Rewriting and Evaluating XPath Queries Using Views

Rui Zhou
B.E. (NEU, China)
M.E. (NEU, China)
A dissertation

submitted to the Faculty of Information and Communication Technologies
Swinburne University of Technology
In partial fulfillment of the requirements
For the degree of
Doctor of Philosophy

October 2010
Abstract

Rewriting and evaluating queries using views, also known as answering queries using views, is to utilize previously defined and materialized views to evaluate queries in order to save the cost of accessing large real database. It is a classic problem, and appears in many applications, such as query optimization, data integration, data warehouse and query caching. With the prevalence of XML technologies, rewriting XML queries using XML views has caught the attention of both researchers and system designers, and is believed to be a promising technique in web and database applications. Since XPath serves as the core sub-language of the major XML query languages, such as XQuery and XSLT, it is of immense value to study how to rewrite and evaluate XPath queries using XPath views.

In this thesis, contained rewriting for XPath queries using XPath views is studied. Contained rewriting is proposed to provide best effort to answer a user’s query when equivalent rewriting does not exit. Given a query and a view, there may be an exponential number of contained rewritings. Some of them are redundant because the query answers produced by these redundant contained rewritings can be covered by other contained rewritings. Obviously, it is unnecessary to evaluate redundant contained rewritings on materialized views. We investigate the problem and propose a series of methods to discover all irredundant contained rewritings, and we show the correctness of the proposed methods. Furthermore, we study the evaluation of a group of contained rewritings on a given materialized
view. The problem is transformed into evaluating component patterns, and four pruning rules and three heuristic rules are developed to speed up the evaluation dramatically. When a schema is available, we discuss how to utilize schema information for rewriting a query using a view, because a query, not answerable using a view in general, may be answerable using the view if the query and the view are complied with schema constraints. Finally, a set of filtering techniques are given to select promising views to answer users’ query. Indexes and three algorithms are developed to quickly identify whether a view should be used or not.
Acknowledgment

Firstly, I would like to express my deep and sincere gratitude to my supervisor, Professor Chengfei Liu, for his patience, encouragement and help. He brought me into research world and taught me how to do solid research from critical reasoning to how to present an idea in a paper. His thoughtful and valuable comments have covered every corner in my study and benefited me throughout these years. I cannot finish my study without his guidance.

Besides my supervisor, I would like to thank Professor Guoren Wang at Northeastern University of China, who mentored me for my master degree and also acted as my associate supervisor. He had a remarkable influence on my career, because, without his support and encouragement, I will not be interested in pursuing a PhD. degree.

During the study of my PhD. study, Professor Jeffrey Xu Yu at Chinese University of Hong Kong, Dr. Junhu Wang at Griffith University of Australia, Dr. Wei Wang at New South Wales University of Australia, Associate Professor Weifa Liang at Australian National University, gave me tremendous help and insightful comments on my research work.

In addition to the mentors above, the members in Swinburne Web & Data Engineering group were always available for discussion. Especially, Dr. Jianxin Li has given me many helpful advice on my research over the three years. Dr. Xiaohui Zhao has provided me a lot of valuable information to apply in both
research and daily life. The other members Chi Yang, Hai Huang, Jiajie Xu, Shangfeng Hu, Sira Yongchareon, Saiful Islam have been great people to share ideas. I also would like to thank my friend Baichen Chen at Australian National University for his support and encouragement.

Professor Jun Han and Dr. Jinjun Chen was kind to serve in my PhD. review committee and to provide helpful comments on my research work. Ms. Mandish Webb, Ms. Li Sim Loh and Ms. Gillian Foster have been very nice to help me with all kinds of research problems.

My parents have been always a source of energy for me in my life. Finally, but more importantly, I wish to thank my wife, Ms. Min Zhao. When I met obstacles, she was always there to support me and encourage me; when I had a little success, she was also there to calm me down. I could not finish my PhD. study without her.
Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Name: Rui Zhou

Signature: _________
Contents

Abstract ii

Acknowledgment iv

Declaration vi

1 Introduction 1
  1.1 Application Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
    1.2.1 Problem Background . . . . . . . . . . . . . . . . . . . . . 5
    1.2.2 Research Problems and Challenges . . . . . . . . . . . . . 6
  1.3 Contributions of This Thesis . . . . . . . . . . . . . . . . . . . . 8
  1.4 Structure of This Thesis . . . . . . . . . . . . . . . . . . . . . . 11

2 Preliminaries 13
  2.1 XML Trees and XPath Tree Patterns . . . . . . . . . . . . . . . . . 13
    2.1.1 XML Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    2.1.2 XPath Tree Pattern . . . . . . . . . . . . . . . . . . . . . . 15
    2.1.3 Matching an XPath Tree Pattern on an XML Tree . . . . . 16
  2.2 XPath Query Rewriting Using View . . . . . . . . . . . . . . . . . 18
    2.2.1 Equivalent Rewriting . . . . . . . . . . . . . . . . . . . . . 18
2.2.2 Contained Rewriting .............................................. 19
2.2.3 Formulations of ER, CR, MCR and IMCR ................. 22
2.2.4 Trimming the View ............................................ 23

3 Related works ......................................................... 24
3.1 View Semantics .................................................... 26
3.2 Selecting Views to Materialize .................................. 27
3.3 Materializing the Views .......................................... 29
  3.3.1 Query as View ............................................... 29
  3.3.2 Index as View ............................................... 30
3.4 Rewriting Queries Using Views .................................. 32
  3.4.1 Single View .................................................. 32
  3.4.2 Multiple Views .............................................. 34
3.5 XPath Containment ................................................ 37

4 Finding the IMCR ...................................................... 39
4.1 Motivation ......................................................... 39
4.2 Avoid Containment Check for a Union of CRs ................. 40
4.3 Minimizie the Candidate CR Set ............................... 43
  4.3.1 Represent a CR ............................................. 43
  4.3.2 A Sufficient Condition to Eliminate Redundant CRs .... 45
  4.3.3 Algorithms to eliminate part of redundant CRs ......... 49
4.4 Experiments ......................................................... 53
  4.4.1 The number of Irredundant CRs ............................. 56
  4.4.2 Performance on Finding the IMCR ......................... 59
4.5 Summary .......................................................... 61
List of Tables

1.1 Examples of CR, MCR, IMCR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
# List of Figures

2.1 An example of useful embedding .............................................. 21
2.2 An example of trimming the view .............................................. 23

4.1 Different useful embeddings producing the same CR ...................... 43
4.2 Comparing anchor nodes and link nodes ................................. 45
4.3 Component pattern containment on the same path ....................... 46
4.4 Component pattern containment on different paths ..................... 47
4.5 The number of CRs and ICRs varing $\beta_2$ ............................. 57
4.6 The number of CRs and ICRs varing $\beta_1$ ............................. 58
4.7 Performance on Finding the IMCR ........................................... 60

5.1 CATs of the IMCR ................................................................. 69
5.2 Average Case Study ............................................................. 77
5.3 Best Case Study ................................................................. 79
5.4 Worst Case Study ............................................................... 80

6.1 Query $q$ is answerable using view $v$ under schema $G$ ................ 83
6.2 An unusable view ..................................................................... 84
6.3 The reason to eliminate ad-edges .......................................... 92
6.4 Rewriting under Recursive Schema ..................................... 96

7.1 The framework of answering queries without filtering ................ 99
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>The framework of answering queries with filtering</td>
<td>100</td>
</tr>
<tr>
<td>7.3</td>
<td>Index $I_{pc}$ and $I_{ad}$ built on view $V$</td>
<td>104</td>
</tr>
<tr>
<td>7.4</td>
<td>Basic idea of filtering algorithms</td>
<td>106</td>
</tr>
<tr>
<td>7.5</td>
<td>Filtering queries with respect to equivalent rewriting</td>
<td>109</td>
</tr>
<tr>
<td>7.6</td>
<td>Drawbacks of Lazy Algorithm</td>
<td>110</td>
</tr>
<tr>
<td>7.7</td>
<td>$Q'$ is a contained rewriting for $Q$ using $V$</td>
<td>114</td>
</tr>
<tr>
<td>7.8</td>
<td>Considering wildcard nodes in queries and views</td>
<td>115</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Rewriting and evaluating queries using views (also known as answering queries using views) is to utilize previously defined (possibly materialized) views to evaluate queries in order to save the cost of accessing large real database or provide a privacy-preserving publishing [RMSR04]. It is a classic problem, and appears in many applications, such as query optimization, data integration, data warehouse and query caching. With the prevalence of XML technologies, rewriting XML queries using XML views has caught the attention of both researchers and system designers, and is believed to be a promising technique in web and database applications. Since XPath [BBC+07] serves as the core sub-language of the major XML query languages such as XQuery [BCF+07] and XSLT [Kay07], we focus on rewriting and evaluating XPath queries using XPath views.

1.1 Application Scenarios

We start from the backing applications for answering queries using views in this section so that we can realize the importance of studying this topic. Most of
the following applications are originally drawn from the relational database context [Hal01], but they fit well in XML database management. We will explain in detail as we dive into each application scenario.

The first class of applications, in which we encounter the problem of rewriting and evaluating queries using views, is query optimization and database design. For query optimization, evaluating a query using previously materialized views can accelerate query processing because part of the evaluation for the query may have already been completed at the time when the views were prepared. Furthermore, to some extent, indexes can be modeled as precomputed views and deciding which index to use is a part of the query rewriting problem. In the context of database design, view definitions provide a mechanism for separating the physical storage of the data from its logical schema. This separation enables us to change the storage schema of the data (i.e., the physical view) without changing its logical schema. Given these choices of the storage, the problem of finding out a query execution plan equals to figuring out how to use the views to answer the query. In native XML databases, structural join [AKJP+02] results can be reused if they have been computed. We can also separate the storage of an XML document from its physical view. Some storage methods include parent-child first or sibling-first corresponding depth-first traversal or breath-first traversal of the XML document from the root.

A second class of applications in which the problem arises is data integration. Data integration systems provide a universally-agreed query interface for a number of heterogeneous data sources. Those data sources may reside within an enterprise database or on the web. Data integration systems can relieve users from having to explicitly locate sources relevant to a query, access each data source individually, and manually combine the needed data from multiple sources. When leveraging data integration systems, users are not required to
issue queries according to the schema where the data is stored, but to follow a mediated schema. The mediated schema is designed by experts for a specific data integration application, including the representative aspects of the domain. The data of the mediated schema are not physically stored in the data integration system. Instead, the system includes a set of source descriptions that provide mappings between the relations in the source schemata and the relations in the mediated schema. Most data integration systems follow an approach, called local as view, in which the data of the sources are considered as views over the mediated schema. Consequently, the problem of rewriting a user query, issued on the mediated schema, into a query that can be evaluated on a source schema becomes the problem of answering queries using views. In the XML context, due to the flexibility of designing an XML schema, it is of high possibility that different data sources have different XML schema. Therefore, to query heterogeneous XML data sources, we need to rewrite user’s queries with individual XML views.

Another application area is data warehouse design, where we often need to select a set of views and indexes on the views to materialize in the warehouse. Similarly, in the web site design, the performance of a web site can be significantly improved by choosing a set of views to materialize. In both of these problems, the first step is to determine a choice of a set of views so that the views are as sufficient as possible to answer the queries we expect to receive over the data warehouse or the web site. As a result, this problem also becomes the problem of rewriting and evaluating queries using views. In the XML context, there are few data warehouses built on XML data, but for web site design, it is important to materialize some selected pages if the pages need to be generated in the run time, especially when we have to accommodate a large number of requests from web users.

Finally, rewriting and evaluating queries using views plays an important role
in developing methods for semantic data caching in client-server systems. Under this scenario, the data cached at the client is modeled semantically as a set of queries, together with a set of physical data pages. Hence, deciding which data needs to be shipped from the server in order to answer a given query requires an analysis of which parts of the query can be answered by the physically cached views. As XML is the standard format of publishing and exchanging data, query caching is a right application that can substantially benefit from rewriting and evaluating queries using views, especially when the base XML document is large, not easy to be shipped. By having the view rewriting mechanism on the client side, (1) we can save part of the transmission cost, because only absent parts (subtrees), necessary for computing the new queries but not covered by the stored views, are needed to be shipped from the server side; (2) we can also save some the computation cost on the client side, because part of the computation has been planted in the materialized views.

1.2 Problems

We have seen the importance of rewriting and evaluating queries using views in various applications. Let us narrow down to the specific problems for answering XPath queries using XPath views. We first introduce the problem background including a set of important concepts used throughout this thesis and also the progress achieved from previous studies. In the later part, we highlight a number of research problems that still need to be investigated. The thesis is dedicated to find the final solution to those open problems.
1.2.1 Problem Background

In the literature, fruitful research achievements have been made on rewriting XML queries using XML views \([BÖB^+04, MS05, XÖ05, LWZ06, ACG^+09, ABMP07, CR02, ODPC06]\). All of these works except \([LWZ06, ABMP07]\) focus on equivalent rewriting (ER), which means, given a view and a query, to evaluate the rewritten query (called compensation query in \([BÖB^+04, XÖ05]\)) on the materialized view will produce the same set of answers as to evaluate the original query on the database. The main purpose of these works is to use materialized views to speed up query processing and save computation cost.

However, an equivalent rewriting may not always exist, and moreover part of answers covered by the view are still valuable. As a result, contained rewriting (CR) can be used to provide possible answers with best effort. In the work \([LWZ06]\), the authors also studied contained rewriting. They point out that, in data integration scenario, data on the view may be integrated from multiple data sources. And due to the limited coverage of data sources, a view itself is usually not complete. As a result, it is impossible to find an equivalent rewriting using this view. However, it is still reasonable to efficiently give users part of the query answers with the view. They propose to find a maximal contained rewriting (MCR) for the query to retrieve all possible answers from the view. An MCR is a set of contained rewritings (CRs) and no other CR set can produce more answers than the MCR. Note that, a contained rewriting does not refer to a compensation query, it is defined as a merged query of the view and a compensation query.

In the meanwhile, an MCR may contain redundant CRs, i.e. CRs that are contained in other CRs, which means answers produced by redundant CRs can be covered by other CRs that are also belonging to the MCR set. Obviously, it is unnecessary to evaluate the redundant CRs. We call an MCR with no redundant
CRs an IMCR (irredundant maximal contained rewriting).

We now give an example to illustrate the concept of CR, MCR, IMCR and compensation patterns in Table 1.1. Formal definitions will be given in Section 2.2.3. In the table, $CR_1$ and $CR_2$ are contained rewritings for query $q_1$ using view $v$, with compensation patterns be $/b$ and $//a/b$ respectively. The MCR $\{CR_1, CR_2\}$ is also an IMCR, because $CR_1$ and $CR_2$ are not contained in each other. However, if we slightly change $q_1$ into $q_2$, then for $q_2$ and view $v$, we have another two CRs, $CR_3$ and $CR_4$. In this case, the MCR $\{CR_3, CR_4\}$ is not an IMCR, because $CR_4$ is contained in $CR_3$. (For the readers not familiar with XPath query containment, the example can be reviewed after finish reading Section 2.1.3.)

<table>
<thead>
<tr>
<th>view $v$</th>
<th>query $q_1$</th>
<th>compensation</th>
<th>$CR_1$</th>
<th>compensation</th>
<th>$CR_2$</th>
<th>MCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>query $q_1$</td>
<td>//a</td>
<td>//a/b</td>
<td>/a</td>
<td>//a/b</td>
<td>/b</td>
<td>${CR_1, CR_2}$</td>
</tr>
<tr>
<td>$CR_1$</td>
<td>/a//b</td>
<td></td>
<td>/a/b</td>
<td></td>
<td>/b</td>
<td>{CR_1, CR_2}</td>
</tr>
<tr>
<td>compensation</td>
<td>//a//b</td>
<td></td>
<td>/a//b</td>
<td></td>
<td>/b</td>
<td>{CR_1, CR_2}</td>
</tr>
<tr>
<td>$CR_2$</td>
<td>/a/b</td>
<td></td>
<td>/a</td>
<td></td>
<td>/b</td>
<td>{CR_1, CR_2}</td>
</tr>
<tr>
<td>compensation</td>
<td>//a//b</td>
<td></td>
<td>/a</td>
<td></td>
<td>/b</td>
<td>{CR_1, CR_2}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>view $v$</th>
<th>query $q_2$</th>
<th>compensation</th>
<th>$CR_3$</th>
<th>compensation</th>
<th>$CR_4$</th>
<th>MCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>query $q_2$</td>
<td>//a//b</td>
<td></td>
<td>/a//b</td>
<td></td>
<td>/b</td>
<td>{CR_3, CR_4}</td>
</tr>
<tr>
<td>$CR_3$</td>
<td></td>
<td></td>
<td>/a//b</td>
<td></td>
<td>/b</td>
<td>{CR_3, CR_4}</td>
</tr>
<tr>
<td>compensation</td>
<td></td>
<td></td>
<td>/a//b</td>
<td></td>
<td>/b</td>
<td>{CR_3, CR_4}</td>
</tr>
<tr>
<td>$CR_4$</td>
<td></td>
<td></td>
<td>/a//b</td>
<td></td>
<td>/b</td>
<td>{CR_3, CR_4}</td>
</tr>
<tr>
<td>compensation</td>
<td>//a//b</td>
<td></td>
<td>/a</td>
<td></td>
<td>/a//b</td>
<td>{CR_3, CR_4}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IMCR</th>
<th>{CR_1, CR_2}</th>
<th>MCR</th>
<th>{CR_3, CR_4}</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>IMCR</th>
<th>{CR_1, CR_2}</th>
</tr>
</thead>
</table>

1.2.2 Research Problems and Challenges

We will investigate the following research problems in this thesis:

1. Given a view definition $v$ and a query $q$, how to efficiently find the irredundant maximal contained rewriting (IMCR) for $q$ using $v$;

2. Given the materialization of view $v$ as $v_m$, how to efficiently evaluate the IMCR on the materialized view $v_m$ to obtain the query result;
CHAPTER 1. INTRODUCTION

3. Given a schema which query $q$ and view $v$ conform to, how to rewrite $q$ using $v$ under schema restriction;

4. Given a large number of views in repository, how to efficiently filter out irrelevant views and select relevant views to answer a given query.

As to problem (1), it has been mentioned in the previous work [LWZ06], but not well studied. Firstly, irredundant CR is defined as “not contained in any other CR”. However, according to common sense, if a CR is not contained in any other CR, but is contained in a union of some other CRs, the CR should still be regarded as redundant. Secondly, the previous work proposed a strategy to find the IMCR by finding useful embeddings (a technique we will review in Section 2.2.2) from query $q$ to view $v$ and then check some conditions on these useful embeddings to recognize those embeddings that are the only embeddings capable to produce irredundant CRs. However, we show that the strategy is not sufficient with an example, and moreover, after analyzing the counterexample, we point out that similar strategies (checking conditions of produced embeddings) are all unlikely to be sufficient.

As to problem (2), to our knowledge, it has not been considered in the state-of-the-art works. An intuitive solution is to evaluate each CR from the IMCR set on the materialized view, and then combine the results of these CR queries. This solution is correct, but not efficient, because the number of CRs tends to be exponential in terms of the number of paths in the query $q$. However, our observation is that these exponential number of CRs are composed by a linear number of subpatterns bounded by $|N_q|$, the number of nodes in the query $q$. Therefore, taking advantage of the special structural characteristics of the irredundant CRs, we are able to develop very efficient algorithms.
As to problem (3), it requires to consider schema constraints when rewriting and evaluating a query using a view. A query, not answerable by a view in general, may be answered by a view under schema restrictions, because schema information imposes a set of constraints onto the query, and the query has to conform to the given schema. The problem becomes how to leverage schema information to transform the query and the view, and then find a rewriting for the transformed query using the transformed view. Our idea is to eliminate wildcard nodes and “//” edges in the query and the view to make these uncertain path steps concrete. Then, established methods can be applied onto the transformed queries and views to solve the problem.

As to problem (4), in real applications, a large number of queries may be submitted to a system at one time, which challenges the scalability and robustness of the system. Assume we have already materialized a set of views, we still need to figure out which queries can be answered with the materialized views in storage and which queries have to be evaluated on the base documents. A straightforward solution is to perform pattern match from the query to the view, resulting in $O(|N_q|N_v|)$ time for equivalent rewriting and $O(|N_q||N_v|^2)$ time for contained rewriting, where $N_q$ and $N_v$ are the number of nodes in the query $q$ and the view $v$ respectively. However, we notice that we are able to develop a set of $O(|N_q|)$ algorithms to filter out most irrelevant views with low computation cost.

1.3 Contributions of This Thesis

The main contributions of this thesis are as follows:

1. We address how to find the irredundant maximal contained rewriting for rewriting a query using a view. We first prove a CR independence property
to avoid checking containment between one CR and a union of CRs, and hence the problem is reduced to pairwise containment check for all the produced CRs. We then propose a set of methods to minimize the number of candidate CRs in order to save the computation for pairwise comparisons. The minimizing methods include (1) representing a CR by a set of link nodes so that the same CR produced by different useful embeddings is recorded only once; (2) using a sufficient condition, CAT containment, to prune part of redundant CRs; (3) developing three algorithms, pFirst, qFirst, Hybrid together with accelerating heuristics to detect CAT containment. We also compare our solution with the naive solution (a brute-force pairwise containment check) using experiments. The experiments demonstrate the efficiency of our methods.

