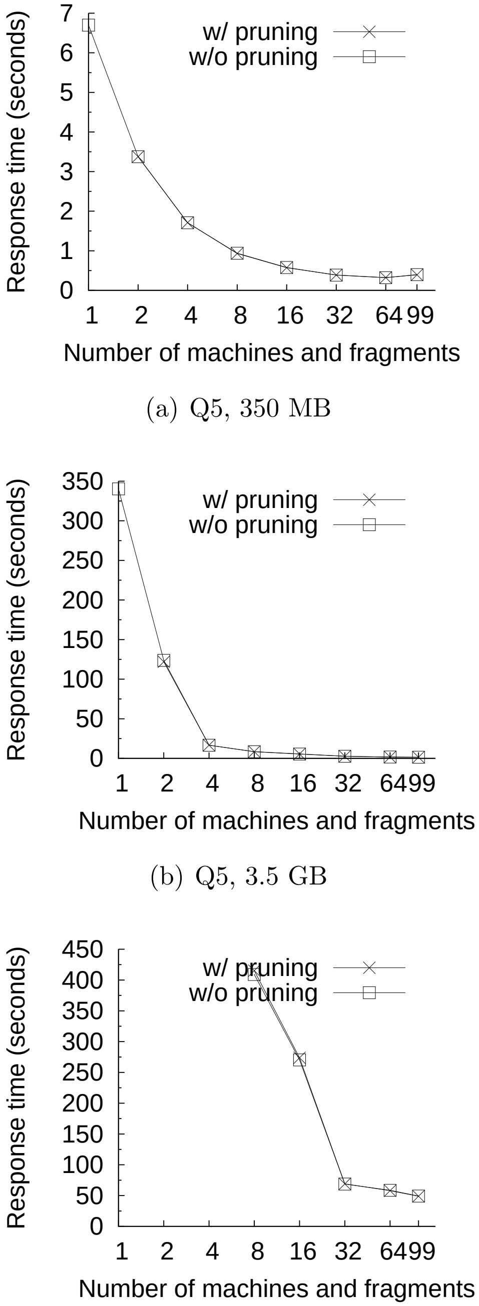
(c) Q3, 3.5 GB

(d) Q4, 3.5 GB

(e) Q3, 35 GB

(f) Q4, 35 GB

Figure 9.7: Response time, balanced horizontal fragmentation (cont'd)



(a) Q5, 350 MB

(b) Q5, 3.5 GB

(c) Q5, 35 GB

Figure 9.8: Response time, balanced horizontal fragmentation (cont'd)

Q1	/open_auction[./interval/end[. = xs:date(’12/28/2001’)]] [initial > 120]//item/name
Q2	/open_auction[./interval/end [. >= xs:date(’01/01/1998’)][. < xs:date(’12/28/1998’)]] [initial > 120]//item/name
Q3	/open_auction[./interval/end [. >= xs:date(’01/01/1998’)][. < xs:date(’12/28/1999’)]] [initial > 120]//item/name
Q4	/open_auction[./interval/end [. >= xs:date(’01/01/1998’)][. < xs:date(’12/28/2000’)]] [initial > 120]//item/name
Q5	/open_auction[./interval/end [. >= xs:date(’01/01/1998’)][. < xs:date(’12/28/2001’)]] [initial > 120]//item/name

Table 9.3: Queries used in horizontal experiments

First, the response time of evaluating each query over the horizontally distributed collection is measured. As in all measurements in this chapter, the results reported in Figures 9.6, 9.7, and 9.8 include the cost of constructing sub-query results at the individual sites, shipping them to the dispatcher and assembling them to the overall query result[6]. In the case of the 35 GB collection, some data points are missing for centralized execution and the fragmentation schemas with a lower number of fragments. In these cases, query evaluation did not finish within the allotted maximum of two hours.

When interpreting the results, it can be seen that even without pruning, horizontal distribution reduces query response time when compared with centralized execution (i.e., the scenario with a single fragment on a single machine). For all queries and collection sizes, response time decreases as the collection is partitioned into an increasingly large number of fragments (and thereby distributed across an increasingly large number of machines). This can be explained by the fact that fragment sizes decrease when a collection of constant size

[6] Note that a logarithmic scale is used on the x-axis.

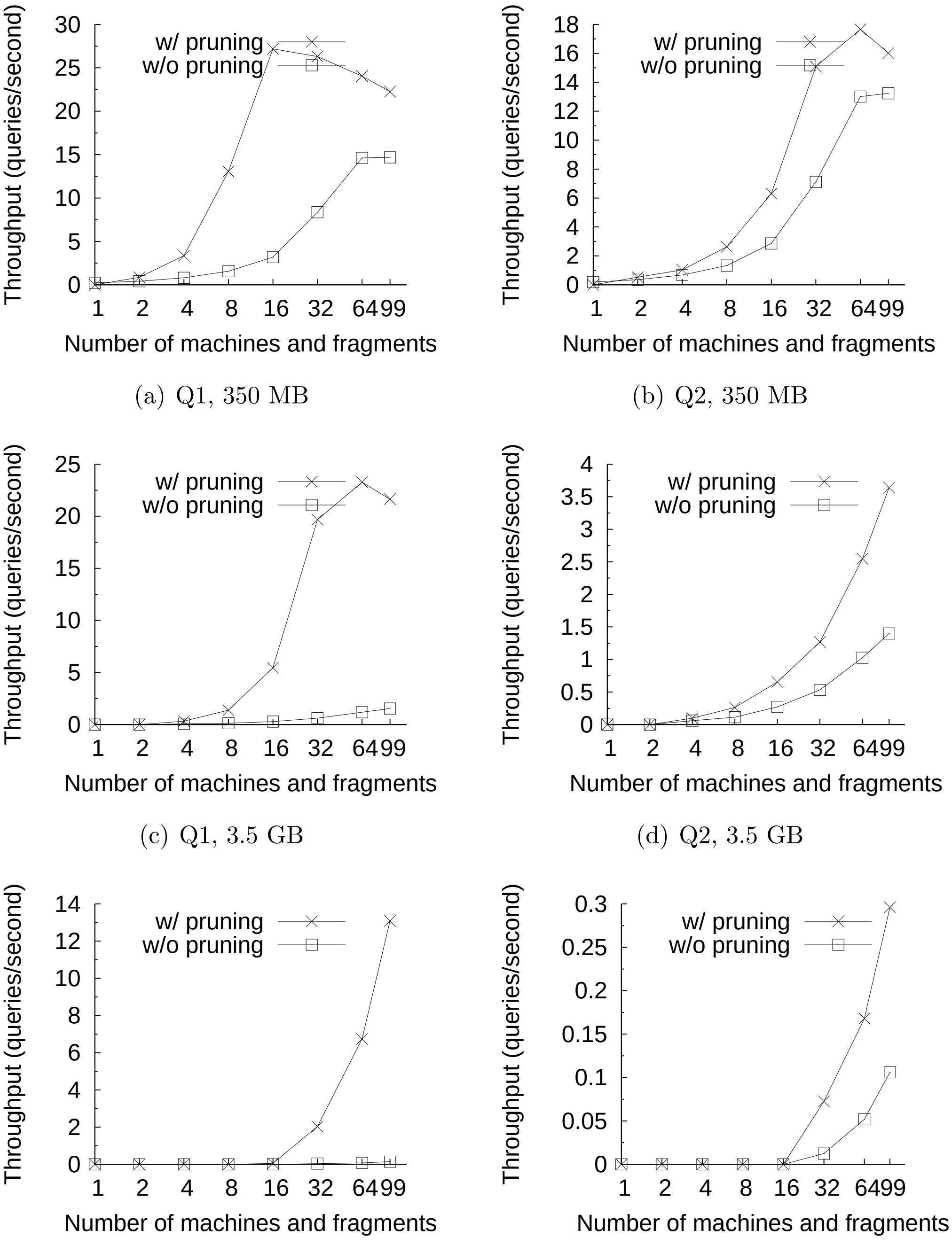
is partitioned into a larger number of fragments. At the same time, the query is decomposed into a larger number of sub-queries, which are then evaluated in parallel. Each of these sub-queries accesses a smaller amount of data (corresponding to the smaller fragment size), thereby reducing the overall response time of the query.

Based on the same reasoning, increasing the number of machines makes it possible to query a larger collection while maintaining the same level of response time. For example, assume that a response time of less than 100 seconds is desired for each of the five classes of queries. As can be seen, for the 3.5 GB collection, this can be achieved by partitioning the collection into at least four fragments. To ensure the same response time for the 35 GB collection, this collection needs to be partitioned into at least 32 fragments.

When considering the impact of pruning, it can be observed that this technique does not result in a major improvement of response time when compared to distributed execution without pruning. This is expected since pruning is primarily intended to improve throughput. It is important, however, to point out that pruning has no negative impact on response time.

Next, the impact of distribution and pruning on throughput is considered. To measure query throughput, multiple dispatcher processes are used to keep the system saturated with queries. In Figures 9.9, 9.10, and 9.11, the maximum throughput rates achieved for each class of queries are reported. Even without pruning, distribution significantly increases throughput and this increase is proportional to the number of fragments. Enabling pruning further improves throughput by a significant margin. Naturally, the impact of pruning is most pronounced for the class of point queries (Q1), where a single date is selected and where the pruning technique can therefore avoid accessing all but one of the fragments for each query. Pruning also helps for the queries that involve a range of dates, particularly when this range is small (i.e., Q2 and Q3), though the effect is less pronounced. For Q4 and Q5, where a large portion of the fragments or all fragments have to be inspected, pruning offers no advantage over simple distribution but it also does not harm performance (apart from some insignificant anomalies in the case of the 35 GB collection where throughput rates are very low).

This illustrates the importance of a fragmentation schema that is well suited to the



(a) Q1, 350 MB

(b) Q2, 350 MB

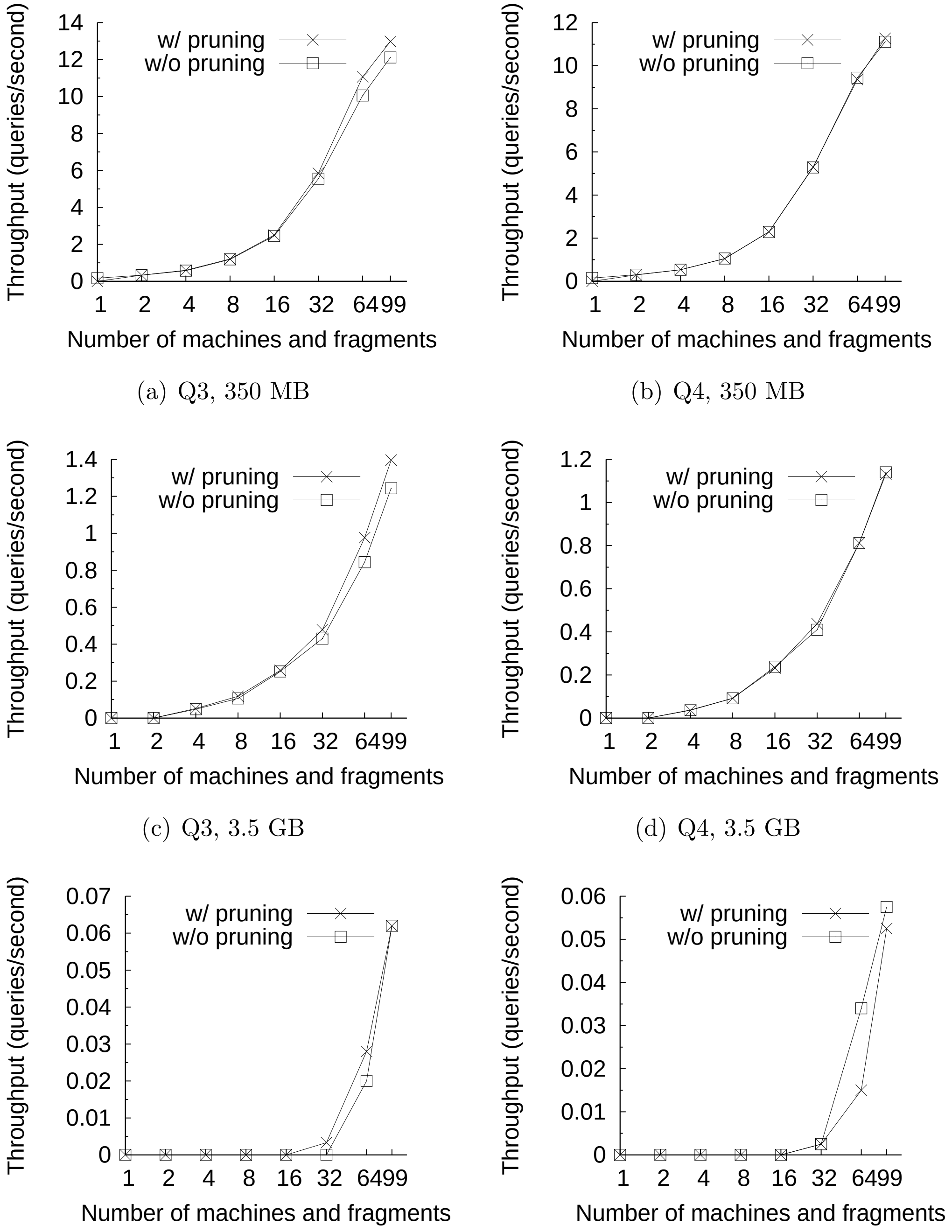
(c) Q1, 3.5 GB

(d) Q2, 3.5 GB

(e) Q1, 35 GB

(f) Q2, 35 GB

Figure 9.9: Throughput, balanced horizontal fragmentation



(a) Q3, 350 MB

(b) Q4, 350 MB

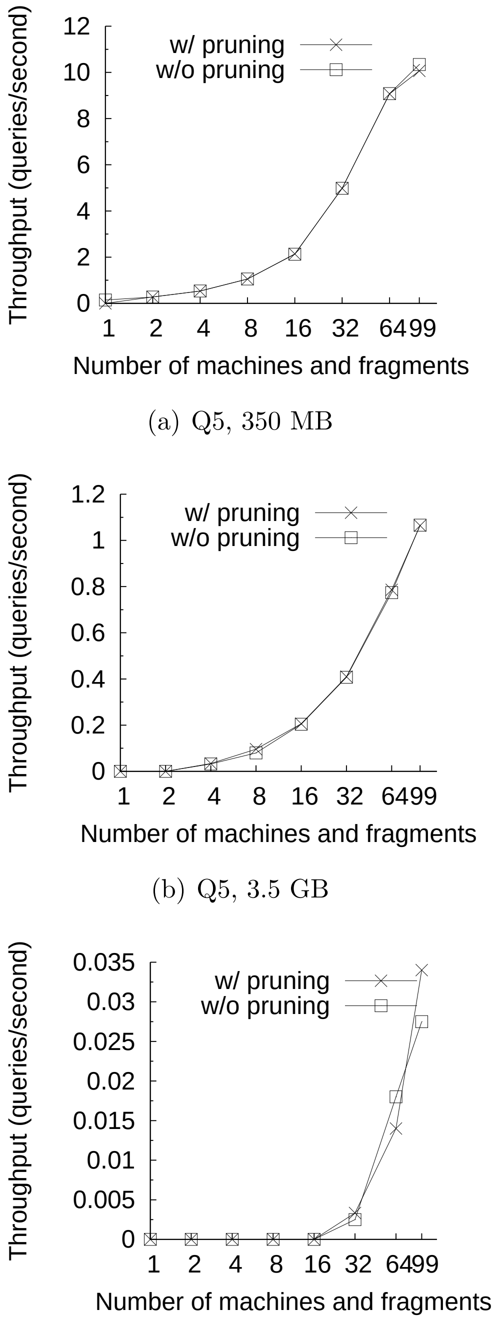
(c) Q3, 3.5 GB

(d) Q4, 3.5 GB

(e) Q3, 35 GB

(f) Q4, 35 GB

Figure 9.10: Throughput, balanced horizontal fragmentation (cont'd)



(a) Q5, 350 MB

(b) Q5, 3.5 GB

(c) Q5, 35 GB

Figure 9.11: Throughput, balanced horizontal fragmentation (cont'd)