2. We investigate how to evaluate the irredundant maximal contained rewriting on materialized views. After finding out all the irredundant CRs, the problem is transformed into how to efficiently evaluate a set of CATs (of those irredundant CRs) on the materialized views. We first give a naive algorithm, which evaluates the CAT of each irredundant CR on the materialized view and combines the results. Then we propose a basic algorithm driven by the observation that one component pattern may be shared by a number of CAT patterns, and the component pattern should be evaluated only once for multiple CATs. We further propose four pruning rules and three heuristic rules to reduce the number of CATs needed to be evaluated. These pruning rules and heuristic rules can be applied on the fly during the computation. A set of experiments show that the basic algorithm with optimizing rules is the best choice. We also point out that our evaluation algorithms for a set of CATs do not rely on specific view materialization.
Therefore, the algorithms are able to work seamlessly together with either subtree fragments as views or node lists as views, and can be integrated into different systems with no difficulty.

3. We propose a set of transformation methods to eliminate wildcard nodes and “//” edges in a pattern according to the schema information. Wildcard nodes can be replaced by concrete labels from the schema by traversing the pattern twice (one bottom-up and one top-down) to instantiate the wildcard nodes. After that, we propose how to replace “//” edges with a subgraph from the schema graph so that “//” edges are replaced by a concrete path or a union of paths. We also modify the condition for detecting the existence of a rewriting for queries and views as the transformed patterns. Finally, recursive schema is considered and how to use recursive schema to confine queries and views is discussed. The idea is to remain one recursive loop in the pattern, and mark the loop start node and loop end node in the pattern. When finding rewriting for queries and views with loop, further conditions are added and need to be checked.

4. We develop a set of filtering techniques for selecting promising views to rewrite a query. We study the filtering function for both equivalent rewriting and contained rewriting. The basic idea is to verify whether the structural relationships in a query could be satisfied in a view, given that label preserving and structure preserving are the key conditions in discovering a homomorphism (for finding an equivalent rewriting) or a useful embedding (for finding a contained rewriting). We use index to capture the structural relationships in the view, and develop two algorithms for equivalent rewriting, i.e. Lazy Algorithm and Eager Algorithm. Moreover, Eager Algorithm can be modified to support contained rewriting. After studying all of the
above for queries and views in subset $XP^{(//)}$, we further give the solution for queries and views in XPath subset $XP^{(//,\star)}$ including wildcards.

### 1.4 Structure of This Thesis

The rest of this thesis is organized as follows:

- **Chapter 2** introduces how to model XML documents as XML trees, XPath queries as tree pattern queries, and also gives the preliminaries on how to find contained rewritings for a query using a view.

- **Chapter 3** introduces the related works. The related works fall into the following aspects how to select representative views to materialize, in what form the materialized views should be stored in database, how to select relative views to answer a query, how to use a single view to find an equivalent rewriting, how to combine multiple views to find an equivalent rewriting, and also XPath containment which is closed related to query rewriting using views.

- **Chapter 4** focuses on the problem of finding the irredundant maximal contained rewriting for a query given a view. The result is a set of irredundant contained rewritings. Contained rewriting independence property is proposed first, and then CAT containment is introduced to prune part of redundant contained rewritings. Afterwards, we develop three algorithms with heuristics to check CAT containment and report experiment results.

- **Chapter 5** presents how to evaluate the irredundant maximal contained rewriting on materialized views. The main idea is to optimize the evaluation of multiple rewritings. We introduce naive algorithm, basic algorithm
and optimizing techniques along the line. The optimize techniques will be elaborated with four pruning rules and three heuristic evaluation rules.

- Chapter 6 addresses how to use schema information to enhance the possibility of rewriting a query using a view. The first step is to eliminate wildcard nodes, followed by a strategy to eliminate “//” edges. After that, we propose to chase the result patterns with two basic constraints so that the patterns do not have uncovered conditions. Finally, we propose how to modify rewriting conditions to accommodate the transformed patterns.

- Chapter 7 proposes a set of filtering techniques to select promising views for answering queries. Both equivalent rewriting and contained rewriting are studied. An index to capture the relationship of nodes in the view is proposed first, then two algorithms to filter equivalent rewriting and one algorithm to filter contained rewriting is discussed. Finally, queries with wildcard nodes are considered.

- Chapter 8 concludes the thesis, summarize the contributions, and provides a discussion of interesting directions for future works.
Chapter 2

Preliminaries

In this chapter, we introduce the preliminaries of the thesis, including a set of basic concepts, some notations and a few necessary techniques. Firstly, we formally introduce how to model XML documents and XPath queries as trees and tree patterns, and then we point out that evaluating an XPath query on an XML document can be modeled as matching a tree pattern to a tree. After that, we focus on the concepts of equivalent rewriting and contained rewriting and the techniques used to find them. The main focus of the thesis is on contained rewriting, but we introduce equivalent rewriting before contained rewriting in order to go from the easy to the difficult. Finally, we formalize the concepts of equivalent rewriting, contained rewriting, maximal contained rewriting and irredundant maximal contained rewriting.

2.1 XML Trees and XPath Tree Patterns

In this section, we formally define XML data as trees and XPath queries as tree patterns. As a result, to evaluate an XPath query on an XML document equals to match an XPath tree pattern to an XML data tree.
2.1.1 XML Trees

In the literature, an XML document is often modelled as an unordered tree with nodes labelled from an infinite alphabet $\Sigma$. The label of each node corresponds to an XML element name, an attribute name or a data value. and the root node of the tree corresponds the root element of the document. We slightly modify the model by adding a new root with a unique label $r \in \Sigma$ to the tree, serving as the document root. In this way, the root element node in the previous model becomes a single child of this document node (we have ignored processing instruction nodes and comment nodes), and every XML document starts with a document root node labeled $r$. This new tree model also complies with the W3C DOM standard [DOM]. We will see the aim of this modification later when we study how to match an XPath pattern to an XML data tree. We denote all possible trees over alphabet set $\Sigma$ as $T_\Sigma$. Now, we have the following definition for an XML tree:

**Definition 2.1** An XML document is a tree $t = (N_t, E_t, r_t, \Sigma)$, where

- $N_t$ is the node set, and $\forall n \in N_t$, $n$ has a label in the alphabet $\Sigma$, denoted as $\text{label}(n)$;

- $E_t$ is the edge set;

- $r_t \in N_t$ is the root node of $t$, and $\text{label}(r_t) = r$;

Here, in an XML document, although the elements directly under a certain XML element are often ordered, we ignore the order when we process the XML document, such as building index for the document, evaluating queries on the document, etc. Therefore, an XML document is modelled as an unordered tree.
CHAPTER 2. PRELIMINARIES

2.1.2 XPath Tree Pattern

XPath [BBC+07] is a language recommended by W3C for locating elements in an XML document. XPath plays an important role, because it is the core subclass of other popular XML query languages, such as XQuery [BCF+07], XSLT [Kay07], XPointer [DMJ01] etc. XPath is currently in version 2.0.

In the thesis, we mainly focus on a subset of XPath language featuring child axes (/), descendant axes (//), predicates ([ ]) and wildcards (*). This subset is the most often used subset of XPath queries, and XPath queries in this subset can be represented by the following grammar:

\[
p \rightarrow .|l| *|p/p|p//p|p[p]
\]

where "." denotes the current context node, "l" is a label from alphabet \(\Sigma\). We denote this subset as \(XP\{/,,\}\). When an XPath query is an absolute path, namely starting with "/" or "//", we intentionally add a label \(r\) before "/" or "//" (\(r\) corresponds to the document root label), so every XPath query (either absolute path query or relative path query) starts with a label. For example, queries /a/b and //a//b, which do not conform to the defined grammar, can be now rewritten into r/a/b and r//a//b, and hence be captured. In this way, an XPath query corresponds to a unique tree pattern query, see the following:

**Definition 2.2** An XPath query \(q\) can be expressed as a tree pattern \((N_q, E_q, r_q, d_q, \Sigma_q)\), where

- \(N_q\) is the node set, and \(\forall n \in N_q\), \(n\) has a label in a finite alphabet \(\Sigma_q \cup \{\ast\}\) (\(\Sigma_q \subset \Sigma\)), denoted as label(\(n\));
- \(E_q\) is the edge set. We use the term "pc-edge" ("ad-edge") to represent the type of an edge, "/" ("//");

• $d_q$ is the distinguished (also called return or answer) node of the query (identified with a circle in the figures throughout this paper);

• $r_q$ is the root node of the query. (If $q$ starts with “/” or “//”, we have added a virtual root node with a unique label $r$, so that every query corresponds to a unique tree pattern.)

Similarly, an XPath view $v$ can be expressed as a 5-tuple, $(N_v, E_v, r_v, d_v, \Sigma_v)$. For ease of discussion, we introduce the following definition.

**Definition 2.3** Given a view query $v = (N_v, E_v, r_v, d_v, \Sigma_v)$ and two nodes $n_1, n_2 \in N_v$: (i) $pc(n_1, n_2)$ holds in $v$ if $n_1$ is the parent node of $n_2$; (ii) $ad(n_1, n_2)$ holds in $v$ if $n_1$ is an ancestor node of $n_2$.

Obviously, we have the following corollaries:

• $(n_1, n_2)$ is a pc-edge. ⇔ $pc(n_1, n_2)$ is true.

• $(n_1, n_2)$ is a ad-edge. ⇒ $ad(n_1, n_2)$ is true.

• $pc(n_1, n_2)$ is true. ⇒ $ad(n_1, n_2)$ is true.

### 2.1.3 Matching an XPath Tree Pattern on an XML Tree

The result of evaluating an XPath query $q \in XP^{(/,//,)}$ on an XML tree $t \in T_\Sigma$, is a set of nodes in the tree $t$, denoted as $q(t)$. To find $q(t)$, equals to find mappings from the tree pattern query $q$ to the tree $t$, which can be represented as $q(t) = \{f(d_q) | f$ is a certain mapping from $q$ to $t\}$.

**Definition 2.4** An mapping from a tree pattern $q = (N_q, E_q, r_q, d_q, \Sigma_q)$ to a tree $t = (N_t, E_t, r_t, \Sigma)$, $f : N_q \rightarrow N_t$, satisfies:
• Root preserving: \( f(r_q) = r_t \);

• Label preserving: \( \forall n \in N_q, \text{label}(n) = \text{label}(f(n)) \);

• Structure preserving: \( \forall (n_1, n_2) \in E_q, \text{if} \ (n_1, n_2) \text{ is a pc-edge in } q, \text{ then } f(n_1) \text{ is the parent node of } f(n_2) \text{ in } t; \text{ otherwise } f(n_1) \text{ is an ancestor node of } f(n_2) \text{ in } t \) including the case \( f(n_1) \) being the parent of \( f(n_2) \).

For any two tree pattern query \( p \) and \( q \), \( p \) is said to be contained in \( q \), denoted as \( p \subseteq q \), iff \( \forall t \in T_\Sigma, p(t) \subseteq q(t) \). To examine \( p \subseteq q \), we do not need to go through all data tree instance. The existence of a homomorphism from \( q \) to \( p \) is a sufficient and necessary condition of \( p \subseteq q \) for XPath subset \( XP^{(//)} \) (proved in [MS04]).

A homomorphism is a mapping from \( N_q \) to \( N_p \) satisfying all three conditions (root preserving, label preserving and structure preserving), together with an additional condition that the distinguished node of \( q \) matching the distinguished node of \( p \). We give the detailed definition as follows:

**Definition 2.5** A homomorphism from an XPath tree pattern
\( q = (N_q, E_q, r_q, d_q, \Sigma_q) \) to another XPath tree pattern \( v = (N_v, E_v, r_v, d_v, \Sigma_v) \), \( h : N_q \to N_v \), satisfies:

• Root preserving: \( h(r_q) = r_v \);

• Label preserving: \( \forall n \in N_q, \text{label}(n) = \text{label}(h(n)) \);

• Structure preserving: \( \forall (n_1, n_2) \in E_q, \text{if} \ (n_1, n_2) \text{ is a pc-edge in } q, \text{ then } h(n_1) \text{ is the parent node of } h(n_2) \text{ in } v; \text{ otherwise } h(n_1) \text{ is an ancestor node of } h(n_2) \text{ in } v \) including the case \( h(n_1) \) being the parent of \( h(n_2) \).

• Distinguished node preserving: \( h(d_q) = d_v \)
To sum up, checking query containment is to find a mapping from a pattern to another pattern, while computing the answer of a query is to find a set of mappings from a pattern to a data tree.

2.2 XPath Query Rewriting Using View

In this section, we introduce basic techniques to find equivalent rewritings and contained rewritings. We will also introduce some important notions related to contained rewriting. These notions, such as CAT, component pattern, will be frequently used throughout this thesis. Finally, we formalize the concepts of ER, CR, MCR and IMCR.

2.2.1 Equivalent Rewriting

Our main focus of the thesis is on contained rewriting. However, before introducing contained rewriting, we would like to start with equivalent rewriting, which is more simple, but shares a similar idea with contained rewriting. The way we find an equivalent rewriting will shed some light on the way we find a contained rewriting.

To find a rewriting for an XPath query using a view, is to find the set of conditions that are not satisfied on the view query, but may be satisfied under the distinguished node of the view query. With the definition of homomorphism, we have: an equivalent rewriting for \( q \) using \( v \) exists, only if there exists a node \( n \) in \( q \) such that two homomorphisms exist between \( q_{\text{up}}(n) \) and \( v_{\text{up}}(d_v) \), and \( d_q \) is not an ancestor of node \( n \). Here, \( q_{\text{up}}(n) \) is the subpattern obtained by removing all descendants of \( n \) in \( q \), \( v_{\text{up}}(d_v) \) is the subpattern obtained by removing all descendants of \( d_v \) in \( v \).
2.2.2 Contained Rewriting

Similar to equivalent rewriting, the solution to find a contained rewriting is to find an embedding from the query to the view, and check whether the unembedded parts are possible to be satisfied under the distinguished node of the view. An embedding is a partial homomorphism. A valid embedding that can produce a contained rewriting is called useful embedding. Embedding and useful embedding are defined below.

**Definition 2.6** Given a query \( q = (N_q, E_q, r_q, d_q, \Sigma_q) \) and a view \( v = (N_v, E_v, r_v, d_v, \Sigma_v) \), an embedding is a mapping \( e : N'_q \rightarrow N'_v \), where \( N'_q \subseteq N_q, N'_v \subseteq N_v \), satisfying:

- **root preserving**: \( e(r_q) = r_v \);
- **label preserving**: \( \forall n \in N'_q, \text{ label}(n) = \text{label}(e(n)) \);
- **structure preserving**: \( \forall (n_1, n_2) \in E_q \text{ and } n_1, n_2 \in N'_q, \text{ if } (n_1, n_2) \text{ is pc-edge, } pc(e(n_1), e(n_2)) \text{ holds in } v; \text{ otherwise } ad(e(n_1), e(n_2)) \text{ holds in } v; \)
- **e is upward closed**: if node \( n \) in \( N_q \) is defined by \( e \) (means \( n \in N'_q \)), all ancestors of \( n \) in \( N_q \) are defined by \( e \). Namely, \( n \in N'_q \Rightarrow \forall n', n' \in N_q \text{ and } ad(n', n) \text{ holds in } q, \text{ we have } n' \in N'_q. \)

Not every node in \( N_q \) needs to be defined by \( e \), and here \( N'_q \) is the set of nodes that are defined by \( e \). An embedding implies that part of the conditions of query \( q \) have been satisfied in the view \( v \), so if the left conditions of \( q \) (unembedded parts) are possible to be satisfied under the distinguished node of the view, we will be able to use the view \( v \) to answer the query \( q \), i.e. we can find a contained rewriting for query \( q \) using view \( v \). Such embedding is called a useful
CHAPTER 2. PRELIMINARIES

 embedding. Before giving the formal definition of useful embedding, we give the definition of anchor node first. Given an embedding \( e \), for each unfully embedded path \( path_i \) of \( q \), the last node on \( path_i \) can be embedded on or above \( d_v \) is called an anchor node of embedding \( e \) with respect to path \( path_i \). An embedding is called a useful embedding, if both of the following conditions hold:

1. every anchor node \( n_a \) satisfying either of (a), (b):
   - (a) \( e(n_a) = d_v \);
   - (b) \( ad(e(n_a), d_v) \) holds in \( v \) and let \( n_b \) be the child node of \( n_a \) on \( path_i \) in \( q \), \( (n_a, n_b) \) is an ad-edge.

2. for the anchor node \( n_d \) on the distinguished path of \( q \), either \( n_d = d_q \) or \( ad(n_d, d_q) \) holds in \( q \).

The idea behind condition (1) is: unfulfilled conditions in \( q \) have the possibility to be satisfied under \( d_v \) in the view, which requires that an anchor node either maps to \( d_v \) (condition (a)) or the anchor node maps to an ancestor of \( d_v \) and the anchor connects its descendant on the corresponding path with “//” (condition (b)). Condition (2) implies: the return node of \( q \) should not be mapped onto a node above \( d_v \).

For ease of understanding, we give an example to illustrate embedding, useful embedding and anchor nodes. In Fig 2.1, \( e_1 \) is an embedding, but not a useful embedding, its anchor node, the \( a \)-node, does not satisfy condition (1)(b). While \( e_2 \) is a useful embedding, both condition (1) and (2) are satisfied by \( e_2 \)’s anchor nodes. To be specific, on path \( path_1 \), the anchor node \( b \)-node maps onto the distinguished node of the view \( v \) (satisfying condition (1)(a)); on path \( path_2 \), the anchor node \( a \)-node maps onto a node above the distinguished node of the view, and \( a \)-node connects its successor \( d \)-node on \( path_2 \) with “//” (satisfying condition
Figure 2.1: An example of useful embedding

(1)(b) ; the anchor node $b$-node on the distinguished path $path_1$ is above the distinguished node in the query $q$ (satisfying condition (2)). As a result, $e_2$ is a useful embedding. The corresponding contained rewriting produced by $e_2$ is also given on the rightmost side of Figure 2.1. In the CR, $b[//d]/c$ is called a compensation pattern to the view, which is often referred as clip-away tree (CAT) in the literature [LWZ06].

We define another concept, component pattern. A component pattern may be a predicate component pattern or a distinguished component pattern. In the CR in Figure 2.1, we call $b//d$ a predicate component pattern, $b/c$ the distinguished component pattern. Note that, a component pattern may be a tree pattern, not confined to be a path pattern. We may use lowercase $p$ to denote a component pattern, and we may denote a CAT $P$ as $\{p_1, \ldots, p_n\}$, since a CAT usually contains several component patterns fusing at the root.
In addition, not every path should have an anchor node, e.g. one path may be fully embedded, and thus does not have an anchor node. In the definition of useful embedding, we have made a modification on the concept of “anchor node” compared to the previous works [LWZ06]. We believe that, using this modification, the problem can become clearer.

2.2.3 Formulations of ER, CR, MCR and IMCR

We now formalize different types of rewritings equivalent rewriting (ER), contained rewriting (CR), maximal contained rewriting (MCR) and irredundant maximal contained rewriting (IMCR). Let \( t \) be an XML tree, \( q \) be an XPath query. Given an XPath view \( v \), a rewriting \( q_1 \) of \( q \) using \( v \) is a pattern composed by the view \( v \) and a compensation pattern (CAT). The compensation pattern (CAT) captures the unfulfilled condition of \( q \) on \( v \):

- \( q_1 \) is an equivalent rewriting (ER) of \( q \), if \( q_1(t) = q(t) \).
- \( q_1 \) is a contained rewriting (CR) of \( q \), if \( q_1(t) \subseteq q(t) \).
- Query set \( Q \) is a contained rewriting set of \( q \), if \( \forall q_i \in Q, q_i(t) \subseteq q(t) \).
  \[ Q(t) = \bigcup_{q_i \in Q} q_i(t) \]
- Query set \( Q \) is a maximal contained rewriting (MCR) of \( q \), when, for any other contained rewriting set \( Q' \), \( \bigcup_{q_j \in Q'} q_j(t) \subseteq \bigcup_{q_i \in Q} q_i(t) \).
- Query set \( Q \) is an irredundant maximal contained rewriting (IMCR) of \( q \), if \( \neg \exists Q' \subset Q, \bigcup_{q_j \in Q'} q_j(t) = \bigcup_{q_i \in Q} q_i(t) \).
2.2.4 Trimming the View

For a given tree pattern view \( v \), according to the previous analysis in Section 2.2.1 and 2.2.2, it is not different to see that query conditions under the distinguished node \( d_v \) of the view \( v \) does not affect the existence of equivalent rewritings or contained rewritings. Let \( v_{up}(n) \) denote the subpattern obtained by removing all descendants of node \( n \) in \( v \). We can safely use \( v_{up}(n) \) to find compensating CATs for users’ queries. To avoid using clumsy notations in the thesis, we always regard a view as having been trimmed at the distinguished node, so we always use \( v \) to refer to \( v_{up}(n) \). Fig. 2.2 shows an example.

```
Figure 2.2: An example of trimming the view
```
Chapter 3

Related works

Answering queries using views has been extensively studies for a long time. Halevy did a survey [Hal01] on this problem over relational database and pointed out its wide impact on a number of data management applications, such as query optimization, data integration, data warehouse design and semantic query caching. Efficient algorithms were developed as well, eg. MiniCon [PH01], bucket [SDJL96], inverse-rules [Qia96, DG97], to tackle the problem in relational context.