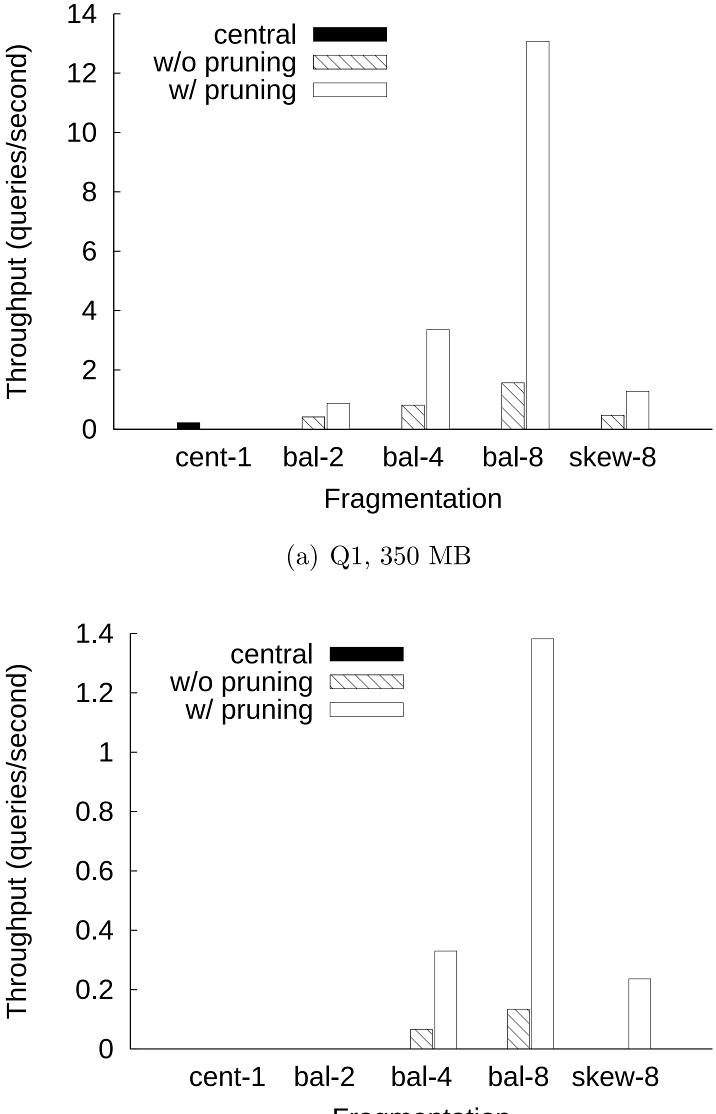
workload: fragmenting on attributes on which single-value selections are performed leads to greater pruning opportunities than fragmenting on attributes that are used in wide range predicates. However, even in the latter case, distributed evaluation by far outperforms centralized querying.

The results also show that once a throughput of approximately 20 queries per second is achieved, further increasing the number of machines does not lead to improved performance. This is because, for simplicity, the experimental setup uses a single query dispatcher instance, which becomes saturated at this point so that distributed query evaluation is no longer the bottleneck. Thus, query performance reaches a plateau and in some cases even decreases slightly (e.g., Q1, 350 MB, as seen in Figure 9.9(a)), which can be explained by thrashing at the dispatcher. In practice, this problem can easily be avoided by dispatching queries from multiple sites.

9.2.2 Skewed Fragmentation

While pruning performs well in the presence of a balanced fragmentation, in practice it is not always possible to achieve this balance. Thus, this section presents an experiment that measures the effect of pruning with a skewed fragmentation consisting of 8 fragments. The skewed fragmentation is defined as follows: the first fragment contains half of the entire collection (corresponding to the first 2 years of the 4-year period), the next fragment contains half of the remaining collection (i.e., 25% of the data), and so forth, with the last fragment containing the remainder of the collection data.

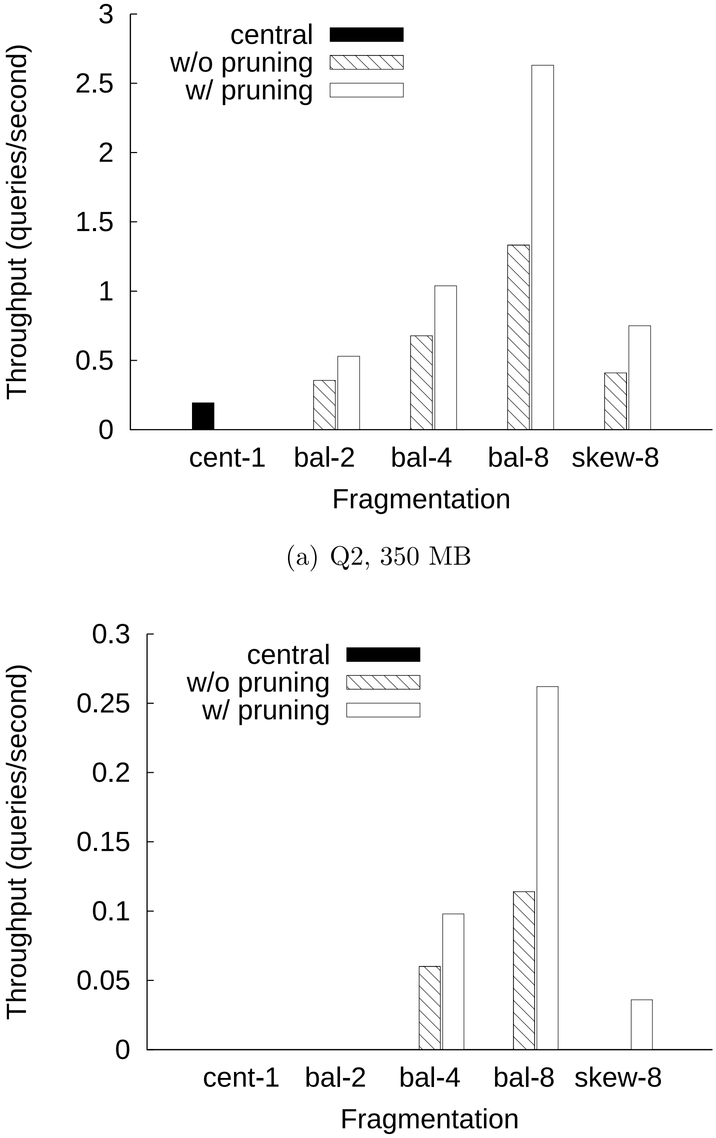
Since the experiments in the previous section have shown that the impact of the horizontal pruning technique on response time is small, the experiments with skewed fragmentation focus on throughput. Figures 9.12 and 9.13 show the throughput rates achieved by centralized query execution (which is vanishingly low in some of the cases shown), as well as distributed query execution (with and without pruning) over a balanced fragmentation consisting of 2, 4 and 8 fragments (denoted “bal-2”, “bal-4”, and “bal-8”, respectively) and over the skewed fragmentation (denoted as “skew-8”). Only queries Q1 and Q2 are used, since these are the queries for which pruning has been shown to be particularly effective. Even in the presence of skew, distribution results in a significant boost in performance



(a) Q1, 350 MB

(b) Q1, 3.5 GB

Figure 9.12: Throughput, balanced and skewed horizontal fragmentation



(a) Q2, 350 MB

(b) Q2, 3.5 GB

Figure 9.13: Throughput, balanced and skewed horizontal fragmentation (cont'd)

over centralized querying in all cases. As with a balanced fragmentation schema, pruning further improves throughput.

The throughput rates obtained with the skewed fragmentation tend to fall between that of a balanced fragmentation with 2 fragments and that of a balanced fragmentation with 4 fragments. This can be explained by the fact that the largest fragment in the skewed fragmentation covers a period of 2 of the 4 years and is therefore the same size as a fragment in the balanced fragmentation with 2 fragments, representing a throughput bottleneck.

To further improve query performance in the presence of a skewed distribution, it may be beneficial to replicate the most heavily loaded fragments. However, this is beyond the scope of this thesis.

9.2.3 Pruning Efficacy

In addition to evaluating the performance impact of pruning, it is interesting to examine how effectively the pruning technique limits query execution to the fragments that actually yield part of the result. To determine this, the fraction of those sites accessed by a pruned query plan that yield part of the query result is measured. The results (based on a balanced fragmentation consisting of 16 fragments) are shown in Figure 9.14. Query Q1 is omitted from this experiment since it can be answered using a single fragment. The cut-off value for the initial bid of the auction is varied from 300 to 800, which affects the selectivity of the queries, with a lower value yielding a larger number of query results from each fragment that is relevant for the query[7]. As can be seen, pruning is more effective for the queries that select a large number of results from each relevant fragment (corresponding to lower bid values). This is because a query that selects a larger portion of the collection is more likely to find a match within a given fragment. The results reported here are derived from the 35 GB collection. With the smaller collections, efficacy tends to be slightly lower, which can be attributed to the lower numbers of results derived from these collections.

Overall, the results of the experiments on horizontal fragmentation show that horizontal

[7] Since bid values are not used in the fragmentation predicates used to define the horizontal fragmentation, altering the bid value has no impact on which fragments can be pruned for a given query.

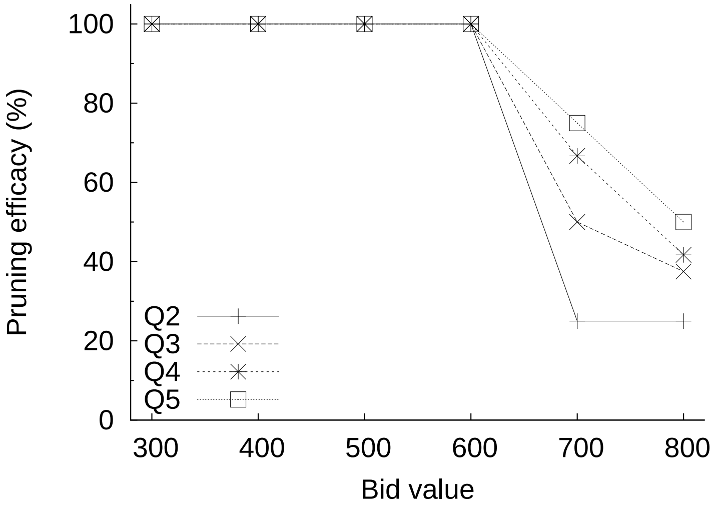


Figure 9.14: Pruning efficacy

fragmentation is highly effective at improving query performance and scalability, in particular when fragmentation skew can be avoided. In addition, pruning is confirmed to be a valuable tool for improving query throughput beyond the level achieved by fragmentation alone, while having no significant impact on response time.

9.3 Techniques for Vertical Fragmentation

The experimental evaluation of the query evaluation techniques for vertically fragmented collections focuses on query response time. In a vertically fragmented system, a single type of query always accesses the same fragments resulting in a closed system in which throughput can only be improved by reducing the response time[8]. This makes a separate study of throughput unnecessary.

[8] In a scenario where multiple different queries are processed at the same time, each query may need to access a different subset of the fragments and thus there might be a potential for optimizing throughput independently of response time. Since multiple-query optimization is not considered in this thesis, this case is not examined here.

To evaluate the performance of the query evaluation techniques for vertically fragmented collections, two sets of experiments are performed:

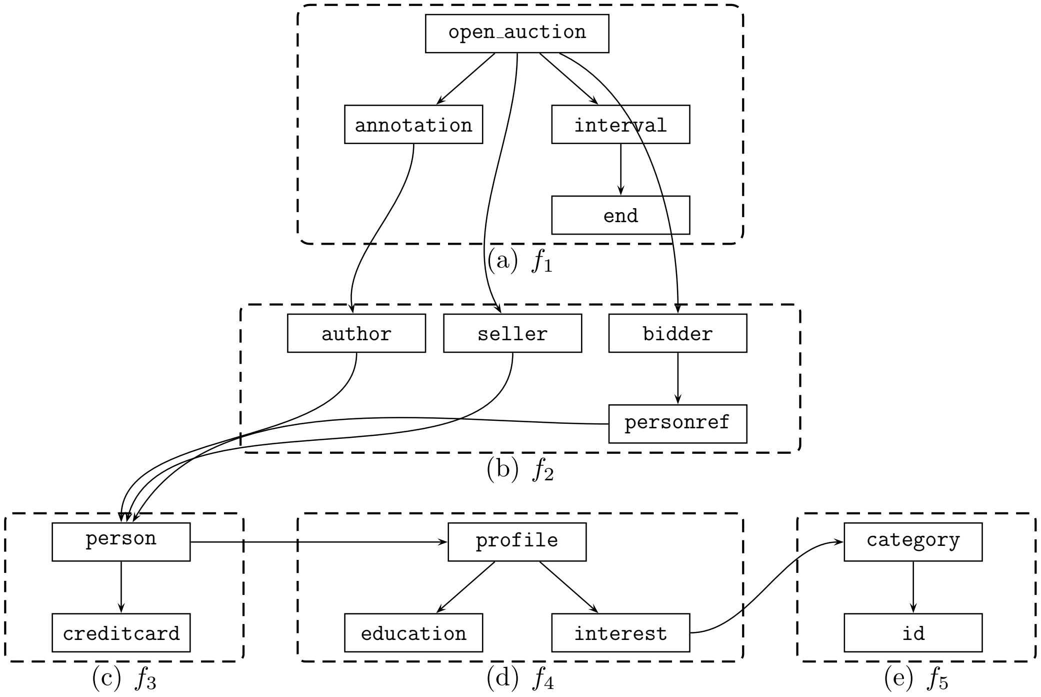
- First, the impact of vertical fragmentation on query performance is evaluated. For this experiment, the performance of centralized query evaluation is compared to that of distributed evaluation using the naïve query evaluation strategy presented in Section 5.2. Additionally, the impact of the various pruning techniques presented in Section 6.2.1 is considered.
- Next, the impact of cross-fragment join pushing (as described in Section 6.2.2.1) and node type path filtering (as described in Section 6.2.2.4) is examined in detail.

9.3.1 Fragmentation and Pruning

To evaluate the performance impact of vertical fragmentation and the pruning techniques that can be applied in this scenario, the same decomposed XMark collection used in the horizontal experiments is employed. This collection is scaled to 350 MB and 3.5 GB[9] and then partitioned based on the vertical fragmentation schema shown in simplified form in Figure 9.15. This fragmentation schema was chosen because it provides the opportunity to examine cases where the number of fragments accessed by each query (before and after pruning) varies widely. Fragmenting the collection in this fashion results in a skewed fragmentation because different node types in the collection occur with different frequencies.

Over this collection, queries that have been chosen based on their characteristics (shown in Table 9.4) are evaluated. Q6 involves only a single fragment (fragment f_1 in Figure 9.15). Previous work has shown that this is the ideal case for vertical fragmentation [12]. The remaining queries, however, reach all five of the fragments shown in Figure 9.15. While Q7 to Q10 reach the same number of fragments, the number of structural and value constraints they contain increases from Q7 to Q10.

[9] As shown in Section 9.2.1, centralized query evaluation over the 35 GB collection and distributed query evaluation over a 35 GB collection fragmented into fewer than eight fragments did not finish within two hours for any of the queries. Thus, this collection size is omitted here.



(a) f_1

(b) f_2

(c) f_3

(d) f_4

(e) f_5

Figure 9.15: Fragmentation schema used to evaluate vertical fragmentation and pruning

Q6	`/open_auction[initial > 200]/interval/end`
Q7	`/open_auction//person//category[id='category10']`
Q8	`/open_auction/bidder//person//category[id='category10']`
Q9	`/open_auction/bidder//person[creditcard]//category[id='category10']`

Q10	`/open_auction/bidder//person[creditcard]/profile[education]`
	`//category[id='category10']`

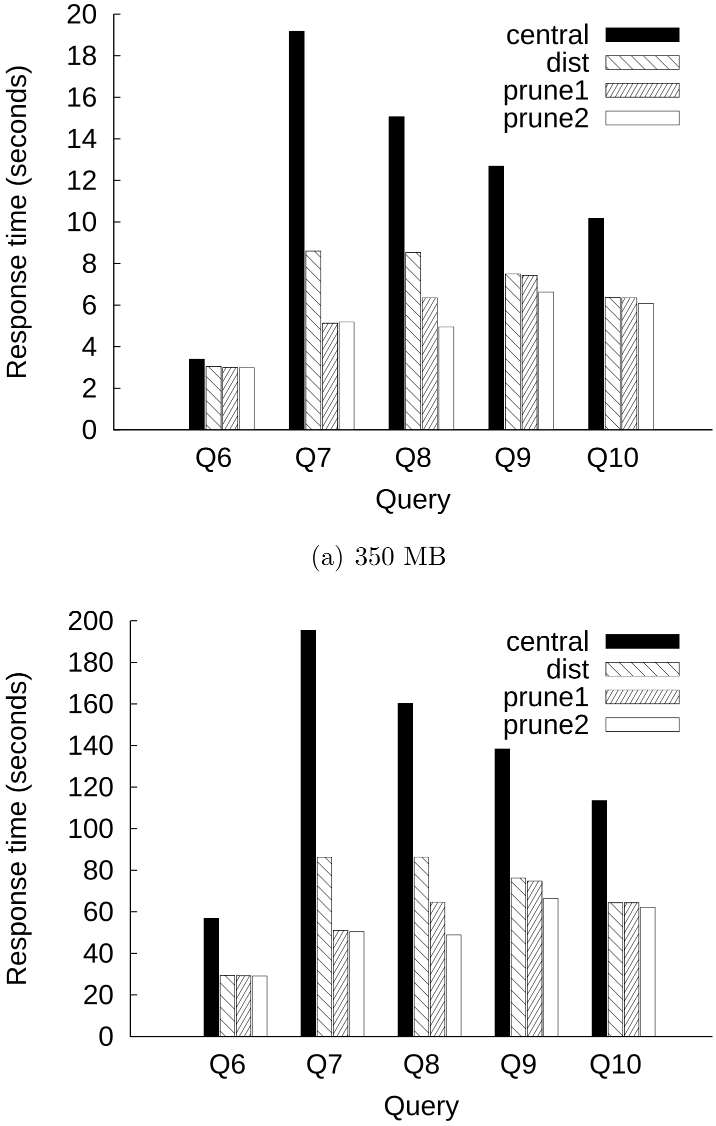
Table 9.4: Queries used to evaluate vertical fragmentation and pruning

Figure 9.16 shows, for each collection and query, the response time obtained by centralized query execution (denoted as "central"), naïve distributed execution without any pruning (denoted as "dist"), distributed execution with pruning of fragments on which no structural constraints are placed (denoted as "prune1") and distributed execution with additional pruning based on node type paths (denoted as "prune2").

As can be seen, distributed execution outperforms centralized execution by a significant margin in all cases. In most cases, both pruning techniques further improve performance but their effectiveness varies depending on the query. To analyze the impact of the pruning techniques, it is useful to consider the number of fragments that each technique accesses for each query, which is shown in Table 9.5. For Q6, which can be answered by accessing a single fragment, all distributed execution techniques yield approximately the same response time. For Q7, naïve distributed execution needs to access five fragments, whereas

Query	Fragments accessed		
	Dist	Prune 1	Prune 2
Q6	1	1	1
Q7	5	1	1
Q8	5	2	1
Q9	5	3	2
Q10	5	4	3

Table 9.5: Number of fragments accessed, vertical fragmentation



(a) 350 MB

(b) 3.5 GB

Figure 9.16: Response time, vertical fragmentation

both pruning techniques access only a single fragment. This explains why both pruning techniques yield comparable response times, which are approximately half of that of naïve distributed execution. In the case of Q8, pruning of fragments without structural constraints performs better than naïve distributed execution. Additional pruning based on node type paths in turn performs better than pruning only the fragments without structural constraints. Again, these results are reflected in the number of fragments accessed by each of these techniques. For Q9 and Q10, even with pruning, multiple fragments need to be accessed. Thus, response times for all distributed techniques are approximately on par with each other.

Together, these results show the impact of pruning on query performance in a vertically fragmented scenario. As can be seen, query performance is related both to the opportunity for pruning provided by a given query and fragmentation schema, as well as to the effectiveness of the pruning technique used. This corroborates the usefulness of pruning as a technique for improving the performance of query evaluation over vertically fragmented collections.

Q11	/open_auction[initial > 200]//item//mail/from
Q12	/open_auction[initial > 200][.//author/person /name[starts-with(., ’Ry’)]]//item//mail/from
Q13	/open_auction[initial > 200][.//author/person/ name[starts-with(., ’Ry’)]]//item//category/id
Q14	/open_auction[initial > 200][.//author/person[profile/age > 30] /name[starts-with(., ’Ry’)]]//item//category/id
Q15	/open_auction[initial > 200]//author/person[starts-with(name, ’Ry’)] /profile/interest/category/description

Table 9.6: Queries used to evaluate cross-fragment join pushing and node type path filtering

9.3.2 Cross-Fragment Join Pushing and Node Type Path Filtering

Next, the performance impact of cross-fragment join pushing and node type path filtering is examined. To do this, two sets of experiments are performed. First, a set of carefully chosen queries is evaluated over the decomposed XMark collection consisting of many small documents. These queries were chosen to represent different scenarios encountered by distributed query evaluation with join pushing and node type path filtering. In a second set of experiments, the impact of these query evaluation techniques is evaluated when applied to the XPathMark queries in the context of an unmodified XMark collection.

9.3.2.1 Effects in Various Scenarios

The first experiment is based on a set of queries that is designed to test several different cases that affect how cross-fragment join pushing and node type path filtering can be applied (Q11-Q15 in Table 9.6). After scaling the decomposed XMark collection used in the previous experiment to 350 MB and 3.5 GB, the collection is partitioned vertically according to the vertical fragmentation schema shown in Figure 9.17. This fragmentation schema was chosen because it yields fragments that differ widely in size, leading to an interesting variety of opportunities for pushing cross-fragment joins.

The results of this experiment are shown in Figure 9.18. For each query and collection size, the response time achieved by naïve distributed query evaluation (denoted as "dist") is compared to that of distributed query evaluation with node type path filtering (denoted as "filter") and of distributed query evaluation with cross-fragment join pushing (denoted as "push"). As before, all measurements include the cost of constructing sub-query results, shipping them between sites, and shipping the overall query result to the dispatcher.

As can be seen, cross-fragment join pushing improves the performance of distributed query evaluation by a significant margin for all queries and collection sizes. The performance benefit is particularly large for queries Q13, Q14, and Q15. This can be explained by the fact that these queries access fragment f_8, which is the largest fragment in the collection. Since these queries contain highly selective value constraints on the content of

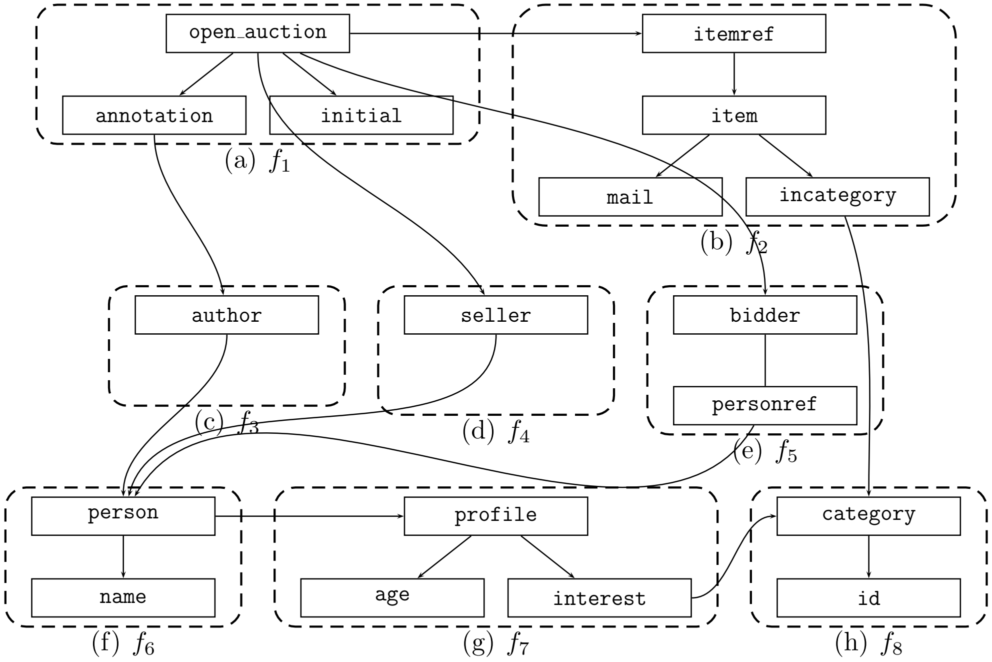
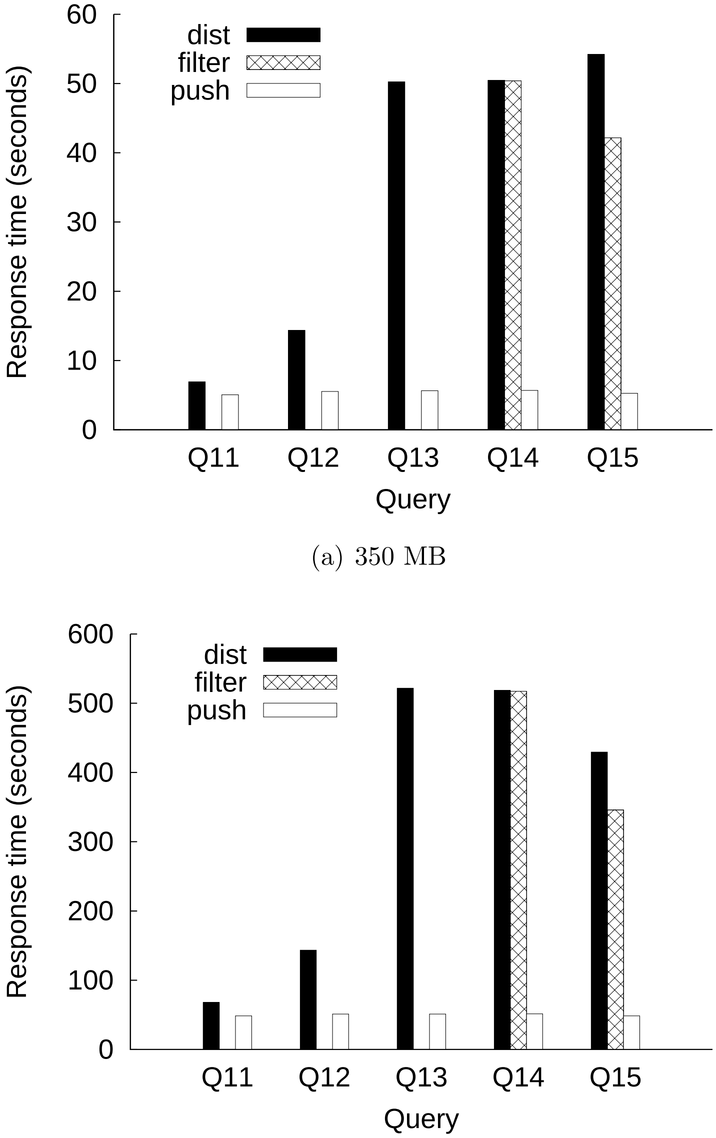


Figure 9.17: Fragmentation schema used to evaluate join cross-fragment pushing and node type path filtering



(a) 350 MB

(b) 3.5 GB

Figure 9.18: Impact of cross-fragment join pushing and node type path filtering

node of types `initial` (in fragment f_1) and `name` (in fragment f_6), cross-fragment join pushing successfully limits query execution to a relatively small fraction of the sub-trees in fragment f_8, thus improving query performance.

Node type path filtering, in contrast, can only be applied to queries Q14 and Q15 (with the other queries, there is no opportunity for filtering based on node type paths). For Q14, where one fragment (f_7) contains sub-trees that can be filtered, this technique does not lead to a significant improvement in query performance when compared to naïve distributed query evaluation. For Q15, however, where node type path filtering can be applied to two fragments (f_7 and f_8), a significant performance improvement can be observed. However, the performance of query execution with node type path filtering is still inferior to that of cross-fragment join pushing. This supports the intuition that join pushing should be preferred unless other considerations make it impossible to use this technique (e.g., if a non-left-deep plan is required).

9.3.2.2 Effects with XPathMark Queries

For the second experiment, queries from the XPathMark benchmark are used (A1-A6 and B7, shown in Table 9.7). Since these queries are primarily designed to evaluate the performance of evaluating XPath axes, they contain few filtering predicates and each return a large portion of the nodes in the collection as their result. While this is an important use case, it is also important to capture the equally realistic scenario of queries that do have such filtering predicates. Therefore, a value predicate was added to each query, resulting in the selective XPathMark queries A1S-A6S and B7S. Both the original and the selective XPathMark queries are evaluated over an unmodified XMark collection, scaled to 120 MB, 1.2 GB and 12 GB. This collection is fragmented into 10 vertical fragments according to the fragmentation schema shown in Figure 9.19. This fragmentation schema was chosen because it provides opportunity for cross-fragment join pushing. Note that no pruning was performed during this experiment.

Figures 9.20, 9.21, and 9.22 show the results of this experiment. As can be seen, for most of the unmodified XPathMark queries, cross-fragment join pushing leads to a significant improvement in performance. The sole exception to this is query A6, where

A1	/site/closed_auctions/closed_auction/annotation/description/text/keyword
A2	//closed_auction//keyword
A3	/site/closed_auctions/closed_auction//keyword
A4	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
A5	/site/closed_auctions/closed_auction[descendant::keyword]/date
A6	/site/people/person[profile/gender and profile/age]/name
B7	//person[profile/@income]/name
A1S	/site/closed_auctions/closed_auction[price > 600]/annotation/description/text/keyword
A2S	//closed_auction[price > 600]//keyword
A3S	/site/closed_auctions/closed_auction[price > 600]//keyword
A4S	/site/closed_auctions/closed_auction[price > 600][annotation/description/text/keyword]/date
A5S	/site/closed_auctions/closed_auction[price > 600][descendant::keyword]/date
A6S	/site/people/person[starts-with(name, ’Ry’)][profile/gender and profile/age]/name
B7S	//person[starts-with(name, ’Ry’)][profile/@income]/name

Table 9.7: XPathMark queries and selective XPathMark queries

join pushing leads to a slight decrease in performance. This can be explained by the fact that for this query, even with cross-fragment join pushing, all sub-trees in fragment f_9 need to be accessed. In practice, queries for which cross-fragment join pushing decreases performance do not present a problem since cost-based optimization will not yield a plan that uses this technique in these cases.