It then immediately started to draw the attention of researchers on XML data. Since XPath serves as the core sub-language of the major XML query languages such as XQuery and XSLT. Fruitful achievements have been made on rewriting XPath Queries with XPath Views [BÖB+04, MS05, XÖ05, LWZ06, ABMP07, ACG+09]. Two types of rewritings for XPath queries have been studied in the literature. One is equivalent rewriting [XÖ05]: given a materialized view $V$ of a database $D$, an equivalent rewriting $Q'$ of a query $Q$, runs over the view $V$ producing the same set of answers as evaluating $Q$ over $D$, i.e. $Q'(V) = Q(D)$. Here, we use $Q(V)$ and $Q(D)$ to denote the returned query results by evaluating $Q$ on $V$ and $D$ respectively. However, an equivalent rewriting may not always
exist, and moreover part of answers covered by the view are still valuable. In the data integration scenario, data sources are limited to cover the domain. It is very common that we cannot find an equivalent rewriting for a query. Therefore contained rewriting [LWZ06] is introduced and can be described as follows: given a view $V$ on a database $D$, a contained rewriting $Q'$ of a query $Q$, runs over $D$ producing a subset of answers as evaluating $Q$ over $D$, i.e. $Q'(D) \subseteq Q(D)$.

The works [LWZ06, ABMP07] focus on contained rewritings. They also consider schema [LWZ06] or data summary [ABMP07] constraints, but they did not well address the redundancy of contained rewritings. Neither the evaluation of the contained rewritings on materialized views is studied in the previous works. Our work fills in these blanks.

In this section, we systematically review the state-of-the-art works. We neither classify the works according to rewriting types (equivalent or contained), nor classify the works according to query types (XPath or XQuery). Instead, we categorize the works according to the problems they attempt to solve. Therefore, some work may be introduced in different sections, because it addressed different problems. That means, a work may be mentioned in different sections, but of course on different aspects. In the following, we will firstly introduce the semantics of views. The thesis will focus on two popular semantics, and ignore the less used one. Afterwards, we discuss how to select views to materialize, how to store the materialized views, how to rewrite a query using a single view or multiple views in the sequel. In the end, we introduce another topic, XPath containment, which is considered to be closely related to rewriting queries using views.
3.1 View Semantics

In a broad sense, a view of a database is a way to express the whole or (most of the time) part of the database. Views are of many types in the relational context: (a) the most used type is predefined queries by the system manager or historical queries from users. The main aim of using these views is to speed up query evaluation, because part of the computation for the later posed queries have been completed when preparing the views. We can call this type, \textit{query as view}. (b) another type of view is index. Some work [Val87] regarded precomputed indexes as views, so that the precomputed views can be properly chosen to answer new queries. The rationale is to accelerate query evaluation as well. We call this type, \textit{index as view}. (c) the last type arises in the application of database design [TSI96], which we call \textit{physical storage as view}. The idea is to provide a mechanism to separate the physical storage of the data from the logical schema of the data. As a result, we are able to try different storage schemata and pick the most suitable one.

In the XML context, views can be divided into the above three categories as well. For \textit{query as view}, query caching systems [CR02, MS05, XÖ05] aim to store previous queries as views in order to answer new queries. For \textit{index as view}, B+-tree [CVZ+02], XR-tree [JLO03] indexes can be built to accelerate the evaluation of structural joins [AKJP+02]. These indexes can be considered as precomputed views in XML databases. There are also different storage schemes for XML data, which can be regarded as \textit{physical storage as view}. In the thesis, we limit our attention on the first category, \textit{query as view}. We will also mention some recent progress on materializing views as index in Section 3.3. For the most parts of the thesis, we will regard a view as a query, and a materialized view as a materialization of the view query.
CHAPTER 3. RELATED WORKS

3.2 Selecting Views to Materialize

Another related aspect is how to select representative views to materialize or manually generate such representative views. It is also a classic problem in relational database context [CHS01]. Materialized views can be regarded as cached results for a large amount of computation. If the cached results can be frequently reused, we can avoid repeated accesses to base data and also repeated computation for the same set of data. While, on the other hand, memory and disk storage space is limited. Consequently, it needs a wise choice to select which views to materialize.

In the work [MS05], the authors model the problem as a variant of the set cover problem: given a workload $S$ made up of a number of queries, each view can answer a subset of $S$, we want to pick subsets which together cover a maximal subset of $S$, under the constraint that we can pick at most $k$ subsets. A greedy approximation algorithm is used to choose a set of view. There are also other heuristics to improve the view selection, such as (1) putting a bound on the maximum number of values to store for a parameter; (2) choosing the views with looser predicate values to cover more strict conditions; (3) truncating a view to generalize a pattern so that it can contain more future queries, but still remain selective. The weak point of this work is that the workload considered in the paper follows a number of predefined templates. It may not be suitable to the applications where queries cannot be classified into predefined categories.

Tang et al. [TYT'09] studied the view selection problem as an optimization problem. Let $S$ be the query workload and each $S_i \in S$ is associated with a non-negative weight $w_i$, $X$ be an XML database, $B$ be the available storage space, and $COST()$ be an estimation function for query processing. The problem is to find a set of views $V$ whose total size is at most $B$ and minimizes:
\[ \text{COST}(X, V, S) = \sum_{s_i \in S} \text{COST}(X, V, s_i) \times w_i \]

A new way to generate a small number of candidate views while ensuring optimality is proposed. The set of candidate views is organized in a graph structure called VCUBE. Based on VCUBE, two heuristic algorithms for view selection are developed: Space-Optimized algorithm aims to materialize the smallest XML fragments to answer a query workload; Space-Time algorithm trades off the space for time.

Yang et al. [YLH03] studied how to mine frequent tree patterns from a large query set. There representative tree patterns can be materialized to answer new queries. They developed theorems to prove that only a small subset of the generated candidate patterns needs to undergo costly tree containment tests, and an efficient algorithm called FastXMiner is proposed, to discover frequent XML query patterns, and demonstrate how these query patterns can be incorporated into a query caching system for XML data. Their experiment results show that FastXMiner is efficient and scalable, and that utilizing the frequent query patterns in caching strategy increases the cost-saving ratio and reduces the average query response time.

In the work [WYPL09], a minimal common container of tree patterns is proposed. For queries in \( XP^{(//)} \) and \( XP^{(//,\star)} \) (here, \( XP^{(//,\star)} \) is a subset of \( XP^{(//,\star)} \), where *-nodes do not incident on // -edges.), given a set of queries as \( q_1, \ldots, q_n \), there exists a query \( q \) such that (1) each \( q_i \subseteq q \ (i \in [1, n]) \), and (2) for any \( q' \) satisfying that each \( q_i \subseteq q' \ (i \in [1, n]) \), we have \( q \subseteq q' \). In other words, \( q \) is the minimal common container of \( q_1, \ldots, q_n \). The way to construct a minimal common container for a set of queries is also developed in [WYPL09]. The idea of minimal common container may be helpful to select representative views...
CHAPTER 3. RELATED WORKS

to materialize. For example, if we choose to materialize the minimal common container, we will be able to answer more queries. Since the common container is minimal, the materialized query is not too general.

3.3 Materializing the Views

Views can be virtual or materialized. In the areas of privacy-preserving publishing, access control management, it is proper to provide a virtual view schema to users so that we can hide sensitive data from being abused. Users issue queries on the virtual view, and the queries will be translated and be executed on the base data. Of course, we can also dump data into the virtual view and make it materialized. The view is materialized according to a mapping from the base schema to the virtual view schema, and underlying data can be put into the view according to the mapping. While in the area of query optimization, data warehouse, query caching, it is compelling to materialize a set of views in order to speed up the computation of frequently issued queries. The set of views can be regarded as precomputed queries. One of the major problems is how to store the set of views, i.e. how to store the query results for each view query. In this section, we review some storage implementation for the materialized query views.

3.3.1 Query as View

A majority of works [BÖB+04, MS05, XÖ05, TYÖ+08] fall in the first category: the materialized view is modeled as a set of subtrees rooted at the view query output nodes. We can store the set of subtrees separately from the base XML tree or store the references to the base XML tree according to different application scenario. In a distributed scenario, like query caching, view subtrees need to be stored at local sites separately from the base XML tree, because the base XML
document resides at the server side, and cannot be easily shipped to the client sites. In order to speed up query processing, it is reasonable to store some popular fragments of the base XML tree at the client side. In a centralized scenario, on the contrary, there is no need to materialize the same part of the base XML tree for multiple times, because to store the same part of the base XML tree redundantly will take up extra space cost, yet does not bring in evaluation benefit. Therefore, it is more appropriate to store node references as materialized views.

In the work [BÖB+04], views contain copies of XML fragments or node references. A framework for exploiting materialized views to answer XML queries was proposed. Data values and full paths of the view answer nodes are also materialized. Typed values are helpful to evaluate comparison predicates either in where clause of a FLWOR expression, or in a predicate of an XPath step expression. Full paths can be useful when the defining XPath expression of the view contains wildcard or descendant axes.

### 3.3.2 Index as View

Another stream of works [WTW09, CC10] do not materialize view query results directly. They actually materialize the indexes from which view query results can be computed after a number of joins. The advantages of these works are that (1) they have materialized the nodes beyond the view output subtrees, which will increase the chance to answer a query by using multiple views; (2) they do not need to store redundant information, because redundant results are stored implicitly in the indexes, rather than explicitly materialized. The results can be simply computed by joins. The disadvantages are that (1) even an identical query to a predefined view needs to be recomputed because the views are materialized as indexes, not answers; (2) their materialized views must be used in a centralized environment. The works [WTW09, CC10] can be regarded as materializing index
as view. They share a similar initiative, but have technical differences. We now introduce them in individual.

Wu et al. [WTW09] assume the base XML document is large and has been encoded by an encoding scheme, such as the popular 3-tuple scheme following the pioneer work [LM01]. Namely, each element, attribute or text value is a tree node and is associated with a 3-tuple \(<\text{start}, \text{end}, \text{level}\>\), where “start” and “end” are determined by the position of the start tag and end tag of the node in a preorder traversal of the XML document tree, and “level” is the length from the root node to the current node. This encoding is well-known for its convenience to determine ancestor-descendant and parent-child relationship. If a node \(n_1\) is an ancestor of node \(n_2\), then \(n_1,\text{start} < n_2,\text{start} \) and \(n_1,\text{end} > n_2,\text{end} \). Furthermore, if \(n_1\) is the parent of \(n_2\), we have \(n_1,\text{level} = n_2,\text{level} - 1 \) besides \(n_1,\text{start} < n_2,\text{start} \) and \(n_1,\text{end} > n_2,\text{end} \). For each label in the XML tree, an inverted list of the nodes with this label is stored as index. To be strict, each inverted list stores a list of 3-tuples ordered by their \(\text{start}\) field. The view queries considered in [WTW09] are twig queries. The difference from XPath queries is that, in their work, the view output node is not differentiated. All query nodes are output nodes. The materialized view of a query is a subset of the entire inverted lists containing only the nodes that contribute to an answer to the view query. As a result, the sublists can encode an exponential number of view query answers in linear space.

View usability and view manipulation are changed corresponding to the new index as view semantics. For usability, (1) a view \(v\) can be used to answer a query \(q\) if and only if there exists a homomorphism from \(v\) to \(q\); (2) a view \(v\) can be used to completely answer a query \(q\) if and only if there exist a few homomorphisms from \(v\) to \(q\) such that every node in \(q\) is the image of a node in \(v\) under some homomorphism.

Chen et al. [CC10] propose to add links into the materialized lists. In the
linked-element scheme, each materialized view is conceptually stored as a directed acyclic graph (DAG). A materialized view also contains a number of inverted lists, and each list contains a set of nodes with the same tagname. Each node in an inverted list has three pointers, (1) a following pointer, which links a node \( n \) to a following node \( n' \). Here, \( n' \) has the same tagname with \( n \) and the smallest \textit{start} value after \( n \); (2) a descendant pointer, which links a node \( n \) to a first descendant node \( n'' \) in the inverted list; (if \( n'' \) does not exist, the pointer is a null pointer.) (3) a few child pointers, which connect \( n \) to a number of nodes in other inverted lists, recording the direct descendants of node \( n \). A partial linked-element scheme is also proposed. The idea is to reduce the index space without sacrificing the evaluation performance.

### 3.4 Rewriting Queries Using Views

Rewriting XML queries using views have been studied for a decade. Researchers first focused on using a single view to answer a new query, and then extended the study on using multiple views. We introduce the progress for both trends in this section.

#### 3.4.1 Single View

Rewriting XML queries using views is firstly studied on using a single view. In [BÖB+04], Balmin et al. consider in server side using materialized XPath views, which can include XML fragments, data values, full paths, or node references, to speed up the processing of XPath queries. They use heuristics to decide the existence of rewritings, and also another heuristic to minimize compensation queries. Mandhani and Suciu [MS05] propose to use string match to find promising views to answer a query. They also investigate how to select the materialized views,
which has been introduced in Section 3.2. In [CR02], Chen and Rundensteiner consider in client side using cached results of previous XQuery queries to answer new queries. All of the above works use heuristics to decide rewritings. No theoretic analysis on this problem has been addressed in the early works. However, since a lot of theoretical works have been done for containment and minimization problems of XPath queries, it is natural to think of the theoretical result on the problem of rewriting queries using materialized XPath views without schema.

Xu and Ozsoyoglu [XO05] fill the gap by providing a theoretical study on equivalent rewriting using a view based on query containment [MS04] and query minimization [AYCLS02]. It has shown that finding an equivalent rewriting for queries in $\text{XP}^{[*]}$, including branches, wild cards and descendant axes, is coNP-hard. And for three subclasses (combining any two of the three features), $\text{XP}^{[*]}$, $\text{XP}^{[*]/}$ and $\text{XP}^{[*]/}$, the problem is in PTIME. These conclusions are based on the previous study of XPath containment in [MS04]. Afrati et al. [ACG+09] have extended Xu and Ozsoyoglu’s result in fragment $\text{XP}^{[*]}$. They have discovered a coNP-complete upper bound for some sub-fragments of $\text{XP}^{[*]/}$, and also show to find an equivalent rewriting for $\text{XP}^{[*]/}$ is decidable in the general case.

In fact, the problem of answering queries using views has been rigorously studied for the class of regular path queries [CGLV00, GT03] and in semi-structured graph databases [PV99]. Deutsch and Tannen [DT03] have studied the problem of query reformulation in the context of relational to XML publishing. They reduce the problem for XQuery to relational query reformulation under constraints. They show that their extended chase and back chase procedure is complete for this problem for a subclass of XQueries. Tang and Zhou [TZ05] conducted a formal analysis of answering query using views for XPath, but they adopt a tuples-of-nodes semantics, which is at variance with the standard. Some
later works are [ODPC06, ABMP07, FGJK07]. Onose et al. [ODPC06] have investigated the equivalent rewriting of XQuery queries using XQuery views. While XQuery queries are more expressive, the shortcomings of using them as views are also noted in [ODPC06]. The works [ABMP07, FGJK07] are different from [ODPC06] for considering constraints when computing query rewritings. [FGJK07] considers DTD constraints and treats XML queries as regular XPath queries. While in [ABMP07], constraints are modeled as structure summaries.

As we explained in the introduction, equivalent rewriting may not always exist, and part of answers covered by the view are still valuable. Lakshmanan et al. [LWZ06] studied the contained rewriting problem, especially on finding the maximal contained rewriting (MCR). They also pointed out the redundant contained rewriting problem, though their claimed sufficient and necessary condition to check irredundant contained rewriting is shown to be invalid in this thesis. We have given the correct solution to find all the irredundant contained rewritings, and shown its correctness with proofs. In addition, we have also investigated how to evaluate the irredundant contained rewritings on materialized views. Irredundant rewriting is also mentioned in [WWL09], but in work [WWL09], the authors mainly focus on irredundant embedding rather than irredundant rewriting. Wang et al. [WYL08] pointed out an XPath subset in which contained rewritings can be found in PTIME.

3.4.2 Multiple Views

Later, researchers realize that a query may not be answerable using a single view, but may be answerable using a set of views combined together. There are several ways to combine multiple views: view union, view join and view intersection.

Among these alternatives, using view union can be regarded as using each view to answer the query and uniting the result for each one to get the final
CHAPTER 3. RELATED WORKS

result. Gao et al. [GWY07] proposed to use MQTree to combine similar views together so that, when rewritings are generated for a query against a union of views, shared paths are matched only once in the MQTree in stead of being matched for several times, each time for a single view. The advantage is that if materialized views have more overlapping parts, it is more efficient to use MQTree to find the rewritings. While the weak point is, if a number of views have a typical overlapping skeleton, we may choose to intentionally materialize the skeleton rather than a set of similar views. The work [TF04] studied the problem of equivalently answering a query using a union of views, i.e. given \( q \) and \( v_1, \ldots, v_n \), whether there are \( q_1, \ldots, q_n \) such that \( q_1 \circ v_1 \cup \cdots \cup q_n \circ v_n = q \). And it tries find such a set of queries \( Q = \{ q_1, \ldots, q_n \} \) such that the total size of answers given by the queries in \( Q \) is minimal.

Another stream of works adopt view joins [BÖB+04, TYÖ+08, WTW09, CC10] to combine multiple views together to answer a query. In all of the above works, extra knowledge about the views, such as full path information [BÖB+04], inverted lists for nodes in the view [TYÖ+08, WTW09, CC10], needs to be stored. In [BÖB+04], each useful view is identified individually, and then is combined with other views to answer the queries. In [TYÖ+08], the authors attempt to find a minimal set of views to answer the query, and show the problem is NP-hard. They propose a heuristic algorithm to find a reasonable small view set. The answerability of a query using a set of views is defined by \textit{leaf-cover}. The intuition behind leaf-cover is that, if a leaf node set of \( q \) can be covered by a view \( v \), all ancestors of the leaf node set can be identified from \( v \) by either examining the materialized fragments of \( v \) or verifying the encodings. If a view set \( V \) covers all leaf nodes in \( q \), then all predicates for \( q \) can be identified, and \( q \) is answerable using the set of views \( V \). The materialized fragments of different views can be associated by joins. The works [WTW09, CC10] materialize views differently. They store the
materialized views as a set of inverted list with or without extra pointers. (There are extra pointers in [CC10], but not in [WTW09].) The work [WTW09] has encoded multiple views together, without differentiating them. As a result, multiple views are actually stored together. For a given query, it always tries to use a set of views to answer the query. The work [CC10] proposes a similar solution using structural joins of inverted lists like [TYÖ+08, WTW09]. The differences are: (1) the work [CC10] also materialize nodes, which are in the subtree under the output node of a view, into inverted lists, while the work [TYÖ+08] only materializes the inverted lists for nodes not in the subtree rooted at the output node of a view; (2) the work [CC10] stores the inverted lists for each view separately, while the work [WTW09] stores the inverted lists for all the views together.

Intersections of multiple views are also used to answer queries. Cautis et al. [CDO08] investigate the complexity of intersection-aware rewriting problem. They identify tight restrictions under which sound and complete rewriting can be performed in polynomial time, and beyond the restrictions the problem is coNP-hard. The restrictions are practically interesting as they permit expressive queries and views with descendant navigation and path filter predicates. As a by-product, they show that the containment for wildcard-free patterns allowing pattern intersections is intractable, whereas XPath containment for the subset $XP^{\{/\},\{/\}}$ is PTIME if we disallow pattern intersections in the containment check. Wang et al. [WY08] also studied how to use intersections of views to answer a query. They propose to reformulate the intersection of views into a union of views, and develop several algorithms to find equivalent rewritings and maximal contained rewritings using intersections of views.

There are others work on rewriting queries by exploiting different views. In the ULoad project [ABMP07], a structure summary is materialized as constraints when rewriting queries. The constraints include Dataguides summary [GW97]
and integrity constraints. Another related work [CDOV09] propose to compactly specify a set of queries using Query Set Specifications [PDP03], a formalism close to context-free grammars. In their work, a query $q$ is called *expressible* by the specification $P$ if it is equivalent to some expansion $P$, and query $q$ is *supported* by $P$ if it has an equivalent rewriting using some finite set of $P$’s expansions. The complexity of expressibility and support is studied, and a number of classes of XPath queries for which there are PTIME algorithms are identified.

Wang et al. [WYL07, WYL09] examined the relationship between an equivalent rewriting and a union of contained rewritings. They showed that for queries and views in $X P^{(//, [])}$, if there is no equivalent rewriting in the form of a single tree pattern, then there is no equivalent rewriting in the form of a union of tree patterns. This conclusion further holds in the presence of non-recursive, non-disjunctive DTDs. But the result no longer holds for XPath subset $X P^{(//, [])}$ or if there is a recursive DTD.

### 3.5 XPath Containment

XPath containment is related to XPath query rewriting. In the rewriting process, one sub-step is to find which conditions can be satisfied on the view definition and which conditions need to be further checked in the materialized view. The work [XÖ05] revealed the relationship.

Containment for a fragment of XPath queries $X P^{([], *, //)}$, including branches, wild cards and descendant axes, is shown to be coNP-complete in [MS04], though for three subclasses (combining any two of the three features), $X P^{([], *)}$, $X P^{([], //)}$ and $X P^{(*, //)}$, the containment problem is in PTIME. The work [MS02] also proposed a PTIME-efficient but incomplete algorithm to determine containment in $X P^{([], *, //)}$. And this homomorphism-based algorithm was thereafter utilized by
the works [BÖB⁺04, XÖ05] to evaluate equivalent rewritings of XPath queries using materialized views. Containment for XPath queries under DTD constraints, with disjunctions and variables can be found in [Woo03, NS03]. A high-leveled summarization can be found in [Sch04a]. Recently, the complexity of query containment in expressive fragments of XPath 2.0 has been discussed [tCL07]. Dong et al. [DHT04] studied the theoretical result for a set of nested XML queries. Wang et al. [WYLZ08, WY10] studied how to use non-disjunctive non-recursive [WYLZ08] or recursive [WY10] DTD to chase a tree pattern so that containment test can be done easily after the chasing.