For the more selective queries, the benefit of cross-fragment join pushing is more pronounced. As can be seen, this query evaluation technique is beneficial for all of the selective queries. This result can be explained by the fact that cross-fragment join pushing exploits selective constraints posed over the content of one fragment to reduce the number of sub-trees that need to be accessed in another fragment. Since the selective queries contain

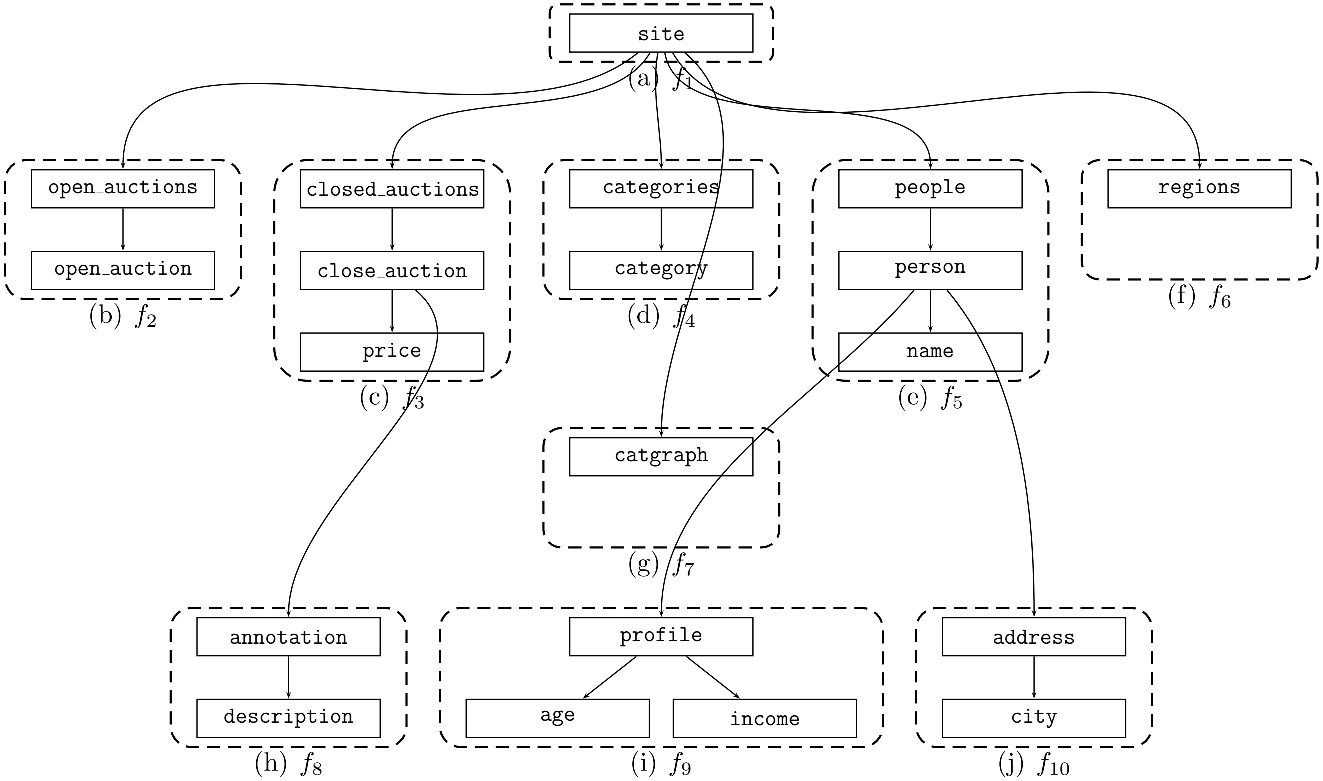
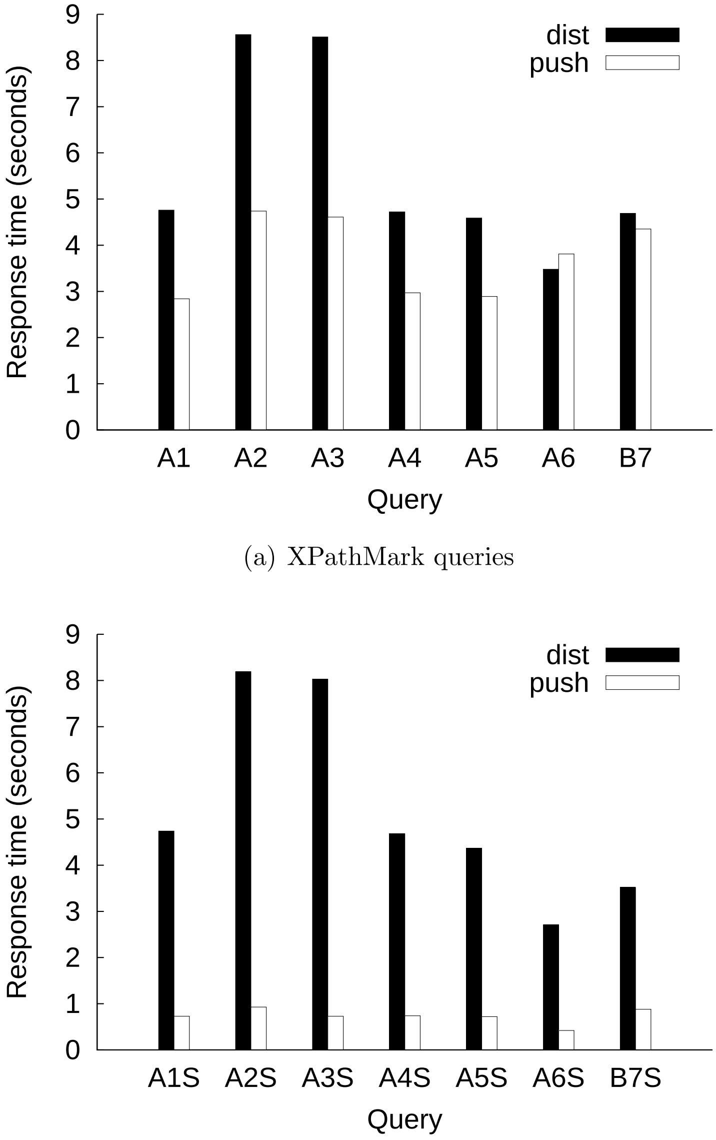


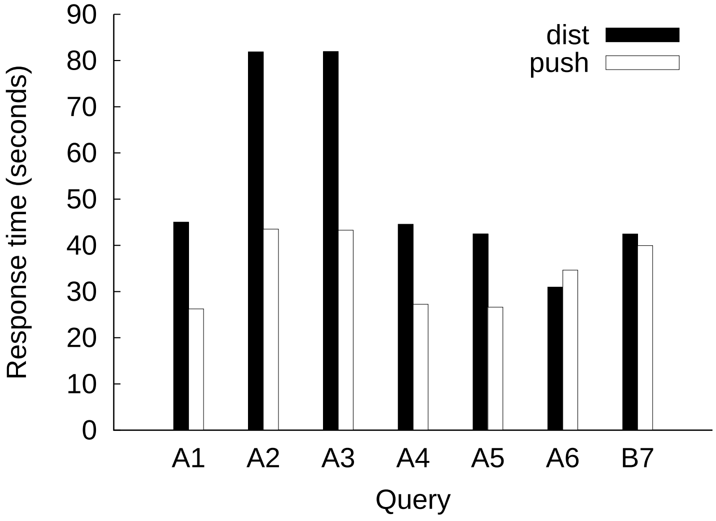
Figure 9.19: Fragmentation schema used to evaluate cross-fragment join pushing with XPathMark queries



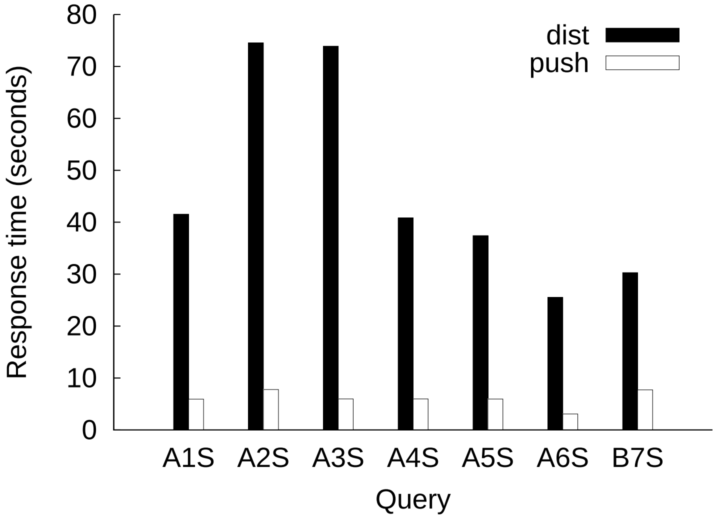
(a) XPathMark queries

(b) Selective XPathMark queries

Figure 9.20: Impact of cross-fragment join pushing, 120 MB

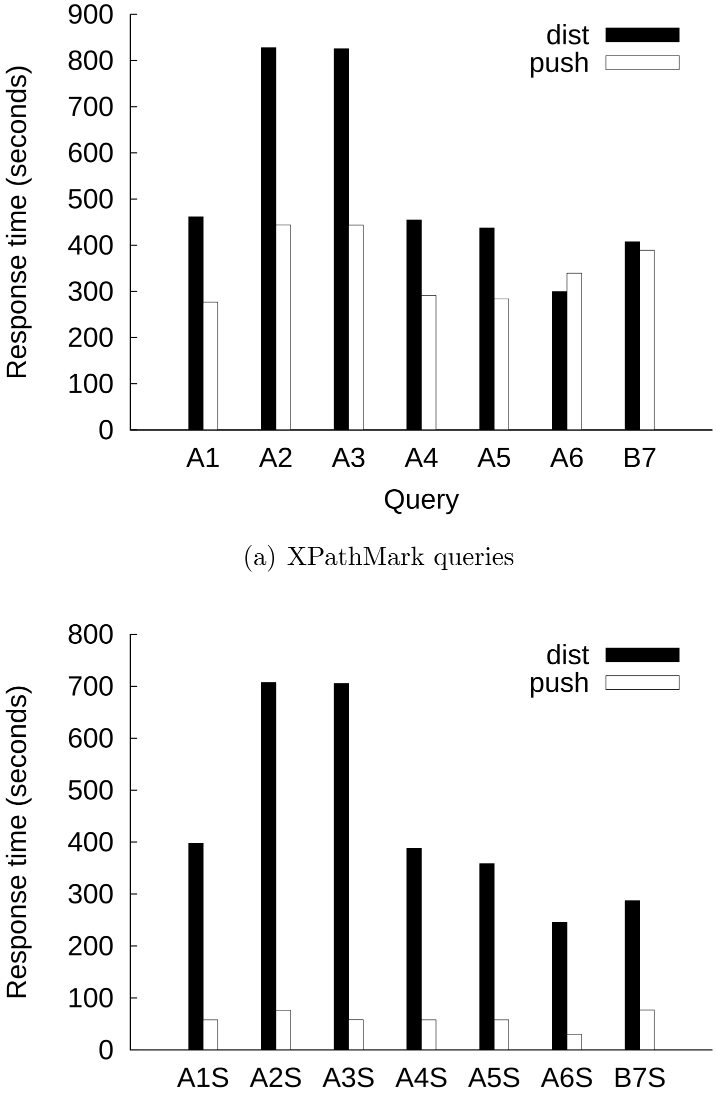


(a) XPathMark queries



(b) Selective XPathMark queries

Figure 9.21: Impact of cross-fragment join pushing, 1.2 GB



(a) XPathMark queries

(b) Selective XPathMark queries

Figure 9.22: Impact of cross-fragment join pushing, 12 GB

more of these constraints, the benefit of cross-fragment join pushing is greater for these queries.

9.4 Cost Model

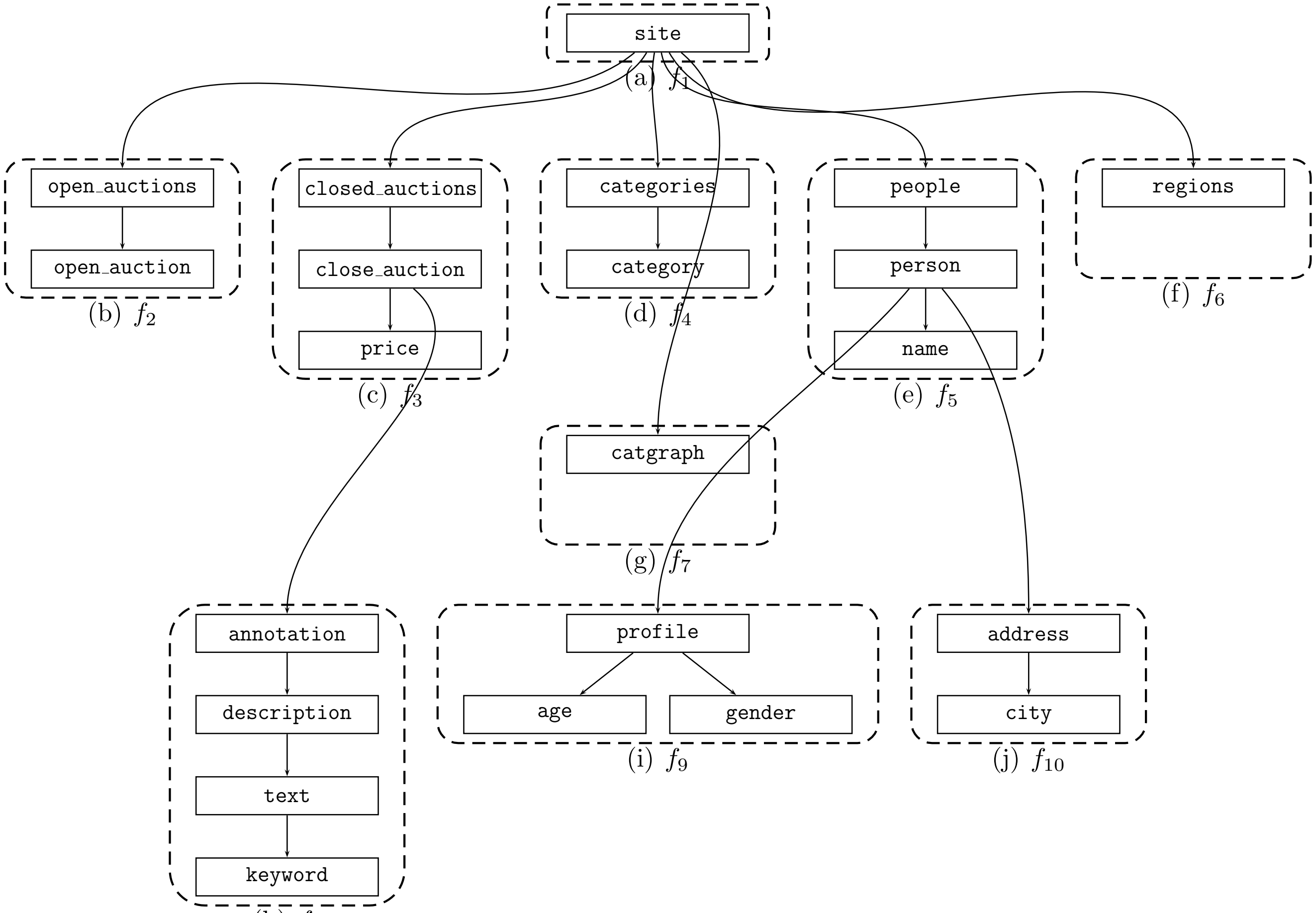
The cost-based query optimization technique presented in Chapter 7 works by enumerating the candidate plans for a given query and distributed collection, estimating the cost of each such candidate plan and then choosing the plan with the lowest estimated cost. This section presents a thorough evaluation of this technique. The methods used in this evaluation are partially inspired by Mackert and Lohman's work on validating the R* optimizer [97, 98].

The goal of this evaluation is twofold: first it is shown that for each candidate plan, estimated cost and actual cost are closely correlated. In particular, it is important that the relative order of candidate plans for the same query is preserved, i.e., if candidate plan G_P^1 has a significantly lower estimated cost than candidate plan G_P^2 then the actual cost (in terms of response time) of G_P^1 should also be significantly lower than the actual cost of G_P^2. The second goal of this evaluation is to show that cost-based optimization successfully determines a near-optimal plan for each query (i.e., a plan whose actual response time cost is close to that of the plan with the lowest actual response time cost).

To estimate the cost of a DEP, the distributed cost estimation techniques presented in this thesis rely on cost estimates for the LQPs contained in this DEP and compose these estimates to a cost estimate for the entire DEP. Since the goal of this experiment is to evaluate the accuracy of distributed cost estimation, rather than that of (centralized) cost estimation for individual LQPs, actual, measured LQP costs are used as inputs to distributed cost estimation.

For the experiments presented in this section, five queries were chosen that exemplify the different scenarios encountered during cost estimation. These queries (C1, C2, C3, C4, and C5) are shown in Table 9.8. The queries are then evaluated over an XMark collection of size 1.2 GB that has been fragmented vertically according to the fragmentation schema shown in Figure 9.23[10]. This fragmentation schema has been chosen such that each of the

[10] Since horizontal fragmentation yields fewer alternative distributed execution plans (differing only in



(a) f_1 (b) f_2 (c) f_3 (d) f_4 (e) f_5 (f) f_6 (g) f_7 (h) f_8 (i) f_9 (j) f_{10}

Figure 9.23: Fragmentation schema used to validate cost model

queries requires access to multiple fragments during query evaluation even after pruning. This results in a variety of possible execution plans for each query and makes it possible to compare estimated cost and actual cost of each alternative.