Tree pattern minimization is also considered as a related component, but independent aspect, because a contained rewriting can be regarded as a merged pattern of a compensation pattern with a view definition. To express the rewriting compactly, a minimization step is needed. Tree pattern minimization has been extensively studied with or without schema [Woo01, AYCLS02, FFM08, CC08]. Similarly, tree pattern satisfaction [LRWZ04, BFG08] can used as a preprocessing step in query processing. Queries can be checked first to identified whether the query is valid for a given schema.
Chapter 4

Finding the IMCR

In this chapter, we will focus on finding all irredundant contained rewritings for an XPath query using a view. We will first take a look at the challenges in Section 4.1, and address how to tackle the challenges in Section 4.2 and Section 4.3. To introduce more, we show a CR independence property in Section 4.2 to avoid checking containment between a CR and a union of CRs. In Section 4.3, we develop three algorithms including several heuristics to serve as a sufficient condition to detect redundant CRs. Experiments and conclusions are given in Section 4.4 and Section 4.5.

4.1 Motivation

Given a query $q$ and a view $v$, our target is to find the irredundant maximal contained rewriting (IMCR) for $q$ using $v$. The IMCR is expressed as a set of irredundant CRs. A general method of finding these irredundant CRs consists of two steps: (1) generate all the CRs including both redundant CRs and irredundant CRs to comprise a candidate CR set, (2) and then conduct containment checks to eliminate the redundant ones. In the first step, the goal can be achieved
by finding all the useful embeddings, because each useful embedding can produce a CR, and also each CR must be produced by (possibly more than) one useful embedding(s). In the second step, given the candidate CR set, we need to check whether a CR can be contained by another CR or by a union of some other CRs. If so, the redundant CR will be removed from the candidate CR set. Finally we obtain the IMCR after removing all the redundant CRs.

We now have a look at the challenges in the two-phase process. Taking into account that the number of CRs may be exponential, step (2) is very expensive, because, (a) one CR may be contained in a union of other CRs, and (b) even though we ignore the above case, we still need to check query containment for an exponential number of CR pairs. Therefore, in step (1), we should try to minimize the size of the candidate CR set. While for the second step, as is mentioned above, there are a potentially large number of containment checks we need to conduct, because a CR may need to be compared with any combination of other CRs as long as it has not been found to be redundant. It is desirable if we can find any property to reduce the containment checks. In Section 4.2, we will show a property that can avoid comparing containment between one CR and a union of CRs. And we will show how to minimize the candidate CR set in Section 4.3.

4.2 Avoid Containment Check for a Union of CRs

In the previous study [LWZ06], irredundant CR is defined as “not contained in any other CR”. This definition implies we only need to do pairwise checks between the CRs. This has made the computation simple. However, if a union
CHAPTER 4. FINDING THE IMCR

of CRs can contain some CR, according to common sense, the contained CR should be regarded as redundant as well. The question is: given \( CR_1, CR_2 \) and \( CR_3 \), is it possible that \( CR_1 \not\subseteq CR_2 \) and \( CR_1 \not\subseteq CR_3 \), but \( CR_1 \subseteq CR_2 \lor CR_3 \)? Fortunately, within XPath query subset \( XP^{\text{/}/[]} \), we found the answer is no, which means if a CR is contained in a union of others, it must be contained in one of them. As a result, the previous definition in [LWZ06] happens to be equivalent to our definition in Section 2.2.3. We now highlight the property as a theorem and prove it afterwards.

**Theorem 4.1** For an XPath pattern \( p \) and a union of XPath patterns \( Q = \{q_1, q_2, \cdots, q_n\} \) in \( XP^{\text{/}/[]} \), we have: \( p \subseteq Q \) if and only if there exists \( q_i \in Q \) such that \( p \subseteq q_i \).

**Proof sketch:** The sufficient condition is obvious. Now we will prove the necessary condition by proving its contrapositive statement, i.e. to show that if \( p \) is not contained in any \( q_i \in Q \), then \( p \) cannot be contained in \( Q \). For simplicity, we consider each pattern as a boolean pattern. It has been shown in [MS04], containment relationship for XPath patterns remains the same for their boolean pattern counterparts. For two boolean patterns \( p \) and \( q \) in \( XP^{\text{/}/[]} \), \( p \subseteq q \) if and only if there exists a homomorphism from \( N_q \) to \( N_p \). In other words, if \( p \not\subseteq q \), there must exist a node \( n_i \) in \( N_q \), such that we cannot find any homomorphism \( h \) that has a corresponding node \( h(n_i) \) in \( N_p \), satisfying label preserving and structure preserving conditions w.r.t. nodes \( n_i \) and \( h(n_i) \). We call such node \( n_i \) a private node of \( q \) against \( p \). We also name, on some path in \( q \) (from root to leaf), the first private node as a transitional node.

To prove the contrapositive statement of the necessary condition in Theorem 4.1, given \( \forall q_i \in Q, p \not\subseteq q_i \), we could construct a tree \( t \), such that \( p(t) \) holds while \( q_i(t) \) is false. And hence \( p(t) \) does not imply \( Q(t) = q_1(t) \lor q_2(t) \lor \cdots \lor q_n(t) \),
CHAPTER 4. FINDING THE IMCR

namely $p$ is not contained in $Q$. The tree $t$ can be constructed as follows: replace each ad-edge in $p$ with two pc-edges and an additional distinct label $z$. For instance, $a//b$ can be transformed into $a/z/b$. Here label $z$ does not appear in any $\Sigma_q_i$ (i.e. $z \in \Sigma - \bigcup_{i=1}^{n} \Sigma_q_i$), where $\Sigma_q_i$ is the alphabet of $q_i$. Since $\Sigma$ is infinite (when there is no schema available) and $\Sigma_q_i$ is finite (because the number of labels in a query is limited), this transformation is always possible. After the transformation, it is straightforward that the result tree $t$ conforms to pattern $p$, and thus $p(t)$ is true. However, for any $q_i$, we can show $q_i(t)$ is false. The reason is: since $p \not\subseteq q_i$, there must be some transitional node $n_i$ in $q_i$, such that for the transitional node, we cannot find a corresponding node $f(n_i)$ in $t$ defined by any mapping $f$ from $q_i$ to $t$. Otherwise, if such mapping $f$ existed, we could obtain a twin homomorphism $h$ from $q_i$ to $p$ based on $f$. Here the twin homomorphism $h$ would have the same mapping function as $f$, because, in $f$, no nodes in $q_i$ can be mapped onto $z$-nodes (nodes with distinct label $z$) in $t$. Therefore, a corresponding node $h(n_i)$ in $p$ would exist for the homomorphism. This result contradicts with the assumption that $n_i$ is a transitional node. Recall that, a transitional node in $q_i$ could not map onto any node in $p$ by any homomorphism, as a result, $\forall q_i \in Q, q_i(t)$ is false, i.e. $Q(t) = q_1(t) \lor q_2(t) \lor \cdots \lor q_n(t)$ is false. In addition, $p(t)$ is true, hence $p(t) \not\rightarrow Q(t)$. The contrapositive statement of the necessary condition is proved.

Theorem 4.1 holds for all XPath patterns in $XP^{//}$ and $XP^{//,[]}$. As the CRs are XPath patterns generated by some special operations, Theorem 4.1 holds for the candidate CR set. As a result, we are able to obtain the IMCR by at least performing a pairwise comparison for the CRs from the candidate CR set. The computation is not that expensive as is assumed if the candidate CR set is small. We now introduce how to reduce the candidate CR set.
4.3 Minimizie the Candidate CR Set

In this section, we first introduce how to represent a CR and show its advantage. Based on the representation, we develop several algorithms to eliminate part of redundant CRs according to a sufficient condition.

### 4.3.1 Represent a CR

Some redundant CRs are introduced into the candidate CR set, because different useful embeddings can produce the same contained rewriting. Fig. 4.1 gives an example. The $b$-node in query $q$ can be embedded onto either the left $b$-node ($e_1$) or the right $b$-node ($e_2$) in view $v$. The resulting CRs ($CR_1$ and $CR_2$) of two different embeddings are the same. Here, $b$-node means a node labeled $b$.

![Figure 4.1: Different useful embeddings producing the same CR](image)

To uniquely identify a CR, we represent a CR with a set of nodes in query $q$. 
We call these nodes *link nodes*, and give the definition of a link node as follows: let $n_a$ be an anchor node on a path of query $q$ w.r.t. some useful embedding $e$, the child node\(^1\) of $n_a$ on the path is a link node w.r.t. embedding $e$. In Fig. 4.1, as to embedding $e_1$, $path_1$ does not have an anchor node, while $path_2$ has an anchor node $a$, the link node on $path_2$ w.r.t. $e_1$ is the node $d$. The link node set of $e_1$ is $\{d\}$. Similarly for embedding $e_2$, the link node set is $\{d\}$ as well. It is straightforward that, given a view and a set of link nodes, there is a unique way to build the CR by connecting each subtree rooted at a link node in $q$ to the distinguished node of view $v$ with an edge, whose type is the same as the one with which the link node connects its parent node (i.e. the anchor node) in query $q$. Like anchor nodes, a set of link nodes corresponds to some useful embedding. However, since different embeddings may produce the same set of link nodes, resulting in the same CR. We can use a set of link nodes to represent a CR no matter which embedding produces the CR.

Here, we represent a CR with link nodes rather than anchor nodes, because it is more convenient to construct a CR with link nodes. Fig. 4.2 shows an example. To build the CR with link node set $\{c, d\}$, we could simply follow the process mentioned above, without knowing which path the $c$-node (or $d$-node) is on. On the contrary, if we choose to use anchor node set, we should record the information that the $a$-node is the anchor node of $path_2$, but not of $path_1$, and select a corresponding path (or set of paths) under an anchor node when building the CR. Obviously, the former representation is preferred.

To find the link node set of a useful embedding is not difficult. An existing algorithm can serve the goal. Readers can refer to the algorithm in Figure 6 in [LWZ06]. We modify the algorithm in the following ways, either to suit our needs or for the sake of improving it: (1) We do not allow an anchor node to map

\(^1\)The child node must exist according to the anchor node definition.
4.3.2 A Sufficient Condition to Eliminate Redundant CRs

To minimize the candidate CR set, we can eliminate part of redundant CRs based on a theoretical property. To understand the property, let’s see some cases of producing a redundant CR.

Consider an example in Fig. 4.3, $CR_1$ is redundant and is contained in $CR_2$, onto a node below $d_v$, therefore the SN rule in step 3 of the original algorithm needs to be modified with our anchor node definition. Note that, this modification will not degrade the algorithm’s performance, but will lead to producing link node set from an embedding easier. (2) The down part under $d_q$ does not need to be considered in the algorithm, because no nodes under $d_q$ can be an anchor node, since $d_q$ can never map onto a node above $d_v$. We can trim the down part of $d_q$ before the algorithm starts. When the algorithm finishes, useful embeddings are encoded by the labeling. A node with entry $(i : L = \phi)$ is a link node.
because component pattern $p_1$ of $CR_1$ is contained in component pattern $p_2$ of $CR_2$. Compared to embedding $e_2$, $e_1$ does not embed as many nodes as possible, this leads to component pattern containment between $CR_1$ and $CR_2$. One might conceive incomplete embedding (not embedding as many nodes as possible) may be one of the reasons of producing redundant CRs, however, it is indeed component pattern containment introduced by incomplete embedding that make one pattern contained in another. For instance, $CR_2$ produced by $e_2$ is not contained in $CR_3$, although $e_3$ tries to embed more nodes than $e_2$.

One may think that a remedy is to develop some condition-checks to recognize a set of embeddings, which are the only embeddings that can produce irredundant CRs, such as Lemma 1 in [LWZ06]. However, we find this family of solutions is unlikely to fulfill the goal, because component pattern containment may take place between different paths, and therefore we cannot recognize an irredundant CR without examining its component patterns. Fig. 4.4 gives an example. $CR_1$ and $CR_2$ are two CRs for rewriting $q$ using $v$. Here, component patterns $p_1$ and

![Figure 4.3: Component pattern containment on the same path](image-url)
CHAPTER 4. FINDING THE IMCR

$p_2$ are on different paths, and $p_2$ is contained in $p_1$ (explained later in the next paragraph). To show $CR_2 \subseteq CR_1$, we can simply minimize $CR_1$ into $CR_3$. Obviously, we will have $CR_2 \subseteq CR_3 = CR_1$. $CR_2$ cannot be found to be redundant without comparing $p_1$ and $p_2$.

To check containment between a distinguished component pattern and a predicate component pattern, we can change the distinguished component pattern into a boolean pattern by adding a node with a special tag as a child of its distinguished node and turn the distinguished node into an ordinary node, and then compare the result boolean pattern with predicate component pattern. Note that predicate component pattern itself is a boolean pattern. Therefore, a distinguished component pattern may be contained by other predicate component patterns, but cannot contain other predicate component patterns, because the specially added node can never map onto some node in predicate component patterns.

Figure 4.4: Component pattern containment on different paths

In the above examples in Fig. 4.3 and 4.4, one common observation is that
CR containment can be attributed to CAT containment. In Fig. 4.3, each CAT is a single component pattern, and the result is obvious. While in Fig. 4.4, each CAT consists of two component patterns. It is not obvious to find the CAT containment without minimizing $CR_1$, although apparently, the CAT containment is not affected by the minimization step. This observation is summarized as Lemma 4.1, which has been found in [WWL09] recently.

**Lemma 4.1** Let $CR_1, CR_2$ be two contained rewritings of query $q$ using view $v$, $P$ and $Q$ are the CAT of $CR_1$ and $CR_2$ respectively, then $CR_1 \subseteq CR_2$ if but not only if $P \subseteq Q$.

**Proof sketch:** The proof is of ‘if’ is straightforward. We give an example to illustrate the ‘only if’ does not hold. Let a view and a query be $v = r//a$ and $q = r//a//b$ respectively. Two CRs are $CR_1 = r//a//b$, $CR_2 = r//a/a//b$, produced by either embedding the a-node of $q$ onto $v$ or not, and we have $CR_2 \subseteq CR_1$. However, for $CAT_1 = a/b$ and $CAT_2 = a//a/b$, we have $CAT_2 \not\subset CAT_1$.

Now, we know CAT containment is a sufficient but not necessary condition of CR containment. To further investigate CAT containment, we introduce the following sufficient and necessary condition, which implies an implementation to check CAT containment. A CAT is expressed as a set of component patterns and is not required to be minimized.

**Theorem 4.2** Let $P = \{p_1, p_2, \ldots, p_m\}$, $Q = \{q_1, q_2, \ldots, q_n\}$ be two CATs of rewritings $CR_1$ and $CR_2$ respectively, we have $P \subseteq Q$, iff $\forall j, j \in [1, n], \exists i, i \in [1, m]$, such that $p_i \subseteq q_j$.

**Proof sketch:** We will prove the theorem by using homomorphism technique. Recall Theorem 3 in [MS04], for patterns in $XP^{(//\,/)\,\,}$, the existence of a homomorphism from one pattern to another is a sufficient and necessary condition of pattern containment.
Proof of if. Given that any component pattern of $Q$ can contain some component pattern of $P$, there exists $n$ homomorphisms which map all the $n$ component patterns of $Q$ onto $k$ ($k \leq n$) component patterns of $P$. ($k \leq n$, because one component pattern in $Q$ may contain more than one component patterns in $P$.) Since component patterns of one CAT share the same root, we can build a single homomorphism from $Q$ to $P$ by merging together $n$ homomorphisms for component pattern, and the existence of the homomorphism from $Q$ to $P$ implies $P \subseteq Q$.

Proof of only if. Similarly, $P \subseteq Q$ implies the existence of a homomorphism from $Q$ to $P$. Apparently, the root of $Q$ maps onto the root of $P$. Considering that the root node is the only node all component patterns share, the result follows directly: any component pattern of $Q$ must map onto some component pattern of $P$, i.e. $\forall j, j \in [1, n], \exists i, i \in [1, m]$, such that $p_i \subseteq q_j$.

### 4.3.3 Algorithms to eliminate part of redundant CRs

Following Theorem 4.2, we develop several algorithms to check CAT containment. To avoid repeated computation, we utilize an auxiliary structure, which can be built on the fly during the computation, to record component pattern containment relationships. We call the auxiliary structure component pattern containment matrix, denoted as $M$. $M$ is a square matrix with order $n$, where $n$ is the total number of possible component patterns when rewriting a query using a view. Entry $M[i][j] = 1$ means component pattern $p_i$ is contained in component pattern $p_j$, otherwise $M[i][j] = 0$. The matrix $M$ may not necessarily be a symmetric matrix. If $M[i][j] = M[j][i] = 1$, it means $p_i = p_j$ ($p_i \subseteq p_j \land p_j \subseteq p_i$). Each entry of $M$ is filled only once when necessary. For example, given four CATs $P_1 = \{p_1, p_2\}$, $P_2 = \{p_1, p_3\}$, $P_3 = \{p_4, p_5\}$, $P_4 = \{p_4, p_6\}$, to check containment relationship $P_1 \subseteq P_3, p_1 \subseteq p_4$ needs to be checked once. (let us assume $p_2 \not\subseteq p_4$.)
Algorithm 4.1 Check CAT containment going through P

**Input:** two CATs $P$ and $Q$

**Output:** return true if $P \subseteq Q$, otherwise false

1: $S := Q$
2: for each component pattern $p_i$ in $P$ do
3:   retrieve $c(p_i)$
4:   $S := S - c(p_i)$
5:   for each $s_j$ in $S$ such that $M[r(p_i)][r(s_j)]$ is not defined do
6:     if $p_i \subseteq s_j$ then
7:       $M[r(p_i)][r(s_j)] := 1$ and $S := S - \{s_j\}$
8:     else
9:       $M[r(p_i)][r(s_j)] := 0$
10:   end if
11: end for
12: if $S = \emptyset$ then
13:   return true;
14: end if
15: end for
16: return false;

This operation may be repeated when checking $P_2 \subseteq P_4$, since $p_1$ and $p_4$ also appear in $P_2$ and $P_4$ respectively. Therefore, with containment matrix built, repeated computation can be saved. On the other hand, the containment matrix may not need to be fully filled. For example, entry $M[1][2]$ never needs to be filled, because $p_1$ and $p_2$ never appear in the different CATs.

We now introduce CAT containment algorithms with containment matrix built on the fly. Given two CATs $P$ and $Q$, suppose we want to check if $P \subseteq Q$. 
One alternative is to go through patterns in $P$, shown in Algorithm 4.1. For each component pattern $p_i$ in $P$, remove the patterns in $Q$ that can contain $p_i$. Let $S$ denote the patterns in $Q$ that have not been removed by any $p_i$. Once $S$ becomes $\phi$, we know every $q_j$ in $Q$ can contain some $p_i$, and thus $P \subseteq Q$. Here, $c(p_i)$ denotes the set of patterns in $Q$ that can contain $p_i$ having been found in containment matrix. In line 5-11, the algorithm tries to find the undiscovered patterns that can contain $p_i$ and fills the undetermined entries of $M$. Another choice is to investigate $Q$ first, shown in Algorithm 4.2. For each $q_j$ in $Q$, if we cannot find a $p_i$ such that $p_i \subseteq q_j$, it means $P \not\subseteq Q$. Here, $\overline{c}(q_j)$ denotes the patterns in $P$ that are contained in $q_j$. In both algorithms, a set of component patterns, such as $P$, $Q$, $S$, $c(p_i)$, $\overline{c}(q_j)$ can be realized by a bit vector; the set operations, such as $S := S - c(p_i)$, $\overline{c}(q_j) \cap P$, can be realized by efficient bit vector operations.

One may wonder which algorithm could be superior and whether we can improve them. We now take a deeper look at the algorithms, introduce some heuristic speed-up strategies and propose a more robust hybrid algorithm.

- In the initial step, set $Q = Q - P$. Obviously, patterns in $Q$ that also appear in $P$ will definitely contain some patterns in $P$.

- To investigate larger $p_i$ first. The reason is larger $p_i$ is more likely to be contained in other patterns. This heuristic will make Algorithm 4.1 stop earlier, and also benefit Algorithm 4.2 to break from inner loop earlier.

- To investigate larger $q_j$ first. Similar to the above one, larger $q_j$ is less likely to contain other patterns. This heuristic will help Algorithm 4.2 stop earlier.