Queries C1 and C2 share the same structure. When evaluating these queries over the fragmented collection, fragments f_1, f_3, and f_8 need to be accessed. However, C1 and C2 differ in the constant used in the value constraint evaluated over nodes of type `price`. This influences the cardinality of the LQP evaluated over fragment f_3 and thereby the effectiveness of pushing the cross-fragment join between this LQP and the LQP evaluated over fragment f_8 into the latter LQP.

Evaluating query C3 requires access to fragments f_1, f_5, and f_9. Query C4 is evaluated over the same fragments. However, it contains a negation. When evaluating query C4, the negation is pushed into the cross-fragment join between the LQP evaluated over fragment f_5 and the LQP evaluated over fragment f_9. This makes it possible to evaluate the accuracy of cost estimation for cross-fragment joins into which a negation has been folded.

Query C5, which is also evaluated over fragments f_1, f_5, and f_9, contains no selective value constraints in the LQPs evaluated over fragments f_1 and f_5. Thus, for this query, join pushing should not yield a significant benefit, making it possible to verify that this scenario is handled correctly by cost estimation.

To validate the cost estimation techniques presented in Chapter 7, the candidate plans considered by cost-based query optimization are enumerated for each of the five queries[11].

C1	`/site/closed_auctions/closed_auction[price > 600]/annotation/description/text/keyword`
C2	`/site/closed_auctions/closed_auction[price > 100]/annotation/description/text/keyword`
C3	`/site/people/person[starts-with(name, 'Ry')][profile/gender and profile/age]/name`
C4	`/site/people/person[starts-with(name, 'Ry') and not(profile/age > 60)]/name`
C5	`//person[profile/@income]/name`

Table 9.8: Queries used to validate cost model

the physical merge operator used), and since all of these plan alternatives can be expected to have similar response time costs (differing primarily in whether a separate sorting step is necessary), the experiments presented in this section focus on cost-based optimization for vertically fragmented collections.

[11] Note that since an implementation of the symmetric hash join operator is not available in Natix, plans

For each candidate plan, estimated cost and actual cost are then compared.

Figures 9.24, 9.25, 9.26, 9.27, and 9.28 show the results of the experiments for queries C1, C2, C3, C4, and C5, respectively. For each query, two diagrams are shown:

- a scatter plot, in which each data point corresponds to a candidate plan for the query, with the estimated cost shown on the x-axis and the actual cost shown on the y-axis, and
- a diagram in which candidate plans are ordered by their estimated cost. For each candidate plan, estimated and actual cost are shown side by side.

Additionally, the candidate plans considered for queries C1, C2, C3, C4, and C5 are shown in Tables 9.9, 9.10, 9.11, 9.12, and 9.13, respectively. For queries C1 and C2, the LQPs p_k^1, p_k^2, and p_k^3 (where k denotes the query) correspond to fragments f_1, f_3 and f_8, respectively. For queries C3, C4, and C5, the LQPs p_k^1, p_k^2, and p_k^3 correspond to fragments f_1, f_5, and f_9, respectively.

As can be seen, in the case of query C3, the performance of the best plan in the search space is more than eight times better than the performance of the worst plan. This confirms the motivation for cost-based optimization and shows that choosing an appropriate plan has a large impact on query performance.

To evaluate the relationship between estimated cost and actual cost, one can compare the data points in the scatter plot with the dashed line, which represents a perfect correspondence between estimated cost and actual cost. Analyzing these results for queries C1 through C5 shows that, while there is some error, cost estimates for all candidate plans are close to the actual cost. Thus, for all candidate plans for each of the five queries, estimated cost and actual cost are well correlated.

Analytically, this is confirmed by the Pearson correlation coefficients [117] shown in Table 9.14. These coefficients are a standard measure of correlation. A Pearson coefficient of 1 indicates perfect linear correlation, a coefficient of 0 indicates no linear correlation, and a coefficient of -1 indicates perfect inverse correlation. For all queries for which Pearson

containing this operator are not considered in this experiment.

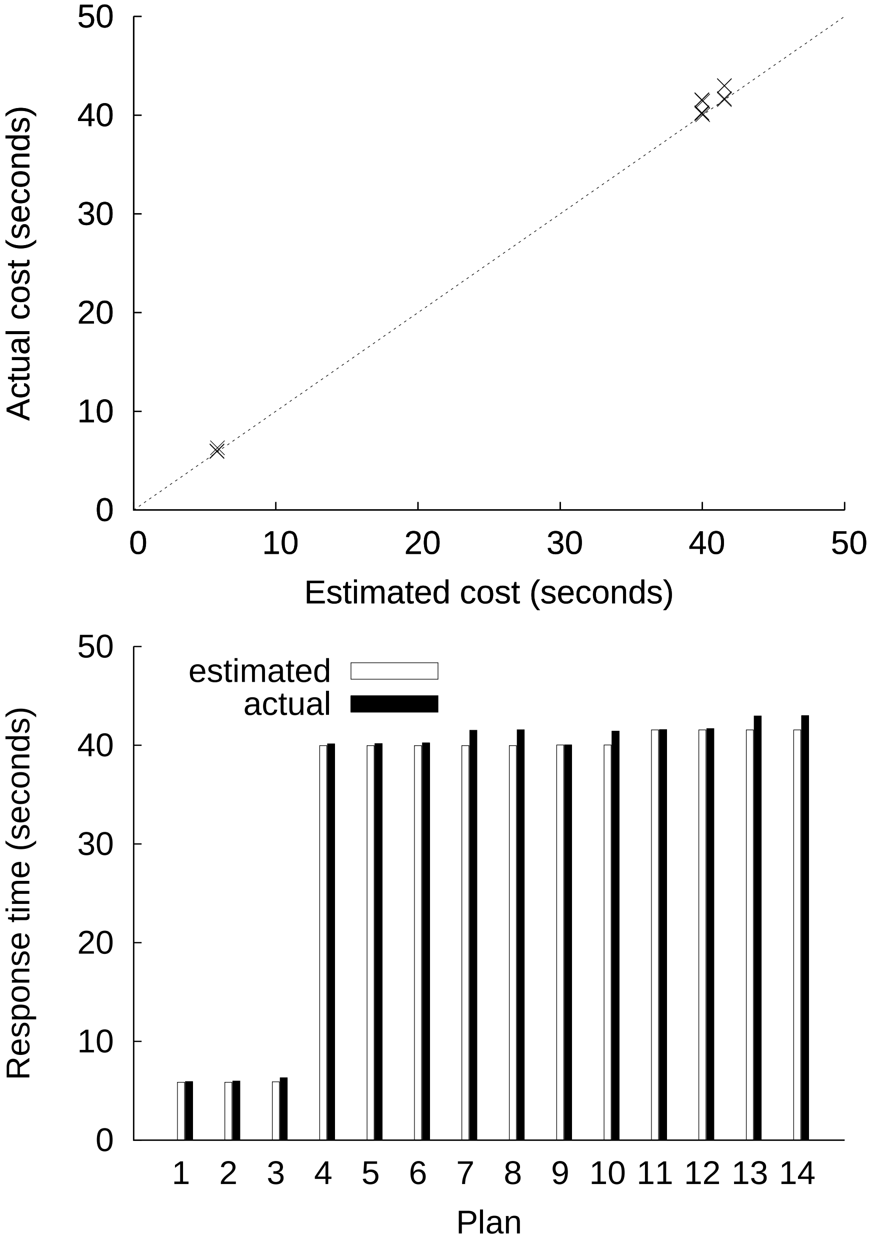


Figure 9.24: Query C1, estimated cost vs. actual cost

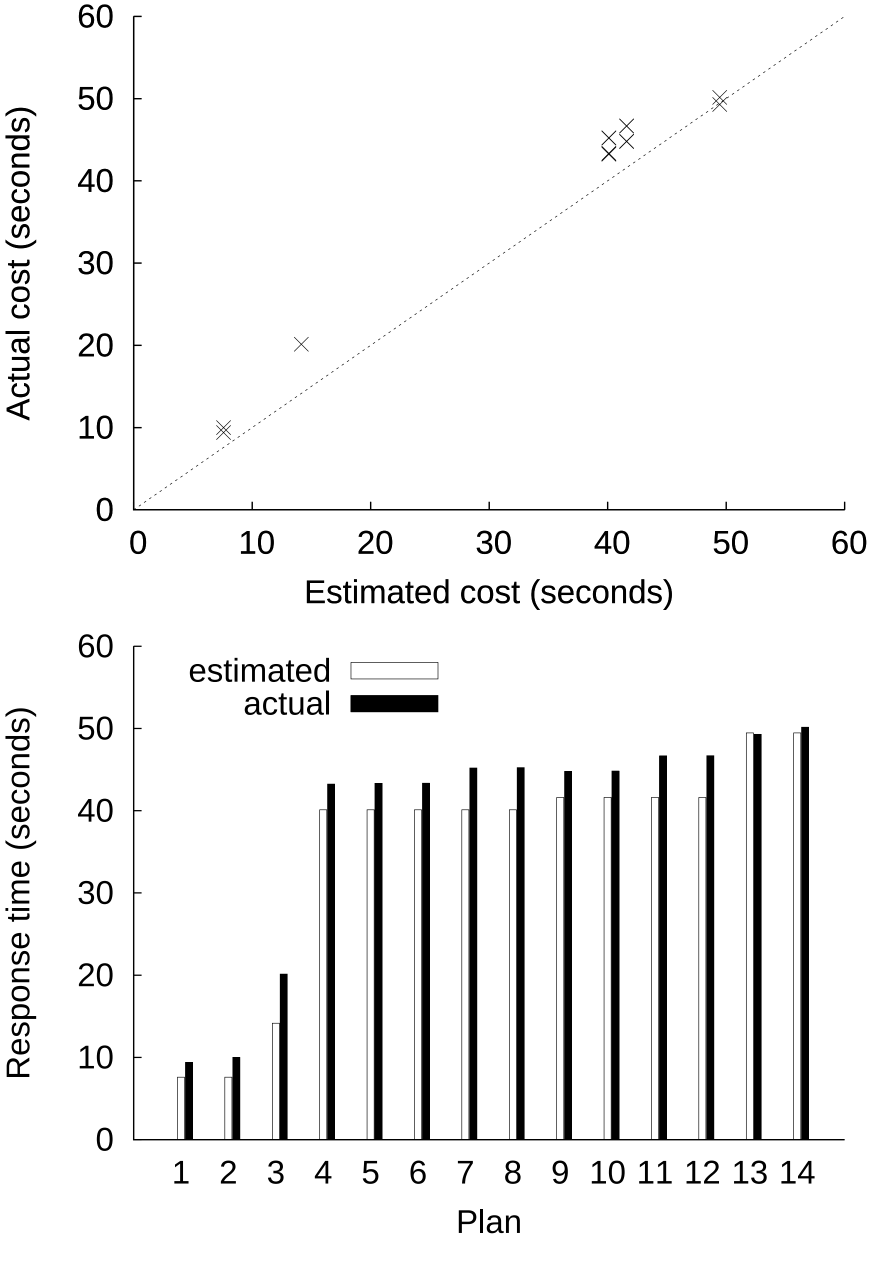


Figure 9.25: Query C2, estimated cost vs. actual cost

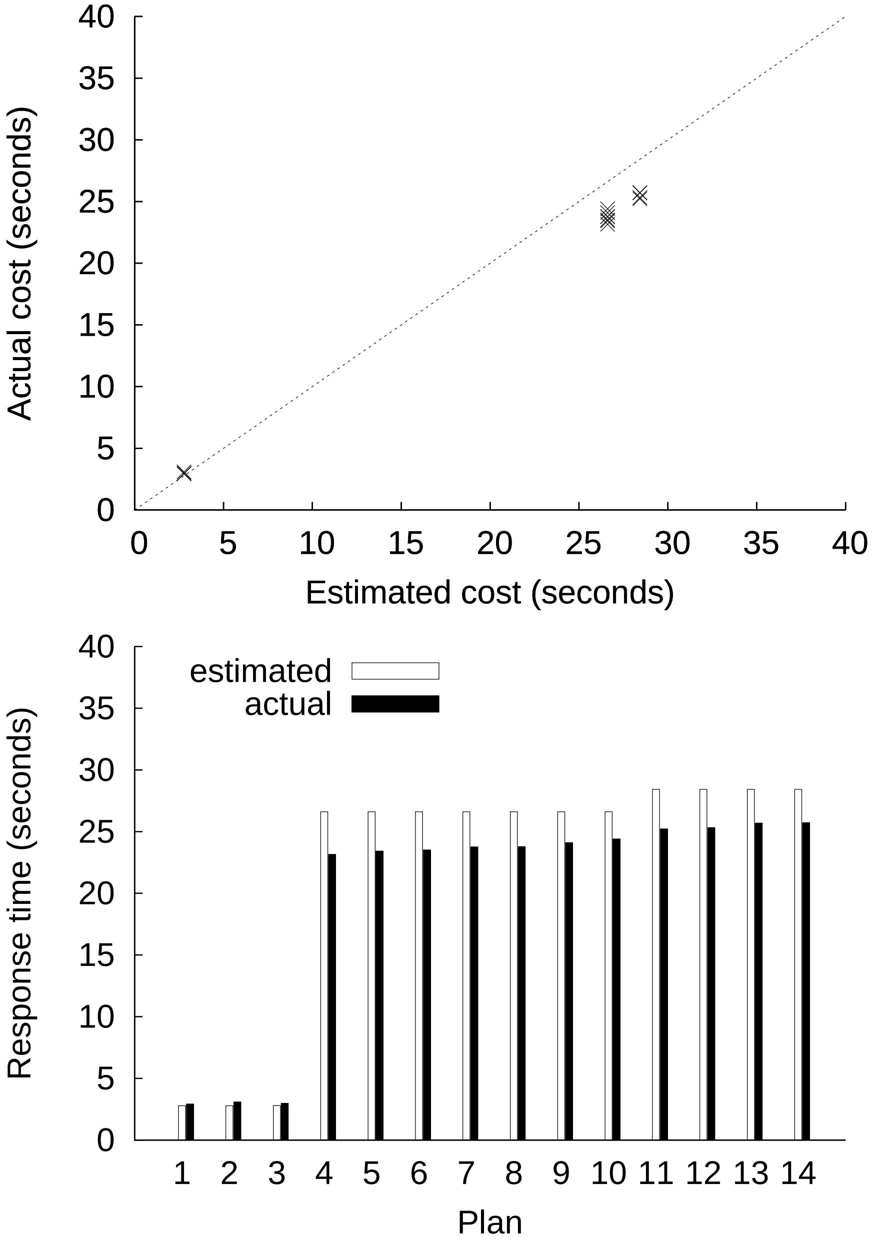


Figure 9.26: Query C3, estimated cost vs. actual cost

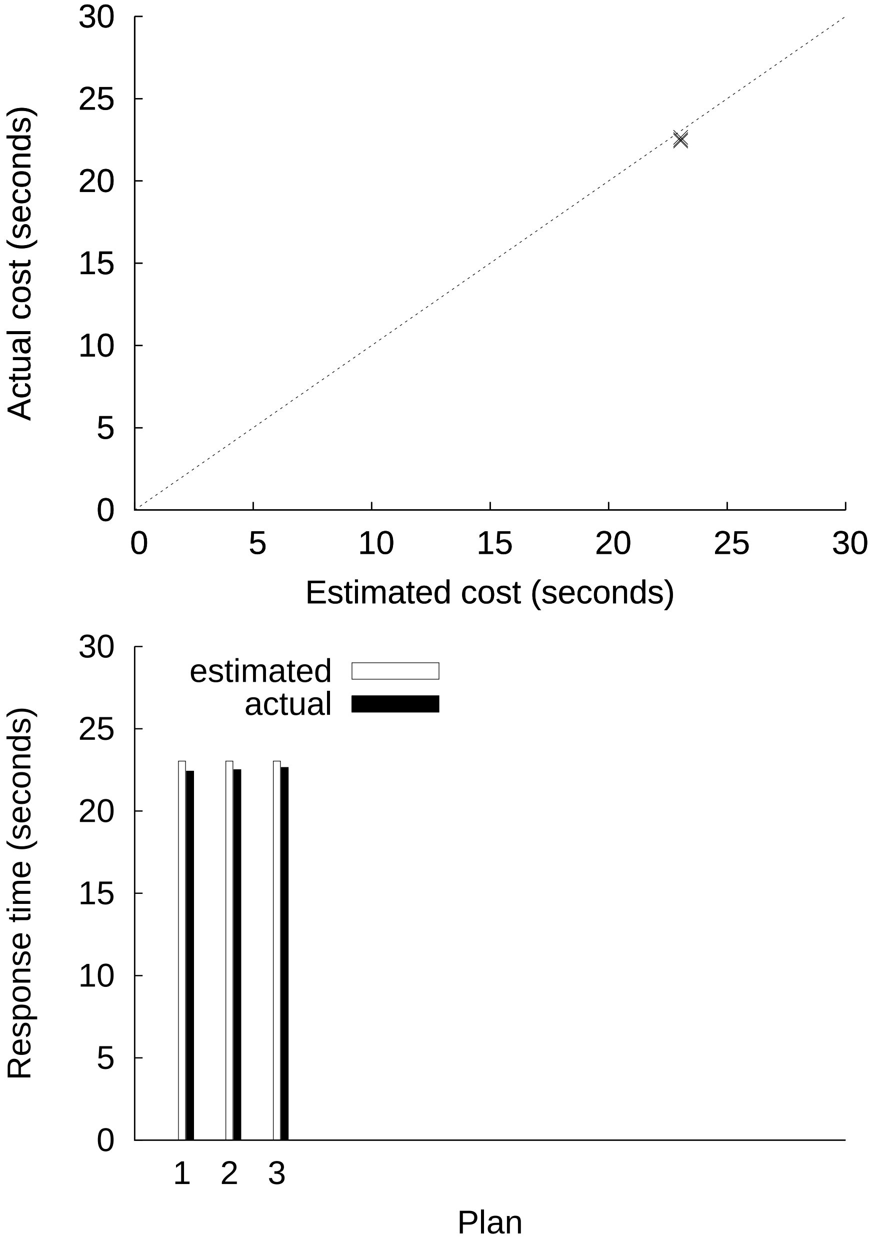


Figure 9.27: Query C4, estimated cost vs. actual cost

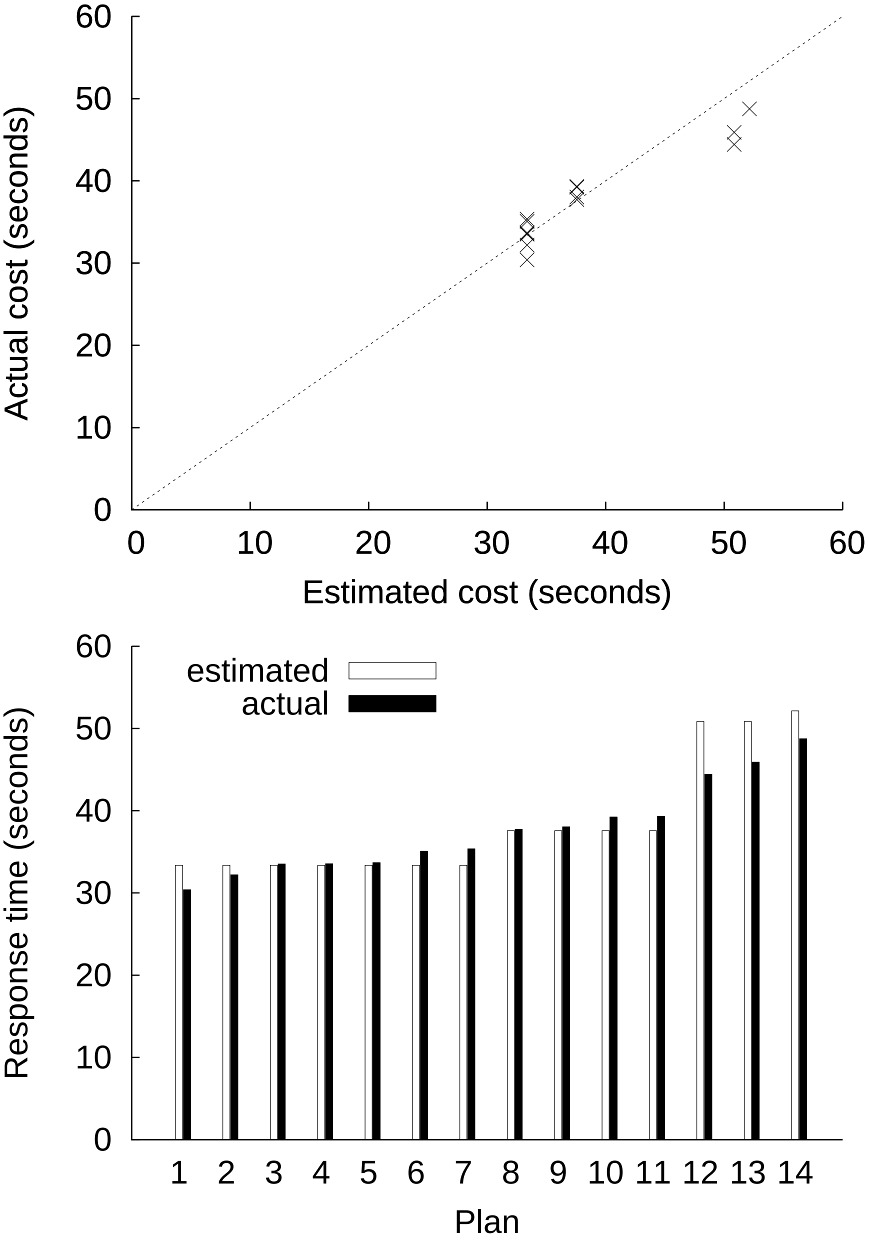


Figure 9.28: Query C5, estimated cost vs. actual cost

No.	Plan	Est. cost	Act. cost
1	$\bar{p}_1^3(\text{scan}(RP) \bowtie^{\text{I}} \bar{p}_1^2(\text{scan}(RP) \bowtie^{\text{I}} p_1^1))$	5.86	5.93
2	$\bar{p}_1^3(\text{scan}(RP) \bowtie^{\text{I}} (p_1^1 \bowtie^{\text{M}} p_1^2))$	5.86	5.98
3	$\bar{p}_1^3(\text{scan}(RP) \bowtie^{\text{I}} (\mathbb{S}(p_1^1) \bowtie^{\text{M}} \mathbb{S}(p_1^2)))$	5.90	6.30
4	$\bar{p}_1^2(\text{scan}(RP) \bowtie^{\text{I}} p_1^1) \bowtie^{\text{M}} p_1^3$	39.96	40.14
5	$(p_1^1 \bowtie^{\text{M}} p_1^2) \bowtie^{\text{M}} p_1^3$	39.96	40.17
6	$(\mathbb{S}(p_1^1) \bowtie^{\text{M}} p_1^2) \bowtie^{\text{M}} p_1^3$	39.96	40.25
7	$(\mathbb{S}(p_1^1) \bowtie^{\text{M}} \mathbb{S}(p_1^2)) \bowtie^{\text{M}} p_1^3$	39.96	41.51
8	$(p_1^1 \bowtie^{\text{M}} \mathbb{S}(p_1^2)) \bowtie^{\text{M}} p_1^3$	39.96	41.56
9	$(p_1^1 \bowtie^{\text{M}} p_1^2) \bowtie^{\text{H}} p_1^3)$	40.02	40.04
10	$(p_1^1 \bowtie^{\text{M}} \mathbb{S}(p_1^2)) \bowtie^{\text{H}} p_1^3)$	40.02	41.42
11	$(p_1^1 \bowtie^{\text{M}} p_1^2) \bowtie^{\text{M}} \mathbb{S}(p_1^3)$	41.55	41.58
12	$(\mathbb{S}(p_1^1) \bowtie^{\text{M}} p_1^2) \bowtie^{\text{M}} \mathbb{S}(p_1^3)$	41.55	41.68
13	$(\mathbb{S}(p_1^1) \bowtie^{\text{M}} \mathbb{S}(p_1^2)) \bowtie^{\text{M}} \mathbb{S}(p_1^3)$	41.55	42.96
14	$(p_1^1 \bowtie^{\text{M}} \mathbb{S}(p_1^2)) \bowtie^{\text{M}} \mathbb{S}(p_1^3)$	41.55	42.99

Table 9.9: Plans considered for query C1

No.	Plan	Est. cost	Act. cost
1	$\bar{p}_2^3(\mathrm{scan}(RP) \bowtie^{\mathrm{I}} \bar{p}_2^2(\mathrm{scan}(RP) \bowtie^{\mathrm{I}} p_2^1))$	7.59	9.40
2	$\bar{p}_2^3(\mathrm{scan}(RP) \bowtie^{\mathrm{I}} (p_2^1 \bowtie^{\mathrm{M}} p_2^2))$	7.59	10.00
3	$\bar{p}_2^3(\mathrm{scan}(RP) \bowtie^{\mathrm{I}} (\mathbb{S}(p_2^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)))$	14.15	20.13
4	$\bar{p}_2^2(\mathrm{scan}(RP) \bowtie^{\mathrm{I}} p_2^1) \bowtie^{\mathrm{M}} p_2^3$	40.11	43.24
5	$(\mathbb{S}(p_2^1) \bowtie^{\mathrm{M}} p_2^2) \bowtie^{\mathrm{M}} p_2^3$	40.11	43.33
6	$(p_2^1 \bowtie^{\mathrm{M}} p_2^2) \bowtie^{\mathrm{M}} p_2^3$	40.11	43.35
7	$(\mathbb{S}(p_2^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)) \bowtie^{\mathrm{M}} p_2^3$	40.11	45.19
8	$(p_2^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)) \bowtie^{\mathrm{M}} p_2^3$	40.11	45.24
9	$(p_2^1 \bowtie^{\mathrm{M}} p_2^2) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^3)$	41.60	44.78
10	$(\mathbb{S}(p_2^1) \bowtie^{\mathrm{M}} p_2^2) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^3)$	41.60	44.82
11	$(p_2^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^3)$	41.60	46.66
12	$(\mathbb{S}(p_2^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)) \bowtie^{\mathrm{M}} \mathbb{S}(p_2^3)$	41.60	46.69
13	$(p_2^1 \bowtie^{\mathrm{M}} p_2^2) \bowtie^{\mathrm{H}} p_2^3)$	49.46	49.29
14	$(p_2^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_2^2)) \bowtie^{\mathrm{H}} p_2^3)$	49.46	50.15

Table 9.10: Plans considered for query C2

No.	Plan	Est. cost	Act. cost
1	$\bar{p}_3^3(\text{scan}(RP) \bowtie^{\text{I}} (p_3^1 \bowtie^{\text{M}} p_3^2))$	2.78	2.93
2	$\bar{p}_3^3(\text{scan}(RP) \bowtie^{\text{I}} \bar{p}_3^2(\text{scan}(RP) \bowtie^{\text{I}} p_3^1))$	2.78	3.09
3	$\bar{p}_3^3(\text{scan}(RP) \bowtie^{\text{I}} (\mathbb{S}(p_3^1) \bowtie^{\text{M}} \mathbb{S}(p_3^2)))$	2.79	2.98
4	$(p_3^1 \bowtie^{\text{M}} p_3^2) \bowtie^{\text{H}} p_3^3)$	26.61	23.16
5	$(p_3^1 \bowtie^{\text{M}} p_3^2) \bowtie^{\text{M}} p_3^3$	26.61	23.43
6	$\bar{p}_3^2(\text{scan}(RP) \bowtie^{\text{I}} p_3^1) \bowtie^{\text{M}} p_3^3$	26.61	23.52
7	$(\mathbb{S}(p_3^1) \bowtie^{\text{M}} p_3^2) \bowtie^{\text{M}} p_3^3$	26.61	23.76
8	$(p_3^1 \bowtie^{\text{M}} \mathbb{S}(p_3^2)) \bowtie^{\text{H}} p_3^3)$	26.61	23.78
9	$(p_3^1 \bowtie^{\text{M}} \mathbb{S}(p_3^2)) \bowtie^{\text{M}} p_3^3$	26.61	24.10
10	$(\mathbb{S}(p_3^1) \bowtie^{\text{M}} \mathbb{S}(p_3^2)) \bowtie^{\text{M}} p_3^3$	26.61	24.41
11	$(\mathbb{S}(p_3^1) \bowtie^{\text{M}} p_3^2) \bowtie^{\text{M}} \mathbb{S}(p_3^3)$	28.43	25.22
12	$(p_3^1 \bowtie^{\text{M}} p_3^2) \bowtie^{\text{M}} \mathbb{S}(p_3^3)$	28.43	25.32
13	$(p_3^1 \bowtie^{\text{M}} \mathbb{S}(p_3^2)) \bowtie^{\text{M}} \mathbb{S}(p_3^3)$	28.43	25.69
14	$(\mathbb{S}(p_3^1) \bowtie^{\text{M}} \mathbb{S}(p_3^2)) \bowtie^{\text{M}} \mathbb{S}(p_3^3)$	28.43	25.73

Table 9.11: Plans considered for query C3

No.	Plan	Est. cost	Act. cost
1	$(p_4^1 \bowtie^{\text{M}} \mathbb{S}(p_4^2))\mathbb{G}\mathbb{A}⟕ p_4^3)$	23.04	22.43
2	$\bar{p}_4^2(\text{scan}(RP) \bowtie^{\text{I}} p_4^1)\mathbb{G}\mathbb{A}⟕ p_4^3)$	23.04	22.51
3	$(p_4^1 \bowtie^{\text{M}} p_4^2)\mathbb{G}\mathbb{A}⟕ p_4^3)$	23.04	22.65

Table 9.12: Plans considered for query C4

No.	Plan	Est. cost	Act. cost
1	$\bar{p}_5^3(\text{scan}(RP) \bowtie^{\mathrm{I}} (p_5^1 \bowtie^{\mathrm{M}} p_5^2))$	33.37	30.40
2	$\bar{p}_5^3(\text{scan}(RP) \bowtie^{\mathrm{I}} \bar{p}_5^2(\text{scan}(RP) \bowtie^{\mathrm{I}} p_5^1))$	33.37	32.19
3	$(p_5^1 \bowtie^{\mathrm{M}} p_5^2) \bowtie^{\mathrm{M}} p_5^3$	33.37	33.52
4	$\bar{p}_5^2(\text{scan}(RP) \bowtie^{\mathrm{I}} p_5^1) \bowtie^{\mathrm{M}} p_5^3$	33.37	33.55
5	$(\mathbb{S}(p_5^1) \bowtie^{\mathrm{M}} p_5^2) \bowtie^{\mathrm{M}} p_5^3$	33.37	33.69
6	$(p_5^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)) \bowtie^{\mathrm{M}} p_5^3$	33.37	35.08
7	$(\mathbb{S}(p_5^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)) \bowtie^{\mathrm{M}} p_5^3$	33.37	35.35
8	$(p_5^1 \bowtie^{\mathrm{M}} p_5^2) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^3)$	37.57	37.72
9	$(\mathbb{S}(p_5^1) \bowtie^{\mathrm{M}} p_5^2) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^3)$	37.57	38.04
10	$(p_5^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^3)$	37.57	39.24
11	$(\mathbb{S}(p_5^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^3)$	37.57	39.33
12	$(p_5^1 \bowtie^{\mathrm{M}} p_5^2) \bowtie^{\mathrm{H}} p_5^3)$	50.85	44.42
13	$(p_5^1 \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)) \bowtie^{\mathrm{H}} p_5^3)$	50.85	45.90
14	$\bar{p}_5^3(\text{scan}(RP) \bowtie^{\mathrm{I}} (\mathbb{S}(p_5^1) \bowtie^{\mathrm{M}} \mathbb{S}(p_5^2)))$	52.14	48.76

Table 9.13: Plans considered for query C5

Query	Correlation
C1	0.99914
C2	0.99210
C3	0.99938
C4	N/A
C5	0.94928

Table 9.14: Pearson correlation coefficient between estimated cost and actual cost of the candidate plans for a given query

coefficients can be computed[12], the coefficients between estimated cost and actual cost are close to 1. This validates the cost model presented in Chapter 7 and shows that it accurately predicts the actual cost of distributed execution plans.