Observe Algorithm 4.1, it always tries to demonstrate $P$ is contained in $Q$ until at the last step it ends up with $P \not\subseteq Q$. Algorithm 4.2 does the opposite.
Algorithm 4.2 Check CAT containment going through Q

**Input:** two CATs $P$ and $Q$

**Output:** return true if $P \subseteq Q$, otherwise false

1: for each component pattern $q_j$ in $Q$ do
2:   retrieve $\bar{c}(q_j)$;
3:   if $\bar{c}(q_j) \cap P = \emptyset$ then
4:     temResult := false;
5:     for each $p_i \in P$ such that $M[r(p_i)][r(q_j)]$ is not defined do
6:       if $p_i \subseteq q_j$ then
7:         $M[r(p_i)][r(q_j)] := 1$ and temResult := true;
8:         break;
9:       else
10:         $M[r(p_i)][r(q_j)] := 0$;
11:     end if
12:   end for
13:   if temResult = false then
14:     return false;
15:   end if
16: end if
17: end for
18: return true;
Therefore, if $P$ is indeed contained in $Q$, Algorithm 4.1 may stop earlier, and hence perform better than Algorithm 4.2. If $P \nsubseteq Q$, Algorithm 4.2 may beat its brother. With the heuristic information (the order of selecting patterns) being considered in the algorithms, we are inspired to develop a hybrid algorithm, which has good performance on the average case. The idea is to inspect patterns from $P$ and $Q$ together, i.e. patterns from $P$ and $Q$ are inspected interweavingly, following pattern size from large to small. The hybrid algorithm is shown in Algorithm 4.3, where overlapping parts with Algorithm 4.1 and 4.2 are described with natural language to save space. We compare the three algorithms in Section 4.4.

An important atomic operation in these algorithms is to check containment between two component patterns, see line 6 in both Algorithm 4.1 and 4.2. To check if $p \subseteq q$, the usual way is to check if there exists a homomorphism from $q$ to $p$. The algorithm of finding a homomorphism runs in $O(|p||q|)$. However, in some cases, to discover that $p$ is contained in $q$ only needs $O(1)$ time. We introduce the property as Lemma 4.2 below.

**Lemma 4.2** Let $p$, $q$ be two component patterns leading by two link nodes $n_p$ and $n_q$ respectively. If (1) $ad(n_p, n_q)$ holds, and (2) edges $(parent(n_p), n_p)$ and $(parent(n_q), n_q)$ are both ad-edges, then we have $p \subseteq q$. Here, $parent(n_p)$ and $parent(n_q)$ denotes the parent nodes of $n_p$ and $n_q$.

Lemma 4.2 is not difficult to prove. With Lemma 4.2, we can check component pattern containment on the same path easily.

### 4.4 Experiments

We use a prototype system IMCRE (IMCR Evaluator) to evaluate a generated IMCR on materialized views. Our experiments are conducted on a PC with Pentium(R) 4 3GHz CPU and 1G memory.
Algorithm 4.3 Check CAT containment Hybrid Algorithm

Input: two CATs $P$ and $Q$

Output: return true if $P \subseteq Q$, otherwise false

1: $S := Q - P$;
2: $R := P$;
3: for select the largest component pattern $x$ in $P \cup Q$ do
4: if $x \in P$ then
5: $p_i := x$;
6: do line 3-11 in Algorithm 1;
7: $R := R - \{x\}$;
8: else
9: $q_j := x$;
10: do line 2-16 in Algorithm 2;
11: $S := S - \{x\}$;
12: end if
13: if $S = \emptyset$ then
14: return true;
15: end if
16: if $R = \emptyset$ then
17: return false;
18: end if
19: end for
View and Query Generation Due to the challenge of collecting view and query specimens deployed in real applications (also mentioned in [ODPC06]), we generated views and queries synthetically. In order to cover a variety of cases, the parameters can be tuned within a wide range. To make the generated patterns reasonable, we enforce the generated views and queries to conform to a given DTD, though the query evaluation can be done without knowing the DTD. The view patterns are generated in a top-down manner. For each node in the view query, its children are selected with four parameters: (1) a child node is selected with probability $\alpha_1$, and (2) the edge connecting the child to its parent is labeled as // with probability $\alpha_2$, (3) a descendant node is selected with probability $\alpha_3$ directly connecting to its parent. The maximum fanout $f$ is fixed and set within a limit $f$. We do not generate value predicates in the pattern, because checking value predicate is simple and do not affect the algorithm performance.

The queries are generated based on the views to ensure some rewritings exist. The generation is performed in a bottom-up manner. For each view pattern, (1) a node is deleted with probability $\beta_1$. After deleting the node, if the node is an internal node, we should connect the node’s parent to its children with //’. Note that we never delete the view root. (2) A pc-edge is replaced by an ad-edge with probability $\beta_2$. (3) Some new nodes are added under a node with probabilities similar to the view generation part. We do not set a limit to the fanout of query patterns, because deleting an internal node may increase the out degree of its parent (if the internal node has multiple children), and thus increase the fanout of the result pattern.

Datasets We test our algorithms on two datasets, XMark [TXg] and BIOML [CG]. The former is widely used in the literature, and the latter is famous for its recursive feature, and is ideal to build materialized views with. BIOML DTD is tailored with only “chromosome” and its descendant elements. XML data is
generated with IBM XML Generator [IBM]. We use eXist [eg] database as the underlying engine to store and query the documents. Each materialized view is generated by evaluating the view pattern on documents and saving back into eXist database.

4.4.1 The number of Irredundant CRs

In this section, we test the effectiveness of our method. We investigate the fraction of irredundant CRs among all the CRs, and also the fraction we are able to obtain by carrying out CAT containment pruning. In the figures in this section, CR tags the total number of CRs found by useful embeddings, PCR denotes the left-over CRs after performing CAT containment pruning, ICR means the total number of irredundant CRs.

Firstly, we fix the probability of $\alpha_1$, $\alpha_2$ and $\alpha_3$, and randomly generate 50 views. Then, based on each query, we extract 20 queries varying parameters $\beta_1$ or $\beta_2$. In Fig. 4.5, we fix $\beta_1$ to 0.2 and vary $\beta_2$ from 0.2 to 0.8 with a 0.2 step. The number of CRs is increasing dramatically with the increase of $\beta_2$, while the number of ICRs is increasing as well, but in a gentle trend. This can be explained as: when there is a larger number of ad-edges in a query, there could be more CRs and ICRs. The reason is on each path, there will be more component patterns, since any node, which is leading an ad-edge and also can be mapped onto the distinguished path between $r_v$ and $d_v$ of the view, will produce a component pattern. Component patterns of different paths may be combined arbitrarily, resulting in a large number of CRs. Meanwhile, many of the CRs are redundant due to component pattern containment.

We also fix $\beta_2 = 0$ (do not intentionally change edge type) and vary $\beta_1$ from 0.2 to 0.8. The result is interesting. The number of CRs increases first and then decreases a little at $\beta_1 = 0.8$, see Fig. 4.6. CRs is becoming more, because the
Figure 4.5: The number of CRs and ICRs varying $\beta_2$
Figure 4.6: The number of CRs and ICRs varying $\beta_1$
deleting operation will turn many pc-edges into ad-edges and also increase the fanout of the query. Although the number of component patterns on each path is decreased, the number of paths is increased. As a result, the total number of CRs is still increasing. While at $\beta_1 = 0.8$, the decrease is due to high probability of deleting operation. Four out of five nodes are deleted, resulting the query pattern to shrink, and hence the number of CRs also shrinks.

4.4.2 Performance on Finding the IMCR

We test the time cost of four different algorithms on finding the IMCR. NaivePair is to generate all the CRs and perform a pairwise check between these CRs, while the other three all use CAT containment to prune some redundant CRs first, and they are referred as pFirst, qFirst and Hybrid respectively. The difference is that they use different CAT containment checking algorithms, i.e. pFirst(Algorithm 4.1), qFirst(Algorithm 4.2), Hybrid(Algorithm 4.3). Experiments in Fig. 4.7 show that all the three algorithms with contained CR filtering beat the NaivePair algorithm. Among the better ones, qFirst wins pFirst a little, the reason is that in Algorithm 4.1 and 4.2 the size of $Q$ is usually smaller than the size of $P$. Hybrid is superior to any other algorithm, which confirms our assumption in Section 4.3.3.

We observe that, for both datasets, CAT containment pruning can dramatically reduce the candidate CR set. Although the queries and views we generated from different DTDs are not alike, the conclusions coincide: Most redundant CRs can be attributed to CAT containment. Counterexamples like the one we gave in Section 4.3.2 are very rare.
Figure 4.7: Performance on Finding the IMCR
4.5 Summary

The problem consists of two parts: finding all the irredundant CRs and evaluating them on materialized view. Compared to the previous works, we have given a more precise definition to redundant CR and pointed out the flaw of the previous method in finding irredundant CRs. In the meanwhile, we have given our own solution to this problem, which we believe is complete and accurate. We achieve the goal by firstly expressing a CR in a proper way, and then prune a large part of redundant ones by efficient CAT containment checking algorithms, and finally we compare the left CRs in a pairwise way to find all the truly irredundant ones. Our approach is proved to be correct.
Chapter 5

Evaluating the IMCR

In some applications, such as query caching, query optimization, to find the irredundant maximal contained rewriting (IMCR) is only an intermediate step, not the final goal. We need to utilize the cached query results or intentionally precomputed query results, which we will refer to as materialized views, to quickly answer new queries from the users. Assume we have obtained all the contained rewritings of the IMCR set for a query using a view according to the finding techniques introduced in the former chapter, to continue and complete the computing process, we discuss how to evaluate these contained rewritings on materialized views in this chapter.

5.1 Motivation

A materialized view is often defined as the result of evaluating the view query on a base document. The materialized view is used to speed up the evaluation of other queries. In data integration applications, the base document may be virtual and needs to be integrated from multiple sources.

According to our discussion in Section 2.1.3, the result of an XPath query
on an XML tree is a set of nodes matching the distinguished node of the XPath query. There are two ways to materialize a view query: (1) record the view query and a set of answer nodes; (2) record the view query and a set of subtrees rooted at the answer nodes. The former is suitable for local optimization, when the document is large and indexed, and thus there is no need or it is even infeasible to store a lot of small materialized subtrees, because these materialized views will take up a lot of extra space cost. On the other hand, the second view implementation has its advantage in distributed computing scenarios, such as caching query results. To be specific, previous query results are transmitted from other sites to the query site, and can be used to answer new queries raised at the current site. However, in this chapter, the algorithms we will introduce do not rely on a specific implementation of materialized views, in fact they are orthogonal to the physical implementation of the views. The reason is that our focus is to optimize the evaluation of a set of rewritten queries rather than to speed up the evaluation of a single rewritten query. This point will be clear after we introduce what kind of rewritten queries needed to be evaluated on the materialized view, and how to evaluate the rewritten queries.

First, as to the rewritten queries: After we find the IMCR for a query $q$ using view $v$ according to the techniques from the previous section, each CR in the IMCR is a composed pattern of the view $v$ and a CAT pattern. The view definition part has been satisfied by the materialized view, therefore, we only need to evaluate the CAT pattern on the materialized view. As a result, the problem becomes to evaluate a set of CATs on the materialized view (because each CR corresponds to a CAT), and combine the results.

Second, as to evaluating the CATs on the materialized view: For the following view implementation, we regard the materialized XPath view as a set of subtrees, whose roots have the same tagname as the distinguished node of the view. Let $v_m$
be the materialized view of \( v \), then answering the query \( q \) equals to evaluating the CATs on each subtree from \( v_m \) and then combining the results. Alternatively, if we regard the materialized view as a set of nodes in the XML tree, then evaluating the CATs on the materialized view equals to traversing the original XML tree from the view nodes to find out answer nodes in the subtrees rooted at the materialized view nodes. Note that if inverted-list index has been built on node tag names in the XML tree, we will not have to traverse the XML tree, but simply use structural join [AKJP+02] to find the answer nodes for the CATs. It is easy to observe that, no matter which view implementation we use, we can regard the materialized view as a set of trees (virtual or materialized). To evaluate a CAT on these trees, equals to match the CAT pattern on each (virtual or materialized) tree, and combine the results. For simplicity, in later sections, we only consider one tree of the materialized view, denoted as \( T_v \). Computations on other trees are the same.

The rest of this chapter is organized as follows. We first give out a naive algorithm as a straightforward solution to evaluate CATs on a materialized view in Section 5.2. Then, we introduce the basic algorithm in Section 5.3, which focuses on evaluating component patterns rather than the whole CAT patterns. In Section 5.4, we develop a set of optimizing rules including four pruning rules and three heuristic rules to speed up the evaluation of component patterns. We use examples to demonstrate the effectiveness of those rules. We also show the efficiency gained by applying these rules in the evaluation process in Section 5.5. Finally, we give a summary in Section 5.6.
Algorithm 5.1 Naive Algorithm

**Input:** all CATs of rewriting $q$ using $v$, a materialized view $T_v$

**Output:** the union of the result of evaluating all CATs on $T_v$

1. $F := \emptyset$;
2. for each CAT $P$ do
3. evaluate $P(T_v)$;
4. $F := F \cup P(T_v)$;
5. end for
6. return $F$;

### 5.2 Naive Algorithm

The problem turns out to be: given a set of CAT queries and a data tree $T_v$, how to efficiently evaluate the CAT queries over the data tree $T_v$. From the previous study, we know an IMCR may consist of an exponential number of CR queries, corresponding to an exponential number of CATs, bounded by $k^l$, where $l$ is the number of paths in $q$ (also the number of leaf nodes), $k$ is the maximum number of component patterns that are residing on the same path of $q$. (Usually, $k = 2$, i.e. two component patterns are produced by either embedding the distinguished node of the view or not.) The naive method is to evaluate the exponential number of CATs one by one, and union the final results. Algorithm 5.1 shows the naive algorithm.

### 5.3 Basic Algorithm

In the naive algorithm, we need to evaluate $O(k^l)$ CAT pattern. However, if we consider the characteristics of the CATs, we are able to produce the same results by only evaluating up to $\max\{|N_q|, kl\}$ component patterns. The observation
is that a CAT is composed by a number of component patterns fusing at their roots, and hence one component pattern may be shared by a few different CATs. To evaluate the CATs one by one on a materialized view will result in repeated computation of the same component patterns. As a result, although the number of CATs is exponential, the number of component patterns is polynomial. We now give the bound for the number of component patterns. Since each link node leads a component pattern, and the number of link nodes is bounded by the total number of nodes in a query, denoted as $|N_q|$, so the number of component patterns never exceeds $N_q$. Meanwhile, $k^l$ is also the bound for the number of component patterns, because there are at most $k$ component patterns on one path of query $q$, and there are $l$ paths in the query $q$. The maximum number of component patterns $max\{|N_q|, kl\}$ could be far less than the maximum number of CATs $k^l$.

The idea of our basic algorithm, shown in Algorithm 5.2, is to break down the CATs into component patterns, and evaluate the component patterns. Thus, each component pattern is examined only once. We now introduce the basic algorithm, and then some optimization techniques on the basic algorithm in the next section. Similar to the previous Chapter, we use $P$ to denote a CAT, and $p$ to denote a component pattern. We first evaluate all the predicate component patterns on the materialized view $T_v$, and then, for each CAT, if all the predicate component patterns of this CAT are satisfied on $T_v$, we evaluate the distinguished component pattern of the CAT on $T_v$ and add the result into the final result set $F$. The basic algorithm can be regarded as an optimization of evaluating multiple CATs on the materialized view by taking advantage of the special feature of CATs.
Algorithm 5.2 Basic Algorithm

Input: all CATs of rewriting \( q \) using \( v \), a materialized view \( T_v \)

Output: the union of the result of evaluating all CATs on \( T_v \)

1: union predicate component patterns of each CAT;
2: evaluate all predicate component patterns on \( T_v \);
3: \( F := \phi \);
4: for each CAT \( P \) such that \( \forall p_i \in P, p_i \) is predicate component pattern and 
\( p_i(T_v) = \text{true} \) do
5: evaluate the distinguished component pattern \( \hat{p} \in P \);
6: \( F := F \cup \hat{p}(T_v) \);
7: end for
8: return \( F \);

5.4 Optimizing Techniques

In this section, we first introduce four pruning rules and three heuristic rules to optimize the basic algorithm, and explain the rationale behind these rules. Then we give out the outline of an optimized algorithm to illustrate how to effectively combine these rules together. Finally, we point out that the input CATs do not have to be CATs of an IMCR, they can be CATs of any MCR.

5.4.1 Pruning Rules

In the basic algorithm (Algorithm 5.2), every component pattern is evaluated against the materialized view \( T_v \). In fact, some component patterns may not need to be evaluated. We now introduce several rules to prune those component patterns that we do not need to evaluate, and use examples to illustrate the effectiveness of our rules.
CHAPTER 5. EVALUATING THE IMCR

• **Rule 1**: If one component pattern \( p \) is not satisfied in the materialized view, those CATs that contain \( p \) as a component pattern do not need to be evaluated.

**Example 5.1** *In Fig. 5.1, if component pattern \( p_3 \) is not satisfied, we only need to evaluate component patterns in CAT \( P_3 \) and CAT \( P_4 \), since \( p_3 \) appears in CAT \( P_1 \) and CAT \( P_2 \).*

A CAT can be regarded as a conjunction of its component patterns, and each component pattern is a condition. Any unsatisfied condition will eliminate the CAT from producing answers, no matter whether other component patterns are satisfied or not.

• **Rule 2**: If the answers of one CAT \( P \) are produced, other CATs having the same distinguished component pattern as \( P \) do not need to be evaluated.

**Example 5.2** *In Fig. 5.1, if CAT \( P_1 \) is satisfied, CAT \( P_4 \) can be pruned, since they share the same distinguished component pattern \( p_1 \).*

The result of evaluating a CAT is the same as evaluating the distinguished component pattern of the CAT, while other predicate component patterns only serve as conditions imposed on the root of the materialized view \( T_v \). Therefore, if another CAT \( Q \) has the same distinguished component pattern as CAT \( P \), then no matter what predicate component patterns \( Q \) possesses, if \( Q \) has answers on the materialized view \( T_v \), the answer set will be the same as \( P \)'s.

There are two other optimizing methods by taking advantage of component pattern containment. This type of optimization is based on the fact that checking pattern containment is usually more light-weighted than evaluating a pattern on a materialized view, because view size is usually much larger than pattern size.
Here, notions of $c(p)$ and $\overline{c}(p)$ are consistent with the definitions in Section 4.3.3. To recall them, $c(p)$ denotes the set of component patterns that contain $p$, $\overline{c}(p)$ denotes the set of component patterns that are contained in $p$.

- **Rule 3**: If a component pattern $p$ is satisfied on the materialized view, then any pattern in $c(p)$ will be satisfied on the view.

**Example 5.3** In Fig. 5.1, if $p_1$ or $p_2$ or $p_3$ is satisfied, $p_4$ can be induced to be satisfied.

- **Rule 4**: If a component pattern $p$ is not satisfied on the materialized view, then any component pattern in $\overline{c}(p)$ cannot be satisfied on the view.

**Example 5.4** In Fig. 5.1, if $p_4$ is not satisfied, we know none of $p_1$, $p_2$ and $p_3$ could be satisfied, then CATs $P_1$, $P_2$, $P_3$, $P_4$ are all unsatisfied on the materialized view.

All of the four rules introduced above are trying to reduce unnecessary evaluation of component patterns, i.e. try every means to prune some pattern, and
hence evaluate as fewer component patterns as possible to achieve better performance. For instance, if an unsatisfied pattern is selected and computed early, the CATs containing this unsatisfied pattern will be eliminated early. Obviously, there is an optimal order to schedule these component patterns, i.e. to decide which one is evaluated first, but it is unlikely that we are able to find this order without knowing in advance whether a component pattern is satisfied or not in the materialized view. Similar to checking CAT containment, we have designed some heuristics to find a good evaluation order.

5.4.2 Heuristic Rules

In this section, we introduce a few heuristics to determine the order of evaluating the component patterns of the input CATs. We will first list the heuristics, and discuss the rationale behind the heuristics afterwards.

1. To evaluate frequently shared component patterns first. This rule can be applied to both distinguished component patterns and predicate component patterns. If the pattern is satisfied in the view, Rule 2 and Rule 3 can be applied to prune other component patterns, otherwise Rule 1 and Rule 4 can be used to prune other component patterns. Frequently shared patterns contribute to more number of CATs than rare patterns, and thus are worth to be evaluated first.

2. To investigate component patterns belonging to the same CAT first. The idea of this heuristic is try to produce answers early. Once the component patterns in one CAT are fully evaluated, we can use Rule 2 to prune other CATs sharing the same distinguished component pattern as the computed CAT. This heuristic is specially driven by Rule 2.
3. To group the CATs by their distinguished component patterns. For the CATs in each group, since they share the same distinguished component pattern, it is sufficient to evaluate only one of them. We can start with the CAT with the least number of component patterns or apply the above two heuristics for evaluating this subset of CATs.

All the above heuristics are designed to maximize the effect of applying the pruning rules in Section 5.4.1. Heuristic 1 is more akin to prunings, Rule 3 and Rule 4, and it also has substantial pruning power if the inspected pattern is not satisfied in the materialized view. In the best case, heuristic 1 could remove a maximum number of CATs in one step. The reason is that it always picks the most shared component pattern, and hence if the picked pattern cannot match the materialized view, all the CATs containing that pattern can be disregarded. Heuristic 2 implies an eager strategy to find some results as early as possible by inspecting component patterns in the same CAT. It is akin to the pruning rule, Rule 2, because, once a CAT is found able to produce answers on the materialized view, other CATs containing the same distinguished component pattern can be disregarded. Heuristic 3 also corresponds to Rule 2. After grouping the CATs by their distinguished component patterns, we can apply other heuristic rules within each group. Once a CAT in a certain group is found to be able to produce some results on the materialized view, the CATs in the same group will be freed of examination, according to the pruning Rule 2.