As can be seen in the bar diagrams (Figures 9.24–9.28), the relative order of candidate plans is also largely preserved by cost estimation. The few cases where the order is disturbed occur with candidate plans that are close in both estimated cost and actual cost (as seen with plans 8 and 9 for query C1 in Figure 9.24).

Most importantly, for each of the five queries, one of the DEPs with the lowest estimated cost[13] is also the DEP with the lowest actual cost. Conversely, no DEP with the lowest estimated cost for any of the queries leads to an actual cost that diverges far from the optimum. Together, these results confirm that cost-based optimization using the cost estimation techniques defined in this work is highly successful in choosing distributed execution plans that yield high query performance.

For a more detailed analysis of these results, it is useful to consider each of the five queries individually:

For query C1, the candidate plans can be clustered into three groups based on their

[12] Since all candidate plans for query C4 have the same estimated cost, it is not possible to compute the correlation coefficient for this query. However, as can be seen in Figure 9.27, all candidate plans for this query also have very similar actual costs, corroborating the claim that estimated cost and actual cost are closely correlated.

[13] There are frequently multiple DEPs that share the same cost estimate.

performance (cf. Figure 9.24). The first group consists of plans 1, 2, and 3 as shown in Table 9.9, which have an average estimated cost of 5.87 seconds and an average actual cost of 6.07 seconds. This group consists of exactly those candidate plans that push a cross-fragment join into LQP p_1^3. This pushing technique is particularly effective for this query as all three candidate plans in this cluster have a cost that is much lower than that of the other candidate plans considered. This can be explained by the fact that LQP p_1^3 is the most expensive LQP in each of these candidate plans and reducing the cost of this LQP by restricting its execution to the sub-trees matched by the cross-fragment join significantly improves overall query performance. Cost estimation successfully predicts this effect.

The next cluster consists of plans 4 through 10. These plans do not apply join pushing to the LQP p_1^3, instead either a hash join ($\bowtie^{\mathrm{H}}$) is employed to evaluate the cross-fragment join between p_1^2 and p_1^3 or a merge join ($\bowtie^{\mathrm{M}}$) is used that exploits the order properties present in the result of p_1^3. While the actual costs of the plans in this cluster vary slightly more than their estimated costs, the estimated cost of each plan nevertheless remains a good predictor of the plan's actual cost.

The last group of plans (consisting of plans 11 through 14) uses an explicit sorting step applied to the result of LQP p_1^3. These plans have the worst overall performance, as is predicted by their cost estimates.

Query C2, whose results are shown in Figure 9.25 and whose plans are shown in Table 9.10, is similar to query C1, except that it has a lower price threshold in LQP p_2^2. As a result, both the cardinality and the cost of p_2^2 are larger than that of p_2^1. Thus, the overall cost of the candidate plans for query C2 is no longer as strongly determined by the cost of p_2^3. This has the effect that pushing the cross-fragment join between p_2^2 and p_2^3 into p_2^3 is somewhat less effective for this query and that the candidate plans that employ this technique (plans 1 through 3 in Table 9.10) vary more widely in their performance. Plan 3 has the highest response time among these plans, which can be explained by the relatively large intermediate result of p_2^2 that needs to be sorted before evaluation of p_2^3 can begin.

For the remaining candidate plans, while sorting the result of p_2^3 still has a slight negative impact on performance (seen in plans 9 through 12), the physical join operator used to evaluate the cross-fragment join between p_2^2 and p_2^3 also has a large impact. Using a hash join operator ($\bowtie^{\mathrm{H}}$), as seen in plans 13 and 14 leads to the candidate plans with the worst performance, which can be explained by the larger intermediate result sizes in this scenario. Despite this, cost estimates are still closely correlated with actual cost and, more importantly, ranking the candidate plans for query C2 by their estimated cost yields roughly the same order as ranking them by their actual cost. The sole exception to this are plans 7 and 8, whose actual costs are slightly higher than those of plans 9 and 10 but whose estimated costs are slightly lower. Due to the minor difference in estimated cost between these plans, this cannot be considered a significant alteration to the plan order and therefore does not invalidate the cost estimation technique.

Query C3, whose results are shown in Figure 9.26, follows a different structure and accesses a different subset of the fragments of the collection. The candidate plans that offer the best performance for this query (plans 1, 2, and 3 in Table 9.11) are the plans that push a cross-fragment join into the LQP p_3^3. The remaining plans all yield similar levels of performance, with the plans that sort the result of p_3^3 (i.e., plans 11, 12, 13, and 14) being slightly worse than the rest. As can be seen in the diagrams shown in Figure 9.26, plan order is mostly preserved by cost estimation.

Query C4 contains a negation that in all candidate plans is pushed into the cross-fragment join between LQPs p_4^2 and p_4^3. Since this work considers only one strategy for evaluating cross-fragment joins with pushed negation, cost-based optimization considers only a small number of plans for this query. For each of these plans, the cost is dominated by the cost of LQP p_4^3, which results in all candidate plans having the same estimated cost (as is shown in Figure 9.27 and Table 9.12), making it impossible to compute a correlation coefficient for this query. Nevertheless, since the estimated cost of each candidate plan considered is close to the plan's actual cost, cost estimation can be said to perform well for this query.

As mentioned before, query C5 lacks selective value constraints in LQPs p_5^1 and p_5^2. Thus, the expected result for this query is that pushing the cross-fragment join between p_5^2 and p_5^3 into p_5^3 should yield little benefit. As can be seen in Figure 9.28 and Table 9.13, this is indeed the case and the plans that do push this join (plans 1 and 2) end up with both estimated costs and actual costs that are comparable to those of plans 3 through 7, which do not push this cross-fragment join. Plan 14, which pushes the join between p_5^2

and p_5^3 into p_5^3 but sorts the result of p_5^2 prior to doing this yields the worst performance of all plans considered for this query. This is because the result of p_5^2 is relatively large and sorting it delays the evaluation of p_5^3, leading to a large overhead for pushing this join whereas the benefit (i.e., the number of sub-trees that p_5^3 can skip) is minimal.

Among the plans that do not push a cross-fragment join into p_5^3 (plans 3–13), sorting the result of p_5^3 (plans 8 through 11) and using a hash join to evaluate the cross-fragment join between p_5^2 and p_5^3 (plans 12 and 13) both yield worse performance than using a merge join that exploits existing ordering (plans 3–7). As can be seen, cost estimation predicts this effect and preserves the order of all candidate plans considered for this query.

While cost estimation preserves the order of plans, for plans 12–14, there is a larger amount of estimation error than for the other plans compared in this experiment. The most likely explanation for this estimation error is a violation of the independence assumption made in Section 7.1.

Together, these results show that in all the cases considered, the error margin between estimated cost and actual cost is low. Thus, the cost model proposed in Chapter 7 has been shown to provide accurate cost estimates. The relative order of candidate plans is also largely preserved by cost estimation, illustrating that the cost model is an effective tool for comparing candidate plans based on their performance. Most importantly, in all cases, cost-based optimization identifies the optimal or a nearly optimal plan in the search space considered. This confirms that cost-based optimization is an effective method for determining a suitable distributed execution plan for a given query and collection.

9.5 Summary

In summary, this chapter has presented a thorough evaluation of the various techniques introduced in this thesis. After validating that a combination of all of these techniques leads to a significant improvement in query performance, each technique is examined individually and its contribution to this performance gain is analyzed. Together, the results presented in this chapter validate the distributed query processing approach taken in this thesis and verify that by using the cost-based optimization procedure, the techniques developed in this thesis lead to significant performance and scalability gains.

Chapter 10

Conclusion

This thesis presents a suite of techniques that use distribution to improve the performance and scalability of XML query evaluation. In the following, a brief summary of these techniques is presented. Then, a final comparison with key contributions of related work is given. Finally, possible directions for future research are outlined.

10.1 Summary

After introducing the necessary background material and discussing related work in this field, a model for specifying a partitioning of an XML collection is introduced. This model supports horizontal fragmentation (based on selection), vertical fragmentation (based on projection), and hybrid fragmentation, based on a combination of both. While the semantics of this fragmentation model are similar to models that are widely used to fragment relational data, the tree structure of XML data leads to particular challenges, which are addressed in this work.

Based on this fragmentation model, a strategy for the distributed evaluation of queries over fragmented collections is then proposed. Distributed query evaluation proceeds by first identifying the fragments that are relevant to a query and then producing a sub-query for each of these fragments. These sub-queries can then be evaluated at the sites holding the corresponding fragments. To obtain the overall query result, a distributed execution plan is defined, which determines how the results of the sub-queries are combined.

A major focus of this thesis is on improving the performance of these distributed execution plans. A suite of techniques are proposed to accomplish this. One of these techniques focuses on pruning the set of fragments that need to be accessed, thus reducing the overall amount of processing that needs to be performed to answer a query. Another technique improves query performance further by skipping irrelevant portions of the remaining fragments. This is achieved by pushing the join operations that combine the results of sub-queries into individual sub-queries.

Together, these techniques open up a large optimization space, in which there are many plan alternatives for a given query and distributed collection. To cope with this, and to obtain the best performance, a cost-based optimization technique is introduced. This technique can be used to accurately predict the cost of evaluating a given distributed execution plan. By enumerating the candidate plans for a query and comparing them based on their estimated cost, the best plan can be chosen.

To fully benefit from the distributed query evaluation techniques presented in this thesis, it is best to partition the collection in a way that allows for the efficient evaluation of the expected query workload. Thus, a heuristic technique is proposed that accomplishes this based on the cost model for distributed execution plans.

Based on an implementation of the techniques from thesis within the context of the XML database system Natix, a comprehensive set of performance experiments is conducted. These confirm that combining the techniques presented here leads to a significant improvement in query performance and scalability when compared to centralized approaches. The techniques from this thesis also perform better than existing distributed approaches for XML query evaluation, because unlike those techniques, this work focuses on end-to-end performance (rather than on a single aspect of performance such as communication cost). Additional experiments verify the individual performance contribution of each of the query evaluation techniques proposed here. They also show that the cost model used during distributed optimization is a good predictor of end-to-end performance.

10.2 Comparison to Related Work

A key difference between the techniques proposed in this thesis and much of related work in the area of distributed XML query processing (e.g., [2, 3, 4, 5]) is this work's focus on distribution as a means to improving query performance and scalability, rather than as a means for integrating multiple collections into a single XML view.

When comparing this work to related techniques that follow a performance motivation (e.g., [31, 57, 39, 33, 40, 124]), a key distinguishing factor is this work's focus on the end-to-end cost of query processing, rather than on a single aspect of this cost such as communication cost. As is shown experimentally in Chapter 9, by focusing on this notion of cost, a performance advantage can be obtained, both compared to centralized query evaluation and to existing distributed techniques.

Another advantage of the techniques presented here is their flexibility with regard to the local query evaluation techniques used to evaluate sub-queries over individual fragments. With this approach, the complexity of local query optimization is avoided and the distributed techniques presented here can benefit from the numerous centralized query evaluation techniques proposed in the literature (e.g., [20, 76, 84, 30, 66, 137, 139, 11, 62, 41, 32]).

Finally, by using a cost-based optimization approach, the full performance benefit of the distributed query evaluation techniques presented in this thesis can be obtained. This is in contrast to much of the existing work, which either does not use a cost model at all (e.g., [39, 33, 40, 124]) or uses a simple, heuristic cost-based approach (e.g., [57]).

10.3 Possible Directions for Future Work

While the system presented in this thesis is designed as an end-to-end solution that improves the performance and scalability of XML query processing through distribution, there are several extensions that could be made for further improvement.

- One such avenue would be a further extension of the query model. Even though the class of queries supported in this work is equivalent to that of related approaches (e.g., [39]) or constitutes a superset of the query models supported by those approaches (e.g., [31, 124]), and even though queries in the class supported by this work form an important building block of queries encountered in many use cases, directly supporting a larger set of XQuery expressions might lead to increased potential for optimization.

- A second area in which this work could be extended is the class of fragmentation schemas that are supported. For this work, the conscious choice was made to focus on fragmentations that constitute a partitioning of the collection. Using this approach, it was shown that query performance can be improved significantly by leveraging distribution. Combining the approaches shown here with techniques that replicate the most heavily loaded fragments (e.g. the techniques described by Machdi et al. [95, 96]) might result in additional performance gains in certain circumstances. Further, it might be possible to extend the definition of vertical fragmentation to take into account the position of a node in the document in addition to the type of the node. This could result in a more flexible definition of fragmentation, which might allow better adaptation to query workloads.

- It might be possible to integrate alternative approaches to distributed query evaluation (such as those based on index structures, e.g., [31]) with the techniques presented in this thesis. By combining both approaches and using a single cost-based optimizer to choose the most appropriate strategy for a given query and collection, further performance gains might be realized.

- The optimizer presented in this thesis focuses on the execution of a single query. By considering the impact of resource contention between multiple queries that are executed simultaneously (such as in [9, 10]), better distributed execution plans might be obtainable. Similarly, the fragmentation algorithm presented in Chapter 8 might be improved by weighting the individual queries in a workload by their frequency.

- Another area that warrants further attention is the management of updates within a distributed XML database as proposed in this thesis. While the techniques presented here are designed not to interfere with update management (for example, by avoiding the use of a replicated index structure), there might be cases where there is a trade-off between the cost of managing updates and the cost of query processing. This is true especially when the techniques from this thesis are combined with approaches that replicate some of the fragments.

References

[1] Amazon Elastic Compute Cloud (EC2), 2006. http://aws.amazon.com/ec2/.

[2] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy query evaluation for Active XML. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 227–238, 2004.

[3] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB Journal*, 17(5):1019–1040, 2008.

[4] Serge Abiteboul, Omar Benjellourn, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: Peer-to-peer data and web services integration. In *Proc. 28th International Conference on Very Large Data Bases*, pages 1087–1090, 2002.

[5] Serge Abiteboul, Angela Bonifati, Grégory Cobéna, Ioana Manolescu, and Tova Milo. Dynamic XML documents with distribution and replication. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 527–538, 2003.

[6] Serge Abiteboul, Georg Gottlob, and Marco Manna. Distributed XML design. In *Proc. 28th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 247–257, 2009.

[7] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *Proc. 27th International Conference on Very Large Data Bases*, pages 591–600, 2001.

[8] Ashraf Aboulnaga and Jeffrey F. Naughton. Building XML statistics for the hidden web. In *Proc. ACM CIKM International Conference on Information and Knowledge Management*, pages 358–365, 2003.

[9] Mumtaz Ahmad, Ashraf Aboulnaga, and Shivnath Babu. Query interactions in database workloads. In *Proc. 2nd International Workshop on Testing Database Systems*, pages 11:1–11:6, 2009.

[10] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and exploiting query interactions in database systems. In *Proc. ACM CIKM International Conference on Information and Knowledge Management*, pages 183–192, 2008.

[11] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. 18th International Conference on Data Engineering*, pages 141–152, 2002.

[12] Alexandre Andrade, Gabriela Ruberg, Fernanda Araujo Baião, Vanessa P. Braganholo, and Marta Mattoso. Efficiently processing XML queries over fragmented repositories with PartiX. In *Current Trends in Database Technology – EDBT 2006 Workshops*, pages 150–163, 2006.

[13] Alexandre Andrade, Gabriela Ruberga, Fernanda A. Baião, Vanessa P. Braganholo, and Marta Mattoso. Partix: processing XQuery queries over fragmented XML repositories. Technical Report ES-691/05, Universidade Federal do Rio de Janeiro, 2005.

[14] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 85–96, 2002.

[15] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Trans. Database Systems*, 29(1):195–232, 2004.

[16] Andrey Balmin, Tom Eliaz, John Hornibrook, Lipyeow Lim, Guy M. Lohman, David Simmen, Min Wang, and Chun Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–319, 2006.

[17] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta J. Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. 30th International Conference on Very Large Data Bases*, pages 60–71, 2004.