5.4.3 Optimized Algorithm

We want to stress that the heuristics in Section 5.4.2 are orthogonal to the pruning rules in Section 5.4.1. Any heuristic to determine an order of evaluating the component patterns can be integrated into the algorithm shown in Algorithm 5.3.
We now go through Algorithm 5.3 step by step. In the beginning, the result set $F$ is set to $\phi$. Then, each component pattern is evaluated according to an order determined by heuristic rules from line 2 to line 14. In each loop, line 3 evaluates the component pattern $p$ on the materialized view $T_v$. If $p(T_v)$ satisfies ($p$ matches the materialized view), we use Rule 3 to find out other satisfied component patterns by comparing pattern containment in line 5. In line 6, if $p$ is a distinguished component pattern, and all other component patterns of a CAT containing $p$ have been satisfied already, $p(T_v)$ will be added into $F$. This means we have found some answers. Other CATs containing $p$ as the distinguished component pattern will be eliminated using pruning Rule 2 according to line 9. On the other hand, if $p(T_v)$ does not satisfy, we first find out other unsatisfied patterns in line 11, and then use Rule 1 to prune a number of unsatisfied CATs (line 12). In the end (line 15), we return the final answer result $F$.

There are possibly many heuristics to determine a better order to evaluate the component patterns. We cannot list all of them. The three heuristics we have found in Section 5.4.2 are easier to understand, not difficult to implement. Using heuristics and pruning rules, the optimized algorithm is shown to be very effective and efficient in our experimental study.

5.4.4 Discussion

In the above sections, we assume that the CATs of the IMCR are generated first, and then for each CAT of an irredundant CR, we do the evaluation on the materialized view. In fact, we do not require the input to be CATs from the IMCR strictly, because to evaluate any MCR will produce the same set of answers as to evaluate the IMCR. Most of the time, users do not care whether it is the IMCR or an MCR being evaluated, as long as the answers returned are correct. Therefore, when the materialized view is not large, which means to evaluate a few
Algorithm 5.3 Optimized algorithm with pruning rules and heuristics

Input: all CATs of rewriting $q$ using $v$, a materialized view $T_v$

Output: the union of the result of evaluating all CATs on $T_v$

1: final result $F := \emptyset$;

2: for each component pattern $p$ chosen by some heuristic do

3: evaluate $p$ on $T_v$;

4: if $p(v_m)$ satisfies or produces some answers then

5: use Rule 3 to find other component patterns that are also satisfied;

6: if $p$ is a distinguished component pattern $\wedge$ all the predicate component patterns of a CAT containing $p$ as the distinguished component pattern are satisfied then

7: add $p(T_v)$ into the final result $F$;

8: end if

9: use Rule 2 to prune other CATs;

10: else

11: use Rule 4 to find other component patterns that are not satisfied;

12: use Rule 1 to prune other CATs;

13: end if

14: end for

15: return $F$;
extra CATs on the view may not be more expensive than to check the redundancy of all the CRs, we can simply inject an MCR into the system. The algorithms also work with input CATs from an MCR. When evaluating an MCR, we can also check component pattern containment and build containment matrix during the process so that Rule 3 and Rule 4 can efficiently prune some component patterns. Finding $c(p)$ can be added between line 4 and line 5, and finding $\tau(p)$ can be added between line 10 and line 11 in Algorithm 5.3 respectively.

### 5.5 Experiments

We use the same prototype system IMCRE (IMCR Evaluator) to evaluate a generated IMCR on materialized views. Our experiments are conducted on a PC with Pentium(R) 4 3GHz CPU and 1G memory.

**View and Query Generation** Due to the challenge of collecting view and query specimens deployed in real applications (also mentioned in [ODPC06]), we generated views and queries synthetically. In order to cover a variety of cases, the parameters can be tuned within a wide range. To make the generated patterns reasonable, we enforce the generated views and queries to conform to a given DTD, though the query evaluation can be done without knowing the DTD. The view patterns are generated in a top-down manner. For each node in the view query, its children are selected with four parameters: (1) a child node is selected with probability $\alpha_1$, and (2) the edge connecting the child to its parent is labeled as // with probability $\alpha_2$, (3) a descendant node is selected with probability $\alpha_3$ directly connecting to its parent. The maximum fanout $f$ is fixed and set within a limit $f$. We do not generate value predicates in the pattern, because checking value predicate is simple and do not affect the algorithm performance.

The queries are generated based on the views to ensure some rewritings exist.
The generation is performed in a bottom-up manner. For each view pattern, (1) a node is deleted with probability $\beta_1$. After deleting the node, if the node is a internal node, we should connect the node’s parent to its children with //. Note that we never delete the view root. (2) A pc-edge is replaced by an ad-edge with probability $\beta_2$. (3) Some new nodes are added under a node with probabilities similar to the view generation part. We do not set a limit to the fanout of query patterns, because deleting an internal node may increase the out degree of its parent (if the internal node has multiple children), and thus increase the fanout of the result pattern.

**Datasets** We test our algorithms on two datasets, XMark [TXg] and BIOML [CG]. The former is widely used in the literature, and the latter is famous for its recursive feature, and is ideal to build materialized views with. BIOML DTD is tailored with only “chromosome” and its descendant elements. XML data is generated with IBM XML Generator [IBM]. We use eXist [eg] database as the underlying engine to store and query the documents. Each materialized view is generated by evaluating the view pattern on documents and saving back into eXist database.

### 5.5.1 Average Case Study

In this study, we investigate the performance of four different algorithms to evaluate the CAT of the IMCRs on materialized views. We use the same set of views and queries generated above. 50 views are materialized, and a set of 20 queries are rewritten and evaluated on each materialized view. In the NAIVE algorithm, each CAT of a IMCR is evaluated on the materialized. In BASIC algorithm, only component patterns are evaluated on the materialized view, but all the component patterns are computed. In the optimized algorithms, we use the proposed
four rules to prune unnecessary component patterns, and also use heuristic information to schedule component evaluating order. Specifically speaking, in HEU1, we use the first two heuristics with heuristic 2 prior to heuristic 1. In HEU2, we first group the CATs by their distinguished component patterns (heuristic 3), and then apply heuristic 2 and 1.

The result is shown in Fig. 5.2, BASIC algorithm takes almost half time of the NAIVE algorithm, because redundant CATs are pruned in advance, and pruning these CATs is not expensive. Furthermore, HEU1 and HEU2 perform even better, which demonstrates our heuristic methods are very effective and encouraging. It is not obvious to find a better one between HEU1 and HEU2. Although HEU2 seems to provide more effective pruning heuristics, it also suffers in updating component pattern statistics for each distinguished pattern group, and may spare some time on maintaining the auxiliary information.

5.5.2 Best Case Study

In the above experiments, our aim is to test the performance of NAIVE, BASIC, HEU1 and HEU2 algorithms in the average case, where a number of views and a number of queries are randomly generated to capture all pattern types as various as possible. It is reasonable that our heuristic performs best in the average case. One may wonder how far our heuristic algorithms can achieve and what is the worst performance our heuristic methods will reach. We examine the best and worst cases by manually designing two queries, because randomly generated queries are not that extreme.

For the best case, the query is designed to have four paths, and on each path there are three component patterns, two out of which are irredundant. And hence there are $81=3^4$ CR CATs, $16=2^4$ irredundant CR CATs, 8 irredundant component pattern. The query time is shown in Fig. 5.3. On BIOML dataset, the
Figure 5.2: Average Case Study
BASIC algorithm beats NAIVE in two-thirds time, and HEU1 needs only 10% query time of the BASIC algorithm. Similar observation is obtained on XMark dataset, but the result is not that dramatic.

### 5.5.3 Worst Case Study

For the worst case part, we designed a query which does not produce redundant CRs. Every component pattern produced from the query is not contained in its pals. Therefore, in the evaluation process, every component pattern is evaluated on the materialized view, with no one can be pruned. In Fig. 5.4, HEU1 performs almost the same as BASIC, because both of them have evaluated all component patterns. The part of updating heuristic statistics in HEU1 does not apparently degrade HEU1, though it may take some extra time. Both BASIC and HEU1 are a little costly than NAIVE. The reason may be that NAIVE has evaluated less number of patterns, although each larger pattern has a larger size.

### 5.6 Summary

In this chapter, we have investigated the problem of rewriting XPath queries using XPath views, especially on evaluating the irredundant maximal contained rewriting (IMCR). When the irredundant CRs need to be evaluated on materialized views, we reduce the query size and query number by considering only the component patterns, which are frequently shared by those irredundant CRs. We have also developed sound rules and heuristics to avoid unnecessary computation of some component patterns. Our experiments have shown the effectiveness and efficiency of our approaches.

There are some interesting directions to look into in future work. Currently, we do not consider schema information (such as DTD) or other constraints (such
Figure 5.3: Best Case Study
Figure 5.4: Worst Case Study
as data summary) when rewriting a query using a view. Some rewritings may not be redundant in general, but are redundant under some constraints. For example, it was shown in [LWZ06] that if a schema is available, there is only one irredundant CR (all the others are redundant). Another direction is to consider answering a query with multiple views. On this track, equivalent rewriting has been studied in a pioneer work [CDO08], while contained rewriting needs to be investigated.
Chapter 6

Rewriting a Query with Schema

In the previous chapters, we have discussed how to rewrite and evaluate a query using a view with no schema information. In this chapter, we will generalize the problem into view rewriting while taking into account schema information. It is possible that, a query not answerable by a view in general, may be answerable in the presence of a schema, because the schema can constrain the patterns and enforce further conditions into the patterns. Our idea to solve the problem is to propagate schema conditions into the patterns in order to eliminate wildcards and ad-edges so that we can utilize established techniques to find contained rewritings.

6.1 Motivation

We have studied how to find the IMCR for a query using a view, and how to evaluate the IMCR using a materialized view. In real applications, there may be a schema that the views and user’s queries should conform to. To make the work complete, we discuss the query rewriting problem in the presence of a schema in this chapter.

XML schema provides a means to define or constrain XML data. A query $q$
not rewritable using view \( v \) in general, may be actually answerable with view \( v \) under schema constraints. We show a simple example in Fig. 6.1. The view \( v \) contains a \(*\)-node. If there is no schema available, the \(*\)-node can match unlimited number of labels. We cannot restrict the \(*\)-node on label \( c \). As a result, view \( v \) cannot be used to answer query \( q \). On the other hand, if we have a schema \( G \) available (shown on the right side), we can infer from the schema to determine the \(*\)-node as a \( c \)-node. Therefore, view \( v \) can be used to answer query \( q \) in the presence of the schema \( G \).

![Figure 6.1: Query \( q \) is answerable using view \( v \) under schema \( G \)](image)

In this chapter, we focus on the views and queries in \( XP^{(1,1,1)*} \). This XPath subset is larger than the one we study in the previous two chapters. The reason is: when there is no schema available, views containing \(*\)-nodes are unlikely to be usable to answer other queries, because the \(*\)-nodes in a view should only be mapped by \(*\)-nodes in a query, which significantly reduces the possibility to find a valid useful embedding. Fig. 6.2 shows an example. There is a \(*\)-node in the view query. With no schema information at hand, the \(*\)-node may represent any label. If we rewrite the query \( q \) using the view \( v \) as the rewriting \( q' \), obviously
$q'$ is not contained in $q$. It is very likely that we may introduce false negative answers. However, if a schema is given, the $*$-nodes are allowed to be replaced by a limited number of alternatives. In some special case, a $*$-node may have a definite label choice, because other label choices make the view pattern violating the schema.

![Figure 6.2: An unusable view](image)

This chapter is organized as follows: in Section 6.2, we introduce some basic knowledge about XML schema, and how to model XML schema as graphs. We will first focus on DAG schema in Section 6.3. Given a DAG schema, we discuss how to eliminate $*$-nodes and ad-edges, how to chase patterns in $XP^{(\ell)}$ and how to modify the useful embedding conditions to serve DAG patterns. In Section 6.4, we will discuss some insights for using a recursive schema.

### 6.2 Modeling the XML schema as a Graph

Schema information is usually modeled as regular expressions or a few number of constraints. The works [NS03, Woo03] show that containment between
patterns in $XP^{(/,\[\],*, DT D)}$ is EXPTIME-complete, and some more theoretical results w.r.t. various pattern subsets can be found in [Sch04b]. Since the containment problem is already difficult for two single patterns, it is unlikely to have an efficient method to rewrite a query using a view in $XP^{(/,\[\],*)}$ in the presence of a schema. As a result, the aim of our work is not to break the proved EXPTIME-complete upperbound for two queries, nor to provide any exact complexity results for the rewriting problem, but to reexamine the problem from another angle and to suggest a strategy to find a contained rewriting for a query using a view with schema information. The idea is to propagate schema constraints into the view and the query so as to eliminate wildcards and descendant edges. Consequently, the problem could be converted into rewriting problems between queries and views in $XP^{(/,[\])}$. Then, after chasing patterns in $XP^{(/,[\])}$ with schema constraints, we can apply existing techniques to find useful embeddings for a query using a view.

In our paper, we model the schema as a directed graph $G = \langle V_G, E_G \rangle$. (We do not consider disjunctions in the schema.) Graph $G$ is a DAG means that the schema is not recursive, otherwise $G$ will have cycles. We will consider $G$ as a DAG first in Section 6.3 and will discuss recursive schema in Section 6.4.

### 6.3 Rewriting the patterns with Schema

In this section, we investigate how to enforce schema restrictions onto the patterns so that they do not contain “*” and “/”. We first propose how to eliminate wildcards and ad-edges, then discuss how to chase the result patterns in $XP^{(/,[\])}$, finally we discuss how to modify the embedding conditions in Section 2.2.2 to cater to the chased patterns, which may be DAG (directed acyclic graph) patterns.
6.3.1 Eliminating Wildcards

With a schema available, a "*" node can be replaced by one or a few specific labels in the schema alphabet \( \Sigma_G \), as long as the result pattern conforms to the given schema \( G \). A naive method to eliminate those wildcard nodes is to assign an arbitrary label in \( \Sigma_G \) for each wildcard node, and then to verify whether the clearly specified query complies with the schema \( G \). This method requires to verify \( |\Sigma_G|^k \) number of queries, where \( k \) is the number of wildcard nodes in a pattern and is obviously not efficient.

In the following, we propose an improved algorithm to eliminate the wildcard nodes shown in Algorithm 6.1. The basic idea is to use existing structural information in the query to avoid wild guesses. It is inspired by [LRWZ04] in which a similar idea was used to test the satisfiability of a tree pattern under a schema \( G \). Here, in our scenario, we need to record detailed label relationships (parent-child relationship or ancestor-descendant relationship) for adjacent node pairs, because these relationship could be further utilized to transform one query with wildcards into a union of queries without wildcards.

Algorithm 6.1 scans the wildcard nodes in \( p \) twice: in BottomUpMarking() at line 7 and in TopDownMarking() at line 10. The algorithm first initializes the possible label set for each node (line 1 to line 6). The bottom-up phase calculates a set of possible labels \( L(x) \) for each wildcard node \( x \), using information about possible labels of its children. The top-down phase further refines the set \( L(x) \) using information about the parent label of \( x \), and the parent-child or ancestor-descendant relationship is recorded as label pairs in \( P(x) \).

In the BottomUpMarking(), we first mark all the leaf nodes and non *-nodes (line 1). Then we traverse the pattern bottom-up and compute \( L(x) \) for each *-node \( x \). Since all the leaf nodes are marked, it is guaranteed that, in a bottom-up process, all \( x \)'s children \( x_{c_1}, x_{c_2}, \ldots, x_{c_k} \) are marked before computing \( L(x) \).
Algorithm 6.1 Eliminating Wildcard Nodes

Description: $L(x)$ denotes a set of possible labels for node $x$ in $p$; $P(x)$ denotes a set of adjacent label pairs for each node $x$ with its parent node;

Input: a tree pattern query $p$ with wildcards, and a schema graph $G$;

Output: a set of queries $R$; (Result set $R$ is empty if $p$ does not conform to the schema $G$.)

1: for each node $v$ that is not a $*$-node in pattern $p$ do
2:   $L(v) \leftarrow \{\text{label}(v)\}$;
3: end for
4: for each leaf $*$-node $v$ in pattern $p$ do
5:   $L(v) \leftarrow \Sigma$;
6: end for
7: if false = BottomUpMarking() then
8:   return $\phi$;
9: end if
10: if false = TopDownMarking() then
11:   return $\phi$;
12: end if
13: Top-down traverse $p$ from the root, enumerate a set of patterns using $L(x)$, $P(x)$ for each node $x$;
Algorithm 6.2 BottomUpMarking()

Description: the same pattern $p$ with every node marked, and a label set $L(x)$ associated with each node $x$;

Input: a tree pattern query $p$ with wildcards, and a schema graph $G$;

Output: true if $p$ conforms to $G$, false otherwise;

1: Mark all leaf nodes and all non $*$-nodes;
   
   { Traverse $p$ bottom-up and do the following; }

2: repeat

3: for each $*$-node $x$ in $p$ whose children $x_{c1}, x_{c2}, \ldots, x_{ck}$ are all marked do

4: for $i = 1$ to $k$ do

5: $S_i \leftarrow \phi$;

6: for each $\beta \in L(x_{c_i})$ do

7: if $(x, x_{c_i})$ is a pc-edge and there is some $\alpha \in L(x)$ such that $(\alpha, \beta)$ is an edge in $G$) or $(x, x_{c_i})$ is an ad-edge and there is some $\alpha \in L(x)$ such that there is a path from $\alpha$ to $\beta$ in $G$) then

8: $S_i \leftarrow S_i \cup \{\alpha\}$;

9: end if

10: end for

11: end for

12: $L(x) \leftarrow \bigcap_{i=1}^{k} S_i$;

13: if $L(x) = \phi$ then

14: return $false$;

15: end if

16: Mark $x$;

17: end for

18: until all $*$-nodes in $p$ are marked

19: return $true$;
Algorithm 6.3 TopDownMarking()

**Description:** the same pattern $p$ with every node marked, a label set $L(x)$ associated with each node $x$, an adjacent label pair set $P(x)$ for each node $x$;

**Input:** a tree pattern query $p$ with wildcards, and a schema graph $G$;

**Output:** true if pattern $p$ conforms to $G$, false otherwise;

1: Unmark the root node of $p$ and all non $\ast$-nodes;
   { Traverse $p$ Top-down and do the following; }
2: repeat
3:   for each $\ast$-node $x$ in $p$ whose parent $x_p$ is unmarked do
4:     for each $\beta \in L(x)$ do
5:       if $(x_p, x)$ is a pc-edge and there is some $\alpha \in L(x_p)$ such that $(\alpha, \beta)$ is an edge in $G$) or $(x_p, x)$ is an ad-edge and there is some $\alpha \in L(x_p)$ such that there is a path from $\alpha$ to $\beta$ in $G$) then
6:         Add $(\beta, \alpha)$ into $P(x)$;
7:       else
8:         Remove $\beta$ from $L(x)$;
9:       if $L(x) = \phi$ then
10:          return $false$;
11:     end if
12:   end for
13: Unmark $x$;
14: end for
15: until all $\ast$-nodes are unmarked
16: return $true$;
(line 3). For each child $x_{c_i}$, we calculate the possible label set $S_i$ for $x$ (line 4 to 11). Line 12 combines the results for all $x_{c_i}, i \in [1, k]$. If $L(x) = \phi$, it means no label can replace the $*$-node, reflecting $p$ cannot be satisfied, so we return $false$ (line 13 to 15). Otherwise, we mark the node $x$ and repeat the loop until every $*$-node is marked (line 16). After the BottomUpMarking() procedure, each $*$-node has associated with an initial label set $L(x)$. The BottomUpMarking() process runs in $O(|E_p||V_G||E_G|)$. In pattern $p$, each edge is visited at most once (line 3). For each edge, the number of labels in $L(x_{c_i})$ is bounded by $|E_G|$ (line 6), and checking line 7 can be done in $|E_G|$. As a result, the complexity of the bottom-up marking process is $O(|V_p||V_G||E_G|)$.

In the TopDownMarking(), we further purify the set $L(x)$. At the beginning, all non $*$-nodes and the root node are unmarked (line 1). Then we traverse the pattern top-down and refine $L(x)$ for each $*$-node $x$. Since the root node is unmarked, it is guaranteed that, in a top-down process, all $x$’s parent $x_p$ is unmarked (line 3). For each label $\beta$ in $L(x)$, we check whether there exists a label $\alpha$ in $L(x_p)$ so that $\alpha$ and $\beta$ can satisfy the edge relationship in schema $G$ (line 5). If so, we add $(\beta, \alpha)$ into $P(x)$ to record the relationship (line 6), otherwise we remove $\beta$ from $L(x)$ (line 8). If $L(x) = \phi$, it means no label can replace the $*$-node, reflecting $p$ cannot be satisfied, so we return $false$ (line 9 to 11). At last, we unmark the node $x$ and repeat the loop until every $*$-node is unmarked (line 14). After the TopDownMarking() procedure, each $*$-node has associated with a final label set $L(x)$ and a label pair set $P(x)$ recording the possible label relationship with the node’s parent. The TopDownMarking() process runs in $O(|V_p||V_G||E_G|)$, each node is checked at most once (line 3). For each node with its parent, to refine the label set $L(x)$ and create the relationship set $P(x)$, we have a similar complexity $O(|V_G||E_G|)$ as the BottomUpMarking() process. As a result, the complexity of TopDownMarking() is $O(|V_p||V_G||E_G|)$. 


Algorithm 6.1 runs in $O((|V_p| + |E_p|)|V_G||E_G|)$ except for the last step (line 13) for enumerating a union of patterns. If there is an index built on the schema $G$ such that $pc(\alpha, \beta)$ or $ad(\alpha, \beta)$ can be checked efficiently, the complexity of the algorithm will be in $O((|V_p| + |E_p|)|V_G|)$ regardless of the last step. Of course, the last step to list all the possible alternatives using $P(x)$ is the most expensive step.

6.3.2 Eliminating ad-edges

Now we have transformed a pattern with wildcards into a union of queries without wildcards. However, applying the techniques in Chapter 4 directly is still not sufficient to decide correctly the answerability for a query using a view. Fig. 6.3 shows an example. Query $q$ is not answerable using view $v$ without schema, because path $a/c/e$ cannot find an embedding on the view. However, after given the schema $G$, we can replace ad-edge $a//e$ with $a/c/e$ using the schema information. Afterwards, view $v$ can be used to answer query $q$. In this section, we introduce how to replace all the ad-edges with concrete paths compromising only pc-edges, because ad-edges must be interpreted in specific ways constrained by the schema.

A naive method to eliminate an ad-edge $(v_1, v_2)$, similar to eliminating wildcard nodes, is to find all the paths between two labels $\text{label}(v_1)$ and $\text{label}(v_2)$ in the schema $G$, and replace the ad-edges with one of these concrete paths. For other ad-edges, the same process is performed. Obviously, there may be many ways to replace an ad-edge, and thus a pattern consisting a number of ad-edges will be transformed into a union of a large number of patterns in $XP^{(\text{\textbackslash\textbackslash})}$. Then with the follow-up treatment in Section 6.3.3 and Section 6.3.4, one can determine the answerability for a query using a union of views in $XP^{(\text{\textbackslash\textbackslash})}$. Follow-up details will be introduced later. We focus on how to eliminate ad-edges effectively and efficiently in this section.
To avoid generating a possibly exponential number of patterns in terms of the number of ad-edges, a better solution is to wisely replace an ad-edge \((v_1, v_2)\) with a subgraph between \(\text{label}(v_1)\) and \(\text{label}(v_2)\) in \(G\), denoted as \(G_s(\text{label}(v_1), \text{label}(v_2))\).

We define \(G_s(\text{label}(v_1), \text{label}(v_2))\) formally:

**Definition 6.1** \(G_s(\text{label}(v_1), \text{label}(v_2)) = <V, E>\) is an induced subgraph of the schema graph \(G = <V_G, E_G>\), where \(V = \{\text{node } v | v \in V_G \land \text{v is reachable from } \text{label}(v_1) \land \text{label}(v_2) \text{ is reachable from } v\}\) and \(E = \{(v_1, v_2)| (v_1, v_2) \in E_G \land v_1 \in V \land v_2 \in V\}\). Here, we regard a node as reachable from itself.

As long as the original given pattern \(p\) conforms to schema \(G\), \(\text{label}(v_2)\) is always reachable from \(\text{label}(v_1)\) in \(G\), i.e. induced subgraph \(G_s(\text{label}(v_1), \text{label}(v_2))\) will always exist, no matter \(G\) has cycles or not. In addition, to find the subgraph \(G_s(\text{label}(v_1), \text{label}(v_2))\) is not expensive if \(G\) is a DAG. Algorithm 6.4 shows the process to find the induced subgraph \(G_s(\text{label}(v_1), \text{label}(v_2))\). It includes a depth-first search (DFS) or breath-first search (BFS) traversal in \(G\) from \(\text{label}(v_1)\) to other nodes, and a reverse depth-first search (DFS) or breath-first search (BFS)
traversal from \( \text{label}(v_2) \) to other nodes. The nodes visited in both traversals belong to the subgraph \( G_s(\text{label}(v_1), \text{label}(v_2)) \). Algorithm 6.4 is based on two graph traversal processes, and hence the complexity is \( O(|V_G| + |E_G|) \);

**Algorithm 6.4** Finding the subgraph between \( \text{label}(v_1) \) and \( \text{label}(v_2) \)

**Input:** a schema graph \( G \) and two given labels \( \text{label}(v_1), \text{label}(v_2) \);

**Output:** an induced subgraph between \( \text{label}(v_1) \) and \( \text{label}(v_2) \) in \( G \).

1. Get a reverse graph \( G' \) from \( G \);
   
   \[
   \{ \text{\( G' \) is empty initially, if edge (a, b) is in \( G \), then add (b, a) into \( G' \). } \}
   \]

2. DFS (or BFS) from \( \text{label}(v_1) \) in \( G \), and mark each visited node in \( G \) RED;

3. DFS (or BFS) from \( \text{label}(v_2) \) in \( G' \), and mark each visited node in \( G' \) BLUE;
   
   \[
   \{ \text{If a label is marked RED in \( G \) and BLUE in \( G' \) respectively, it is in } \]

   \[
   \{ \text{\( G_s(\text{label}(v_1), \text{label}(v_2)) \), because it is reachable from } \text{\( \text{label}(v_1) \) and can reach } \]

   \[
   \text{\( \text{label}(v_2) \)). } \}
   \]

4. return \( G_s(\text{label}(v_1), \text{label}(v_2)) \);

### 6.3.3 Chasing Patterns in \( P\{/, []\} \)

Now we have wildcards and ad-edges eliminated, and all the patterns transformed into \( XP\{/, []\} \). To reduce the problem into one without schema, we have the last step to chase the patterns in \( XP\{/, []\} \) as much as possible with sibling constraints and functional constraints [Woo03]. We introduce the constraints as follows:

- **Sibling Constraint:** \( a : B \downarrow c \) means if whenever a node labeled \( a \) in a pattern has children labeled with each \( b \in B \), it has a child node labeled \( c \). When \( B = \phi \), the sibling constraint is called child constraint, a special case of sibling constraints.
• Functional Constraint: $a \downarrow b$ if no node labeled $a$ in a pattern has two distinct children labeled with $b$.

When the given schema $G$ is a non-recursive schema, the chasing process is not difficult, since the result pattern (after chasing) should be finite. The problem is then converted into rewriting a query with a union of views without schema, and the views are in $XP\{/\}$ or $X\{\}$. Thereafter, we can use existing solutions to solve the problem. However, the conditions for useful embedding should be modified, because the current patterns have been converted into DAGs. We will elaborate this part in Section 6.3.4. We will also discuss how to chase patterns using cyclic schema graph in Section 6.4.

6.3.4 Modified Useful Embedding

After replacing ad-edges with subgraphs, a pattern may become a DAG rather than a rigorous tree pattern in $XP\{/\}$. Therefore, the conditions for identifying useful embeddings need to be modified to adapt to DAG views and queries. In the following, we consider the view pattern and the query pattern as DAGs. Firstly, in the chase process, we may not apply sibling constraints at some node whose child nodes are following or-semantics, because these child nodes are expanded from an ad-edge expressed as alternative paths (or subgraphs), making sibling constraints not satisfied on such node. On the other hand, when homomorphism is used to detect containment between such or-semantic patterns, a final step needs to be added. That is, when we conduct a mapping from pattern $q$ to $v$, we can draw the conclusion that there is a useful embedding from $q$ to $v$, if, other than the conditions stated in Definition 2.6, another two conditions should hold as well:

1. For every alternative subgraph connecting $v_1$ and $v_2$ in the induced graph
\( G_s(label(v_1), label(v_2)) \) in \( q \), one of its subgraph connecting \( v_1 \) and \( v_2 \) must be mapped onto \( v \);

2. For every two nodes \( v'_1 \) and \( v'_2 \) with an ad-edge in \( v \), if \( v'_2 \) has an image in \( q \), then every \( v'_2 \)'s ancestor (on all alternative subgraphs) should be mapped by some node in \( q \).

### 6.4 Recursive Schema

One challenge arises: if the schema is recursive, a pattern can be chased continuously without stop, and a /-path (path only consisting of pc-edges) may contain a cycle repeated for any times. In such cases, we allow the loop to appear once in the chased pattern to keep track of the nodes in a cycle, and we also tag the loop start node and loop end node. Now we are able to rewrite a query in \( XP\{/,\![\],\ast,DTD\} \) into a union of finite number of queries in \( XP\{/\![\]\}\} \).

To find the useful embeddings, a further condition needs to be added when to find an embedding from a query \( q \) to a view \( v \). Here, \( q, v \) are in \( XP\{/\![\]\}\} \) with loop start node and loop end node identified: let \( v_1 \) and \( v_2 \) be a pair of loop start node and loop end node in \( q \), hence there must exist a cycle in \( G \) with labels \( label(h(v_1)) \), \( label(h(v_2)) \) as start label and end label respectively. Here, \( h(v_1) \) and \( h(v_2) \) may not be loop start and end nodes in view \( v \).

Fig. 6.4 shows an example. To simplify the problem, we consider pattern containment only, since useful embedding can be determined after detecting pattern containment. Here, \( p_0 \) and \( q_0 \) are two patterns involving ad-edges. \( G \) is a schema containing a cycle. After expanding ad-edges and chasing with schema \( G \) for \( p_0 \) and \( q_0 \), we get patterns \( p \) and \( q \). In pattern \( q \), \( a//c \) is expanded and chased into \( a/b/c \) with node \( v_1 \) labeled \( a \), node \( v_2 \) labeled \( c \) as the loop start and loop end. Similarly, \( b//a \) is expanded and chased into \( b/c/a \) in \( p \). Considering the result
patterns $p$ and $q$, there is a homomorphism from $q$ to $p$, and moreover, nodes $h(v_1)$ and $h(v_2)$, the corresponding nodes in $p$ of loop start $v_1$ and loop end $v_2$ in $q$, have labels $a = label(h(v_1))$ and $c = label(h(v_2))$ that are start and end nodes of a cycle in $G$. Therefore, pattern $p$ is contained in pattern $q$, and thus the original pattern $p_0$ is contained in $q_0$. Note that the condition does not require $h(v_1)$ and $h(v_2)$ be the loop start and loop end nodes in $p$.

![Figure 6.4: Rewriting under Recursive Schema](image)

In the above discussion, we assume that there are no intersected cycles in $G$, i.e. the recursive loops have no overlaps. This assumption obviously simplifies the problem, and it is still interesting and challenging to investigate the rewriting problem for queries and views under complex recursive schema.

### 6.5 Summary

In this chapter, we have discussed how to utilize schema information to rewrite a query using a view. The discussion is motivated by the observation that a query not answerable using a view in general, may be answerable using a view under
schema constraints. The idea to solve the problem is to enforce schema constraints into the queries and views by eliminating wildcard nodes and ad-edges. Then, after chasing the patterns maximally, we can apply existing techniques to find contained rewritings for queries using views. The conditions for finding useful embeddings need to be modified accordingly. Using recursive schema is also discussed, but not studied completely.
Chapter 7

Filtering Techniques for Finding Rewritings

In real applications, a large number of queries may be submitted to the system at one time, which challenges the scalability and robustness of the system. We need to figure out which queries can be answered with the materialized views in storage and which queries must be evaluated on the real database so that materialized views can be fully exploited to speed up query evaluation. This motivates us to design filtering techniques in order to quickly discover whether a query can be answered by a view, and to take actions early, such as using materialized views to answer the query, or deliver the query to the underlying database or simply reject the query.

7.1 Motivation

Before introducing the filtering techniques for rewriting XPath queries using views, we explain where the filtering part is deployed in the system, how it
works, and under what circumstance it is able to upgrade the system performance. Fig. 7.1 shows a system framework without query filtering function. Users pour a larger number of queries into the system, and these queries are divided into two streams. For those that we can use materialized views to answer, we endeavor to find rewritings for these queries using the virtual view definitions, and then evaluate the rewritings on materialized views. On the other hand, for those that cannot be answered by materialized views, we deliver them directly into the database and evaluate them using the base data.

![Diagram](image)

**Figure 7.1:** The framework of answering queries without filtering

Fig. 7.2 gives out the system framework with filtering function. A filter module is placed at front to quickly discover the unanswerable queries from users. These unanswerable queries are directly evaluated on the underlying database, on the other hand, the rest possibly-answerable queries will follow the normal process as Fig. 7.1, classified into two streams by the virtual views. Those queries which passed the filter but cannot be answered by materialized views, are called false positives of the filter. We summarize three important aspects in designing the filter as follows:
CHAPTER 7. FILTERING TECHNIQUES FOR FINDING REWRITINGS

- Filtering should be more time efficient than computing a rewriting, otherwise we would rather directly find rewritings for the input queries.

- The filter should not introduce false negatives, which means if we can find a rewriting for a query using a view, the query should not be filtered out.

- The filter should be effective, which means the fewer false positives passing the filter, the better the filter is.

![Diagram](image)

Figure 7.2: The framework of answering queries with filtering

We now introduce the cost for directly finding two types of rewritings. Given a query $Q$ and a view $V$, to test whether the query $Q$ can be answered by view $V$ (i.e. to test whether there exists a rewriting for $Q$ using $V$) is P-TIME, with complexity $O(|N_q||N_v|)$ for equivalent rewriting [XÖ05] and $O(|N_q||N_v|^2)$ for contained rewriting [LWZ06], where $|N_q|$, $|N_v|$ are the size of the query and the view respectively. Since the query and the view are tree patterns, we have $|N_q| = |E_q| + 1$, $|N_v| = |E_v| + 1$, because the number of nodes are more than the number of edges by 1 for both the query and the view. Consequently, we can use $|N_q|$, $|N_v|$ to reflect the size of the query and the view. Actually the above
CHAPTER 7. FILTERING TECHNIQUES FOR FINDING REWRITINGS

Complexity result is for XPath subset $XP^{(\cdot/\cdot/\cdot)}$. If we deal with queries and views in $XP^{(\cdot/\cdot/\cdot\cdot)}$, both rewriting-discovery problems will fall into $\text{coNP}$-hard problems [XÜ05, LWZ06]. As a result, considering the case that users issue a large number of queries against a view $V$, it is of great importance if we can cheaply filter out part of unanswerable queries for $V$ as early as possible, and thus a lot of computation cost will be saved. In real applications, there may also be a large number of views. For instance, in data integration, many sources manage their data with their own schema, which can be regarded as views of a virtual global schema (often referred to Local As View architecture). Then, for a query issued on the global schema, we need to rewrite the query using each local schema so that the query can find corresponding answers in each data source. Views will populate as data sources are inflating nowadays. Therefore, it is not difficult to see that filtering techniques can play a significant role when we have to deal with plenty of queries and views.

In this chapter, we devise a set of $O(|N_q|)$ algorithms to filter users’ queries. We study the filtering function for both equivalent rewriting and contained rewriting. The basic idea is to verify whether the structural relationships in a query could be satisfied in a view, given that label preserving and structure preserving are the key conditions in discovering a homomorphism (for finding an equivalent rewriting) or a useful embedding (for finding a contained rewriting). We use index to capture the structural relationships in the view, and develop two algorithms for equivalent rewriting, i.e. Lazy Algorithm and Eager Algorithm. Moreover, Eager Algorithm can be modified to support contained rewriting. After studying all of the above for queries and views in subset $XP^{(\cdot/\cdot/\cdot)}$, featuring child axes, descendant axes and branches in XPath, we further discuss the problem for XPath subset $XP^{(\cdot/\cdot/\cdot\cdot)}$ including wildcards. Our contributions are highlighted as follows:
• We propose a novel index to capture the relationships between nodes in a view, and the index can be utilized to filter unanswerable queries.

• We devise a set of linear algorithms to filter queries with respect to both equivalent rewriting and contained rewriting.

• We study the problem with both queries and views in subclass \( XP^{/}//[\ldots]\), which contains most popular features in XPath.

• We implement extensive experiments to justify the efficiency and effectiveness of our filtering algorithms.

The rest of this chapter is organized as follows. We introduce the filtering index serving as the base for filtering techniques in Section 7.2. The filtering algorithms will follow in Section 7.3 including two algorithms for equivalent rewriting and one for contained rewriting. In Section 7.4, we extend XPath queries from \( XP^{/}//[\ldots]\) to a larger subset \( XP^{/}//[\ldots]\). Finally, we draw a conclusion in Section 7.5.

### 7.2 Building Index for Filtering

In this section, we will introduce how to build an index for a view so that we can use the index to filter out answerable queries. We only focus on index building in this section, and will explain how to use the index in the Section 7.3.

We now introduce the important index \( I \) built on a view \( V \). Index \( I \) consists of two parts, \( I_{pc} \) and \( I_{ad} \). Each part is a matrix (which can be implemented by a two-dimensional array) containing \( |\Sigma_v|^2 \) entries, where \( |\Sigma_v| \) is the number of distinct labels in view \( V \). Each entry is associated with a label pair \( (lab_1, lab_2) \) (where \( lab_1, lab_2 \in \Sigma_v \)), and has a bit vector with length \( |\text{paths}(V)| \). Here, \( |\text{paths}(V)| \)
is the number of paths in view $V$. We denote the bit vector as $I_{pc}(lab_1, lab_2)$. 
(Similarly, $I_{ad}(lab_1, lab_2)$ for index $I_{ad}$.) We use $I_{pc}(lab_1, lab_2)[i]$ to represent the $i$th bit of $I_{pc}(lab_1, lab_2)$, and obviously $I_{pc}(lab_1, lab_2)[i] \in \{0, 1\}$. If every bit of $I_{pc}(lab_1, lab_2)$ is “0”, we say $I_{pc}(lab_1, lab_2) = 0$. (An example of index $I$ will be given later in Example 7.1.)

We utilize index $I$ to capture the pc(ad)-relationship between two nodes in view $V$. Precisely speaking, take pc-relationship as an example, if $pc(n_1, n_2)$ holds on the $i$th path in $V$, and let $label(n_1) = lab_1$, $label(n_2) = lab_2$, then $I_{pc}(lab_1, lab_2)[i] = 1$. Similar principle can be applied on ad-relationship, i.e. if $ad(n_1, n_2)$ holds, then $I_{ad}(label(n_1), label(n_2))[i] = 1$. Note that, here for $I_{ad}(label(n_1), label(n_2))$, $n_1, n_2$ do not have to be adjacent nodes.

**Example 7.1** Fig. 7.3 gives a view $V$ and its corresponding index $I_{pc}$ and $I_{ad}$. For the view $a[/d]/b/c$, there are two pc-relationship, $a/b$ and $b/c$, and both of them are on the path $P_2$, therefore in index $I_{pc}$, $I_{pc}(a, b)[2]$ and $I_{pc}(b, c)[2]$ are set to “1”. Similarly, $ad(a, b)$, $ad(b, c)$, $ad(a, c)$ are satisfied on path $P_2$, and $ad(a, d)$ is satisfied on path $P_1$, therefore in index $I_{ad}$, $I_{ad}(a, b)[2]$, $I_{ad}(b, c)[2]$, $I_{ad}(a, c)[2]$ are set to “1”, and $I_{ad}(a, d)[1]$ is set to “1”.

Finding an algorithm to build index $I$ using view $V$ is not difficult. Algorithm 7.1 shows an example. Line 1 takes $O(|N_v|)$ time, searching for the leaf nodes and numbering them. The number of leaf nodes $N_{leaf}$ equals to $|paths(V)|$. Line 2 runs in $O(|N_v|)$ and finds the corresponding path set $P(n)$ for each node $n$. Each path in $P(n)$ runs through node $n$. Line 3-7 assigns values to the entries in $I_{pc}$ and $I_{ad}$. Since we have to set values for $|\Sigma_v|^2$ entries in $I$, it is not surprising that this step needs $O(|\Sigma_v|^2)$ time, bounded by $O(|N_v|^2)$. Note that we can use a bit-vector to represent $P(n)$, and thus $P(n_1) \cap P(n_2)$ can be regarded as an $O(1)$ operation. Building index is not of linear complexity, but once we have the index
built, we can devise linear algorithms to filter out large part of unanswerable queries.

7.3 Filtering Algorithms for Rewritings

In this section, we will first introduce a basic idea to filter out an unanswerable query \( Q \) using view \( V \). Then based on the basic idea, we will give two algorithms, Lazy Algorithm and Eager Algorithm, to detect whether it is impossible to find an equivalent rewriting for \( Q \) using \( V \). Eager Algorithm can be modified to filter queries with respect to contained rewritings.

7.3.1 Basic Idea

Recall in Section 2.2, label preserving and structure preserving are the key conditions in discovering a homomorphism (for equivalent rewriting) or a useful embedding (for contained rewriting). It is desirable if we can efficiently discover that the structural relationships in query \( Q \) could not be satisfied in view \( V \). We explain it with the following example.
Algorithm 7.1 Constructing Index for View V

Input: a view $V = (N_v, E_v, r_v, d_v, \Sigma_v)$

Output: an Index $I$

1: Find the leaf node set $N_{\text{leaf}} \subset N_v$ and number the leaf nodes;
2: Traversing $V$ bottom-up, find the path set $P(n)$ for each node $n$;
3: for all node pair $(n_1, n_2)$ in $V$ do
4: if $\text{pc}(n_1, n_2)$ holds then
5: For $I_{\text{pc}}(\text{label}(n_1), \text{label}(n_2))$, set bit positions in $P(n_1) \cap P(n_2)$ to 1;
6: end if
7: For $I_{\text{ad}}(\text{label}(n_1), \text{label}(n_2))$, set bit positions in $P(n_1) \cap P(n_2)$ to 1;
8: end for
9: return $I$

Example 7.2 See Fig. 7.4, assume that indexes have been built for $V_1$, $V_2$, $V_3$ and $V_4$ already, $Q$ is unanswerable w.r.t. $V_1$, because $\text{pc}(a, b)$ and $\text{pc}(b, c)$ do not hold in $V_1$; $Q$ is unanswerable w.r.t. $V_2$ as well, because $\text{pc}(a, b)$ and $\text{pc}(b, c)$ do not hold on the same path in $V_2$, whereas $a/b$ and $b/c$ are on the same path in $Q$. However, $Q$ cannot be filtered out by $V_3$ and $V_4$. We can answer $Q$ with $V_3$. But for $V_4$, $Q$ is a false positive since there does not exist a rewriting for $Q$ using $V_4$, though $\text{pc}(a, b)$ and $\text{pc}(b, c)$ hold on the same path in $V_4$ as they are in $Q$.