[18] Andrey Balmin, Fatma Özcan, Ashutosh Singh, and Edison Ting. Grouping and optimization of XPath expressions in DB2 pureXML. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1065–1074, 2008.

[19] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 1–15, 1986.

[20] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, and Marcus Fontoura. Streaming XPath processing with forward and backward axes. In *Proc. 19th International Conference on Data Engineering*, pages 455–466, 2003.

[21] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, George P. Copeland, and Walter G. Wilson. DB2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[22] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, Hui-I Hsiao, Anant Jhingran, Sriram Padmanabhan, and Walter G. Wilson. An overview of DB2 parallel edition. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 460–462, 1995.

[23] Eric Temple Bell. The iterated exponential numbers. *The Annals of Mathematics*, 39(3):539–557, 1938.

[24] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language

(XPath) version 2.0 (second edition). W3C recommendation, W3C, 2010. http://www.w3.org/TR/2010/REC-xpath20-20101214/.

[25] Mike W. Blasgen and Kapali P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):362–377, 1977.

[26] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language (second edition). W3C recommendation, W3C, 2010. http://www.w3.org/TR/2010/REC-xquery-20101214/.

[27] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2006.

[28] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. Statistics-based parallelization of XPath queries in shared memory systems. In *Advances in Database Technology — EDBT'10, Proc. 14th International Conference on Extending Database Technology*, pages 159–170, 2010.

[29] Sujoe Bose and Leonidas Fegaras. XFrag: A query processing framework for fragmented XML data. In *Proc. 8th International Workshop on the World Wide Web and Databases (WebDB)*, pages 97–102, 2005.

[30] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. 21st International Conference on Data Engineering*, pages 705–716, 2005.

[31] Jan-Marco Bremer and Michael Gertz. On distributing XML repositories. In *Proc. 6th International Workshop on the World Wide Web and Databases (WebDB)*, pages 73–78, 2003.

[32] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[33] Peter Buneman, Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Using partial evaluation in distributed query evaluation. In *Proc. 32nd International Conference on Very Large Data Bases*, pages 211–222, 2006.

[34] Stephen Buswell, Stan Devitt, Angel Diaz, Patrick Ion, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) 1.01 Specification, 1999. http://www.w3.org/TR/REC-MathML/.

[35] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XQuery 1.0: An XML query language (second edition). W3C working group note, W3C, 2007. http://www.w3.org/TR/xquery-use-cases/.

[36] Chee-Yong Chan and Yuan Ni. Content-based dissemination of fragmented XML data. In *Proc. 26th International Conference on Distributed Computing Systems*, page 44, 2006.

[37] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of parallel execution for multi-join queries. *IEEE Trans. Knowledge and Data Engineering*, 8(3):416–428, 1996.

[38] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proc. 17th International Conference on Data Engineering*, pages 595–604, 2001.

[39] Gao Cong, Wenfei Fan, and Anastasios Kementsietsidis. Distributed query evaluation with performance guarantees. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 509–520, 2007.

[40] Gao Cong, Wenfei Fan, Anastasios Kementsietsidis, Jianzhong Li, and Xianmin Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM Trans. Database Systems*, 37(1), 2012. (to appear).

[41] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 623–634, 2003.

[42] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 785–796, 2007.

[43] Alin Deutsch and Val Tannen. MaRS: A system for publishing XML from mixed and redundant storage. In *Proc. 29th International Conference on Very Large Data Bases*, pages 201–212, 2003.

[44] Melvil Dewey. A classification and subject index for cataloguing and arranging the books and pamphlets of a library. 1876.

[45] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proc. 27th International Conference on Data Engineering*, pages 864–875, 2011.

[46] Mary F. Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing sorting and duplicate elimination in XQuery path expressions. In *Proc. 16th International Conference on Database and Expert Systems Applications (DEXA)*, pages 554–563, 2005.

[47] Mary F. Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jérôme Siméon. Highly distributed XQuery with DXQ. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1159–1161, 2007.

[48] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM), 2007. http://www.w3.org/TR/xpath-datamodel/.

[49] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

[50] Guilherme Figueiredo, Vanessa P. Braganholo, and Marta Mattoso. A methodology for query processing over distributed XML databases. Technical Report ES-710/07, Universidade Federal do Rio de Janeiro, 2007.

[51] Guilherme Figueiredo, Vanessa P. Braganholo, and Marta Mattoso. Processing queries over distributed XML databases. *Journal of Information and Data Management*, 1(3):455–470, 2010.

[52] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *ACM SIGMOD Record*, 38(1):43–48, 2009.

[53] Massimo Franceschet. XPathMark: An XPath benchmark for XMark generated data. In *Database and XML Technologies, 3rd International XML Database Symposium (XSym)*, pages 129–143, 2005.

[54] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 149–160, 1996.

[55] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. StatiX: making XML count. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 181–191, 2002.

[56] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 9–18, 1992.

[57] Michael Gertz and Jan-Marco Bremer. Distributed XML repositories: Top-down design and transparent query processing. Technical Report TR-CSE-2003-20, University of California, Davis, 2003.

[58] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[59] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[60] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 160–172, 1987.

[61] Bojan Grošelj and Qutaibah M. Malluhi. Combinatorial optimization of distributed queries. *IEEE Trans. Knowledge and Data Engineering*, 7(6):915–927, 1995.

[62] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: teach a relational DBMS to watch its (axis) steps. In *Proc. 29th International Conference on Very Large Data Bases*, pages 524–535, 2003.

[63] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. 23rd International Conference on Very Large Data Bases*, pages 276–285, 1997.

[64] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. On the intersection of XPath expressions. In *Proc. 9th International Database Engineering and Applications Symposium (IDEAS)*, pages 49–57, 2005.

[65] Michael P. Haustein, Theo Härder, Christian Mathis, and Markus Wagner. DeweyIDs – the key to fine-grained management of XML documents. In *Proc. 20th Brazilian Symposium on Databases*, pages 85–99, 2005.

[66] Sven Helmer and Carl-Christian Kanne. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. 3rd International Conference on Web Information Systems Engineering (WISE)*, pages 215–224, 2002.

[67] Soichiro Hidaka, Hiroyuki Kato, and Masatoshi Yoshikawa. An XQuery cost model in relative form. Technical Report NII-2005-016E, National Institute of Informatics Tokyo, 2005.

[68] Soichiro Hidaka, Hiroyuki Kato, and Masatoshi Yoshikawa. A relative cost model for XQuery. In *Proc. 2007 ACM Symp. on Applied Computing*, pages 1332–1333, 2007.

[69] Jan Hidders, Philippe Michiels, and Roel Vercammen. Optimizing sorting and duplicate elimination in XQuery path expressions. *Bulletin of the EATCS*, 86:199–223, 2005.

[70] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM*, 29(1):68–95, 1982.

[71] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Database Systems*, 1(1):9–32, 1993.

[72] Yannis E. Ioannidis and Younkyung Kang. Randomized algorithms for optimizing large join queries. *ACM SIGMOD Record*, 19(2):312–321, 1990.

[73] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 9–22, 1987.

[74] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4):380–402, 2002.

[75] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.

[76] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB Journal*, 14(2):197–210, April 2005.

[77] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-sensitive reordering of navigational primitives. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 742–753, 2005.

[78] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In *Proc. 16th International Conference on Data Engineering*, page 198, 2000.

[79] Kentarou Kido, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Processing XPath queries in PC-clusters using XML data partitioning. In *Proc. 22nd International Conference on Data Engineering Workshops*, page 114, 2006.

[80] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Optimizing distributed XML queries through localization and pruning. Technical Report CS-2009-13, University of Waterloo, 2009.

[81] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Distributed XML query processing: Fragmentation, localization and pruning. Technical Report CS-2010-02, University of Waterloo, 2010.

[82] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Generating efficient execution plans for vertically partitioned XML databases. *Proc. VLDB Endowment*, 4(1):1–11, 2010.

[83] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Scaling XML query processing: Distribution, localization and pruning. *Distributed and Parallel Database Systems*, 29(5):445–490, 2011.

[84] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: a tree automata-based approach. In *Proc. 29th International Conference on Very Large Data Bases*, pages 249–260, 2003.

[85] Georgia Koloniari and Evaggelia Pitoura. Distributed structural relaxation of XPath queries. In *Proc. 25th International Conference on Data Engineering*, pages 529–540, 2009.

[86] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[87] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Systems*, 25(1):43–82, 2000.

[88] Hiroto Kurita, Kenji Hatano, Jun Miyazaki, and Shunsuke Uemura. Efficient query processing for large XML data in distributed environments. In *Proc. 21st International Conference on Advanced Information Networking and Applications*, pages 317–322, 2007.

[89] Mounia Lalmas. XML retrieval. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 1(1):1–111, 2009.

[90] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. 19th International Conference on Very Large Data Bases*, pages 493–504, 1993.

[91] Thi Thu Thuy Le, Dai Duong Doan, Virendrakumar C. Bhavsar, and Harold Boley. A bottom-up algorithm for query decomposition. *International Journal of Innovative Computing and Applications*, 1(3):185–193, 2008.

[92] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. Optimization of multi-way join queries for parallel execution. In *Proc. 17th International Conference on Very Large Data Bases*, pages 549–560, 1991.

[93] Hui Ma and Klaus-Dieter Schewe. Fragmentation of XML documents. In *Proc. 18th Brazilian Symposium on Databases*, pages 200–214, 2003.

[94] Hui Ma and Klaus-Dieter Schewe. Heuristic horizontal XML fragmentation. In *Proc. 17th International Conference on Advanced Information Systems Engineering*, pages 131–136, 2005.

[95] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. GMX: an XML data partitioning scheme for holistic twig joins. In *Proc. 10th International Conference on Information Integration and Web-based Applications & Services*, pages 137–146, 2008.

[96] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. XML data partitioning strategies to improve parallelism in parallel holistic twig joins. In *Proc. 5th International Conference on Ubiquitous Information Management and Communication (ICUIMC)*, pages 471–480, 2009.

[97] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. 12th International Conference on Very Large Data Bases*, pages 149–159, 1986.

[98] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 84–95, 1986.

[99] Eve Maler, Jean Paoli, C. M. Sperberg-McQueen, François Yergeau, and Tim Bray. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, 2008. http://www.w3.org/TR/2004/REC-xml.

[100] Ashok Malhotra and Paul V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[101] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. 27th International Conference on Very Large Data Bases*, pages 241–250, 2001.

[102] Amélie Marian and Jérôme Siméon. Projecting XML documents. In *Proc. 29th International Conference on Very Large Data Bases*, pages 213–224, 2003.

[103] Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Index vs. navigation in XPath evaluation. In *Database and XML Technologies, 4th International XML Database Symposium (XSym)*, pages 16–30, 2006.

[104] Norman May, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. XQuery processing in Natix with an emphasis on join ordering. In *Proc. 1st International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, pages 49–54, 2004.

[105] T. H. Merrett. Why sort-merge gives the best implementation of the natural join. *ACM SIGMOD Record*, 13(2):39–51, 1983.

[106] Philippe Michiels, George Mihăilă, and Jérôme Siméon. Put a Tree Pattern in Your Algebra. In *Proc. 23rd International Conference on Data Engineering*, pages 246–255, 2007.

[107] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[108] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proc. 32nd International Conference on Very Large Data Bases*, pages 930–941, 2006.

[109] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 539–552, 2008.

[110] Augustus De Morgan. Formal logic: or, the calculus of inference, necessary and probable. 1847.

[111] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 247–258, 1990.

[112] Peter Murray-Rust. Chemical markup language. *World Wide Web Journal*, 2(4):135–147, 1997.

[113] Dan Olteanu, Holger Meuss, Tim Furche, and Franois Bry. Symmetry in XPath. Technical Report PMS-FB-2001-16, Ludwig-Maximilians-Universität München, 2001.

[114] Dan Olteanu, Holger Meuss, Tim Furche, and Franois Bry. XPath: Looking forward. In *XML-Based Data Management and Multimedia Engineering – EDBT 2002 Workshops*, pages 892–896, 2002.

[115] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (3rd ed.)*. Springer Verlag, 2011.

[116] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: a native system for querying XML. In

Proc. ACM SIGMOD International Conference on Management of Data, page 672, 2003.

[117] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London*, 58:240–242, 1895.

[118] Christopher Ré, James Brinkley, Kevin P. Hinshaw, and Dan Suciu. Distributed XQuery. In *Proc. VLDB Workshop on Information Integration on the Web*, pages 116–121, 2004.

[119] Laura Rusu, Wenny Rahayu, and David Taniar. Partitioning methods for multi-version XML data warehouses. *Distributed and Parallel Database Systems*, 25(1):47–69, 2009.

[120] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proc. 28th International Conference on Very Large Data Bases*, pages 974–985, 2002.

[121] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[122] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. 15th International Conference on Data Engineering*, pages 302–314, 1999.

[123] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[124] Dan Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Systems*, 27(1):1–62, 2002.

[125] Arun Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 367–376, 1989.

[126] Arun Swami and Anoop Gupta. Optimization of large join queries. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 8–17, 1988.

[127] Keishi Tajima and Yoshiki Fukui. Answering XPath queries over networks by sending minimal views. In *Proc. 30th International Conference on Very Large Data Bases*, pages 48–59, 2004.

[128] Nan Tang, Guoren Wang, Jeffrey Xu Yu, Kam-Fai Wong, and Ge Yu. Win: An efficient data placement strategy for parallel XML databases. In *Proc. 11th International Conference on Parallel and Distributed Systems*, pages 349–355, 2005.

[129] Nan Tang, Jeffrey Xu Yu, Hao Tang, M. Tamer Özsu, and Peter Boncz. Materialized view selection in XML databases. In *Proc. 14th International Conference on Database Systems for Advanced Applications*, pages 616–630, 2009.

[130] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:114–121, 1972.

[131] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable cardinality forecasts for XQuery. *Proc. VLDB Endowment*, 1(1):463–477, 2008.

[132] Henry S. Thompson, Murray Maloney, David Beech, and Noah Mendelsohn. XML schema part 1: Structures second edition. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[133] Priscilla Walmsley and David C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[134] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. 1st International Conference on Parallel and Distributed Information Systems*, pages 68–77, 1991.

[135] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Advances in Database Technology — EDBT'02, Proc. 8th International Conference on Extending Database Technology*, pages 590–608, 2002.

[136] Yuqing Wu, Jignesh M .Patel, and H. V. Jagadish. Using histograms to estimate answer sizes for XML queries. *Information Systems*, 28(1-2):33–59, 2003.

[137] Masatoshi Yoshikawa and Toshiyuki Amagasa. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Technology*, 1:110–141, August 2001.

[138] Yaxin Yu, Guoren Wang, Ge Yu, Gang Wu, Junan Hu, and Nan Tang. Data placement and query processing based on RPE parallelisms. In *Proc. 27th International Computer Software and Applications Conference*, pages 151–156, 2003.

[139] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 425–436, 2001.

[140] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing XML queries. In *Proc. 31st International Conference on Very Large Data Bases*, pages 289–300, 2005.

[141] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. 20th International Conference on Data Engineering*, pages 54–65, 2004.

[142] Ning Zhang and M. Tamer Özsu. Optimizing correlated path queries in XML languages. Technical Report CS-2002-36, University of Waterloo, 2002.

[143] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and fast cardinality estimation for XPath queries. In *Proc. 22nd International Conference on Data Engineering*, pages 61–72, 2006.

[144] Ying Zhang and Peter Boncz. XRPC: interoperable and efficient distributed XQuery. In *Proc. 33rd International Conference on Very Large Data Bases*, pages 99–110, 2007.

[145] Ying Zhang and Peter Boncz. XRPC: distributed XQuery and update processing with heterogeneous XQuery engines. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1331–1336, 2008.

[146] Ying Zhang, Nan Tang, and Peter Boncz. Efficient distribution of full-fledged XQuery. In *Proc. 25th International Conference on Data Engineering*, pages 565–576, 2009.

[147] Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet. Parallel query processing in DBS3. In *Proc. 2nd International Conference on Parallel and Distributed Information Systems*, pages 93–102, 1993.

[148] Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet. Parallel query processing with zigzag trees. *VLDB Journal*, 2(3):277–301, 1993.

Index