To summarize, given a query $Q$ and a view $V$, let $e_1 = (n_1, n_2)$, $e_2 = (n_3, n_4)$ be two edges on the same path in $Q$ (assume $e_1$, $e_2$ to be pc-edges without loss of generality), it is impossible to find an equivalent rewriting for $Q$ using $V$ if the following holds:

$$I_{\text{pc}}(\text{label}(n_1), \text{label}(n_2)) \land I_{\text{pc}}(\text{label}(n_3), \text{label}(n_4)) = 0$$
Figure 7.4: Basic idea of filtering algorithms

which means if label pairs, \((\text{label}(n_1), \text{label}(n_2))\) and \((\text{label}(n_3), \text{label}(n_4))\), do not lie on the same path in view \(V\), we cannot find a rewriting for query \(Q\). Note that, here, if \((n_1, n_2)\) is an ad-edge, we use \(I_{ad}(\text{label}(n_1), \text{label}(n_2))\) instead of \(I_{pc}(\text{label}(n_1), \text{label}(n_2))\). The above observation is the key idea in our following algorithms.

Here, the complexity of the bit-And operation depends on the size of the bit vectors (is proportional to the length of bit vectors), which is \(|\text{paths}(V)|\), the number of paths in the view. But in practical settings, this length is probably limited by computer word length, or a small multiple thereof, so this may not impact the actual performance much. We treated this operation as \(O(1)\) in the following paragraphs.

### 7.3.2 Equivalent Rewriting

For equivalent rewriting, the first step is to trim \(Q\) into \(Q'\) according to Lemma 7.1 (which first appeared as Lemma 4.7 in [XÖ05]). The above step can be done by firstly finding a node \(n\) on the distinguished path of \(Q\), satisfying \(\text{length}(r_v, d_v) = \text{length}(r_q, n)\), where \(\text{length}(n_1, n_2)\) means the number of path steps between node
n_1 \text{ and node } n_2; \text{ and secondly removing all the descendants of } n \text{ in } Q, \text{ i.e. } Q' = Q_{up}(n). \text{ The reason to trim } Q \text{ into } Q_{up}(n) \text{ is: nodes under } n \text{ do not contribute to detecting the existence of an equivalent rewriting, since } n \text{ is supposed to be mapped onto } d_v. \text{ This step runs in } O(|N_q|) \text{ and serves as the first step in both Lazy Algorithm and Eager Algorithm.}

**Lemma 7.1** Let \( p_1 \) and \( p_2 \) be two XPath tree patterns. If \( p_1 \) and \( p_2 \) are equivalent, the selection path of \( p_1 \) has the same size as that of \( p_2 \) for XPath subset \( XP\{/,//,\ast,[\cdot]\} \), where the selection path of a pattern \( p \) means the path from \( p \text{’s root to } p \text{’s output node.}

**Lazy Algorithm**

We now introduce the main part of Lazy Algorithm (see Algorithm 7.2). (The meaning of word “lazy” will be clear after we introduce Eager Algorithm.) After trimming \( Q \) into \( Q_{up}(n) \), we identify, for each path \( p_i \) in \( Q_{up}(n) \), if the edge relationships on \( p_i \) can be satisfied on a single path in view \( V \). We do this by retrieving the bit vector of each label pair \((\text{label}(n_1),\text{label}(n_2))\) that associates an edge \((n_1, n_2)\) on \( p_i \) from index \( I \), and performing bit-AND(\( \wedge \)) operation to the bit vectors (line 3-10). If there does not exist any bit set to “1” in the result bit vector, which means there does not exist a path in \( V \) that can accommodate all edge relationship on \( p_i \), then \( Q \) can be filtered out (line 11-13); otherwise we go on to test other paths in \( Q_{up}(n) \).

**Example 7.3** In Fig. 7.5, query \( Q \) is unanswerable using view \( V_1 \), because \( ad(a,e) \) is not satisfied on any path in \( V_1 \). For view \( V_2 \), \( Q \) is answerable. However, if we want to find a contained rewriting rather than equivalent rewriting, query \( Q \) turns to be answerable using view \( V_1 \). We will introduce the algorithm for dealing with contained rewriting in Section 7.3.3.
Algorithm 7.2 Lazy Algorithm for Filtering Equivalent Rewriting

**Input:** a query $Q$, a view $V$ and an index $I$ on $V$

**Output:** a boolean value (false: if impossible to answer)

1: Find a node $n$ on the distinguished path of $Q$, such that $\text{length}(r_v, d_v) = \text{length}(r_q, n)$, and trim $Q$ into $Q_{up}(n)$;

2: **for** each path $p_i \in Q_{up}(n)$ **do**

3: Initiate a bit vector $I_0$ with length $|\text{paths}(Q_{up}(n))|$ and every position set to 1;

4: **for** each edge $(n_1, n_2)$ on $p_i$ **do**

5: if $(n_1, n_2)$ is a pc-edge **then**

6: $I_0 := I_0 \land I_{pc}(\text{label}(n_1), \text{label}(n_2))$;

7: **else**

8: $I_0 := I_0 \land I_{ad}(\text{label}(n_1), \text{label}(n_2))$;

9: **end if**

10: **end for**

11: if $I_0 = 0$ **then**

12: **return** false;

13: **end if**

14: **end for**

15: **return** true;
Lazy Algorithm runs in $O(|N_q||E_q|)$, where $|N_q|$ is the number of nodes in $Q$, $|E_q|$ is the number of edges in $Q$. $O(|N_q||E_q|)$ is bounded by $O(|N_q|^2)$, since $|E_q| = |N_q| - 1$. Note here, the complexity is $O(|N_q|^2)$, not $O(N_q)$, because one edge may be computed for several times, depending on how many paths are sharing the edge. This is also one of the disadvantageous aspects of the algorithm. In practice, we expect the filtering algorithm could be in $O(|N_q|)$, otherwise the filtering step may not be efficient enough. The reason of Lazy Algorithm being expensive lies in two aspects:

1. **Repeated computation.** See Fig. 7.6 for an example, for view $V_1$, the algorithm computes $I_0 \land I_{pc}(a, b) \land I_{pc}(b, e)$ twice, once for path $a/b/e/c$ and once for $a/b/e/f$. Here, $a/b/e$ is shared by the two paths.

2. **Useless computation.** Refer to the same example in Fig. 7.6, for view $V_2$, $Q$ can be filtered at an early stage. Because $pc(a, b)$ does not hold in $V_2$, whereas all paths in $Q$ contain edge $(a, b)$. As a result, we can draw the conclusion right away after examining $pc(a, b)$ in $V_2$.

We will remedy the two drawbacks in Eager Algorithm.
Eager Algorithm

We now introduce Eager Algorithm (Algorithm 7.3) in detail. After trimming query $Q$ as in Lazy Algorithm, we push a pair $(r_q, I_0)$ into a global queue, where $r_q$ is the root of query $Q$, $I_0$ is bit vector with length $|\text{paths}(V)|$ and every position set to “1”. In each iteration of the loop, pop a pair $(n_0, I_0)$ from the global queue and detect if $I_0 = 0$ (line 5-8). If yes, that means relationships on path from $r_q$ to $n_0$ cannot be fully satisfied in $V$, and thus $Q$ can be filtered out. Otherwise, if $n_0$ is a leaf node, that means path from $r_q$ to $n_0$ in $Q$ can be satisfied in $V$, then we go on to test other paths in $Q$, i.e. pop another pair from the global queue and repeat the iteration. If $n_0$ is not a leaf node, for each child $n_i$ of $n_0$, we obtain the next partial result from $r_q$ to $n_i$ by bit-ANDing $I_0$ with $I_{\text{pe}}(\text{label}(n_0), \text{label}(n_i))$ (or $I_{\text{ad}}(\text{label}(n_0), \text{label}(n_i))$), depending on the edge type of $(n_0, n_i)$. This new bit vector and node $n_i$ are afterwards pushed into the global queue as a pair to be tested in future.

Eager Algorithm runs in $O(|N_q|)$, and is in linear complexity. Edges in query $Q$ are visited only once. This is achieved by utilizing a queue to record the intermediate results. In contrast to Lazy Algorithm, Eager Algorithm will report
**Algorithm 7.3** Eager Algorithm for Filtering Equivalent Rewriting

**Input:** a query $Q$, a view $V$ and an index $I$ on $V$

**Output:** a boolean value (false: if impossible to answer)

1: Find a node $n$ on the distinguished path of $Q$, such that $\text{length}(r_v, d_v) = \text{length}(r_q, n)$, and trim $Q$ into $Q_{up}(n)$;

2: Initiate a queue $\text{globalQueue}$ and initiate a bit vector $I_0$ with length $|\text{paths}(Q_{up}(n))|$ and every position set to 1;

3: $\text{globalQueue}.\text{in}(r_q, I_0)$;

4: while $\neg \text{globalQueue}.\text{empty}()$ do

5:  $(n_0, I_0) := \text{globalQueue}.\text{out}()$;

6:  if $I_0 = 0$ then

7:     return false;

8:  end if

9:  if $n_0$ is not a leaf node then

10:    for each child node $n_i$ of $n_0$ in $Q$ do

11:       if $e = (n_0, n_i)$ is a pc-edge then

12:          $\text{globalQueue}.\text{in}(n_i, I_0 \land I_{pc}(\text{label}(n_0), \text{label}(n_i)))$;

13:       else

14:          $\text{globalQueue}.\text{in}(n_i, I_0 \land I_{ad}(\text{label}(n_0), \text{label}(n_i)))$;

15:     end if

16:    end for

17:  end if

18: end while

19: return true;
whether a query is unanswerable as early as possible. On the other hand, the space cost of Eager Algorithm is $O(|N_q|)$ (basically the size of the queue) worse than $O(1)$ of Lazy Algorithm. In real applications, $|N_q|$ is not very large, since an XPath tree pattern query does not contain a lot of nodes, so the larger space cost of Eager Algorithm is not a problem.

7.3.3 Contained Rewriting

For contained rewriting, we cannot do the trimming step as for equivalent rewriting, since Lemma 7.1 does not hold for contained rewriting. Fig. 7.7 shows an example. The reason is that finding a contained rewriting is based on finding a useful embedding, which is a one-direction partial matching. A useful embedding is different from a homomorphism in that, for a useful embedding $f$, each anchor node $n_a$ (if not a leaf node) possesses either of the following properties: (i) $n_a$ is mapped to $d_v$; (ii) $n_a$ connects its children with $//$. As a result, not every path in query $Q$ needs to be fully matched, and the unembedded part of each path can be further tested in the real (materialized) view. This brings challenges for testing the existence of contained rewritings for $Q$ using $V$. Fortunately, we are able to modify Eager Algorithm to filter $Q$ for contained rewriting, because useful embedding is upward closed and Eager Algorithm accordingly traverses the query tree in a top-down manner. Full algorithm is shown in Algorithm 7.4. We will not explain it detailedly, since it is similar to Algorithm 7.3. Here, line 9-11 and line 12-20 attempt to determine if $n_0$ is an anchor in $Q$ by checking the above two properties (i)(ii) respectively. And $i_d$ in line 9 and 16 means the path number of the distinguished path in $Q$. 
Algorithm 7.4 Algorithm for Filtering Contained Rewriting

**Input:** a query $Q$, a view $V$ and an index $I$ on $V$

**Output:** a boolean value (false: if impossible to answer)

1: Initiate a queue $globalQueue$ and initiate a bit vector $I_0$ with length $|paths(Q)|$ and every position set to 1;
2: $globalQueue.in(r_q, I_0)$;
3: while $¬globalQueue.empty()$ do
4:   $(n_0, I_0) := globalQueue.out();$
5:   if $I_0 = 0$ then
6:       return $false$;
7:   end if
8:   if $n_0$ is not a leaf node then
9:     if $label(n_0) = label(d_v)$ && $I_0[id] = 1$ then
10:        continue;
11:     else
12:        for each child node $n_i$ of $n_0$ in $Q$ do
13:           if $e = (n_0, n_i)$ is a pc-edge then
14:              $globalQueue.in(n_i, I_0 \land I_{pc}(label(n_0), label(n_i)));$
15:           else
16:              if $I_0[id] ≠ 1$ then
17:                 $globalQueue.in(n_i, I_0 \land I_{ad}(label(n_0), label(n_i)));$
18:              end if
19:          end if
20:        end for
21:     end if
22:   end if
23: end while
24: return $true$;
CHAPTER 7. FILTERING TECHNIQUES FOR FINDING REWRITINGS

We have discussed filtering techniques for subclass \( XP^{\{\text{/},\text{/}//\}} \). In this section, we extend our work to a more general subclass \( XP^{\{\text{/},\text{/}//,\ast\}} \), allowing \( \ast \) to appear in both the query and the view.

When we have wildcard nodes in view \( V \), the alphabet turns into \( \Sigma_v \cup \{\ast\} \). However, we choose not to consider \( \ast \) when building index \( I \) for \( V \), because the algorithms will introduce false negatives. Fig. 7.8 (a) gives an example. We cannot find a homomorphism or useful embedding from \( Q \) to \( V \), because the \( \ast \) node in \( Q \) cannot be mapped to the \( \ast \) node in \( V \). But actually, \( /a/\ast//c \) and \( /a/\ast//c \) are equivalent patterns. Rewritings do exist for \( Q \) using \( V \). Therefore, to avoid false negatives (corresponding the second property in Section 7.1), we ignore \( \ast \) nodes when testing structural relationships. Algorithm 7.1 can be reused for \( XP^{\{\text{/},\text{/}//,\ast\}} \), with a last step added to eliminate \( \ast \) entries in index \( I \).

For the query \( Q \), a wildcard node \( n_\ast \) can be eliminated as follows: connect each node in \( \text{Children}(n_\ast) \) to \( \text{Parent}(n_\ast) \) with \( \text{/} \) and then delete \( n_\ast \). By repeatedly performing this operation until there is no \( \ast \) node in \( Q \), we can remove all \( \ast \) nodes.
without losing relationship between all the other nodes. The result query $Q'$ is in $XP^{\{./, []\}}$. A simple example is given in Fig. 7.8 (b). Thereafter we are able to use the established method in Section 7.2 and Section 7.3 to detect if $Q'$ can be filtered. Since $Q'$ is a relaxed form of $Q$ ($Q$ is contained in $Q'$), if $Q'$ is filtered out, $Q$ will be filtered out as well. The filtering algorithms will not introduce false negatives.

### 7.5 Summary

In this chapter, we have developed some filtering algorithms for rewriting XPath queries using views. These algorithms are able to discover unanswerable queries efficiently without actually computing the rewritings using views. Filtering algorithms are studied for both types of rewritings, equivalent rewriting and contained rewriting. The basic idea is to verify whether the structural relationships in a query could be satisfied in a view, given that label preserving and structure preserving are the key conditions in discovering a homomorphism (for finding an equivalent rewriting) or a useful embedding (for finding a contained rewriting). We use index to capture the structural relationships in the view, and develop two
algorithms for equivalent rewriting, i.e. Lazy Algorithm and Eager Algorithm. Furthermore, Eager Algorithm can be modified to support contained rewriting. We also extend the work to allow XPath queries and views to have wildcards, i.e. in XPath subclass $XP^{(\ldots)}$, which is a representative subset of XPath queries including child axes, descendant axes, branches and wildcards.
Chapter 8

Thesis Conclusions

In this chapter, we summarize the major contributions of the thesis in Section 8.1, and propose a few interesting directions which may be further explored in future in Section 8.2.

8.1 Summary of this Thesis

The series of works in this thesis focus on rewriting and evaluating XPath queries using views, which is generally categorized as answering queries using views. We concentrate on XPath queries with child axes “/”, descendant axes “//”, predicates “[ ]” and also have discussed a number of cases involving wildcards “*”. This XPath subset, denoted as $XP^{/}//[\star]$, is a typical XPath subset, and is frequently used in real applications. We summarize our contributions in the following aspects.

Firstly, we have studied how to find the irredundant maximal contained rewriting when we rewrite an XPath query using an XPath view. Contained rewriting is proposed to cater for data integration scenario, where views are unlikely to be complete due to the limited coverage of data sources, and hence
equivalent rewritings are impossible to be found. As a result, it usually requires to find a maximal contained rewriting for a query to provide the best possible answers. A maximal contained rewriting is a set of contained rewritings, and may contain redundant contained rewritings. Obviously, evaluating redundant contained rewritings is unnecessary. To find all irredundant contained rewritings, we first prove a contained rewriting independence property to avoid checking containment between one contained rewriting and a union of contained rewritings, and hence the problem is reduced to pairwise containment check for all the produced contained rewritings. We then propose a set of methods to minimize the number of candidate contained rewritings in order to save the computation for pairwise comparisons. The minimizing methods include (1) representing a contained rewriting by a set of link nodes so that the same contained rewriting produced by different useful embeddings is recorded only once; (2) using a sufficient condition, CAT containment, to prune part of redundant contained rewritings; (3) developing three algorithms, pFirst, qFirst, Hybrid together with accelerating heuristics to detect CAT containment. We also compare our solution with the naive solution using experiments. The experiments demonstrate the efficiency of our methods.

Secondly, we have investigated how to evaluate the irredundant maximal contained rewriting on materialized views. In some applications, such as query caching, query optimization, to find the irredundant maximal contained rewriting is only an intermediate step, not the final goal. We need to utilize the cached query results or intentionally precomputed query results to quickly answer new queries from the users. After finding all irredundant contained rewritings, the problem is transformed into how to efficiently evaluate a set of CATs (of those irredundant rewritings) on the materialized views. Besides a naive algorithm,
which evaluates the CAT of each irredundant contained rewriting on the materialized view and combines the results, we propose a basic algorithm driven by the observation that one component pattern may be shared by a number of CAT patterns, and the component pattern should be evaluated only once for multiple CATs. We further propose four pruning rules and three heuristic rules to reduce the number of CATs needed to be evaluated. These pruning rules and heuristic rules can be applied on the fly during the computation. A set of experiments show that the basic algorithm with optimizing rules is the best choice. We also point out that our evaluation algorithms for a set of CATs does not rely on specific view materialization. Therefore, the algorithms are able to work seamlessly together with either subtree fragments as views or node lists as views, and can be integrated into different systems with no difficulty.

Third, we have proposed a set of transformation methods to eliminate wildcard nodes and “//” edges in a pattern according to the schema information. Wildcard nodes can be replaced by concrete labels from the schema by traversing the pattern twice (one bottom-up and one top-down) to instantiate the wildcard nodes. After that, we propose how to replace “//” edges with a subgraph from the schema graph so that “//” edges are replaced by a concrete path or a union of paths. We also modify the condition for detecting the existence of a rewriting for queries and views as the transformed patterns. Finally, recursive schema is considered and how to use recursive schema to confine queries and views is discussed. The idea is to remain one recursive loop in the pattern, and mark the loop start node and loop end node in the pattern. When finding rewriting for queries and views with loop, further conditions are added and need to be checked.

Finally, we have studied the filtering techniques for finding rewritings. In real applications, a large number of queries may be submitted to a system at one time. We need to figure out which queries can be answered with the materialized
views in storage and which queries have to be evaluated on the real database. We have studied the filtering function for both equivalent rewriting and contained rewriting. The basic idea is to verify whether the structural relationships in a query could be satisfied in a view, given that label preserving and structure preserving are the key conditions in discovering a homomorphism (for finding an equivalent rewriting) or a useful embedding (for finding a contained rewriting). We use index to capture the structural relationships in the view, and develop two algorithms for equivalent rewriting, i.e., Lazy Algorithm and Eager Algorithm. Moreover, Eager Algorithm can be modified to support contained rewriting. After studying all of the above for queries and views in subset $XP^{(/,//,[/],*)}$, we further discuss the problem for XPath subset $XP^{(/,//,[/],*)}$ including wildcards.

8.2 Future Work

We have discussed a series of works to rewrite and evaluate XPath queries using views, but related research never comes to an end. We now summarize the aspects the current studies have touched either in this thesis or in other related works, and point out several interesting future directions.

Given a query and a set of views, the current studies in the literature have covered: (1) how to find an equivalent rewriting or a set of irredundant contained rewritings for the query using a single view; (2) how to find an equivalent or a set of contained rewritings for the query using multiple views, such as view intersections or view joins; (3) when there is a schema, how to find an equivalent or a contained rewriting under schema constraints; (4) when there is a data summary as constraints, how to rewrite a query using data summary; (5) how to select relevant views to answer given queries; (6) how to implement view materialization facilitate the evaluation of a rewritten query on the materialized
views; (7) how to optimize the evaluation of multiple rewritten queries on the materialized views. The above aspects have mentioned almost all the important directions that need to be investigated. For future directions, we do not deny that there may be further improvement for the above works, on the other hand, it is interesting to study the following aspects.

The first aspect is the study on more generalized queries and views: In this thesis and most of the related works, the queries and views that have been studied belong to a subclass of XPath containing child axes “/”, descendant axes “//”, predicates “[]” and wildcards “*”. There are other axes such as parent, following, preceding, ancestor-or-self, following-sibling, preceding-sibling, etc. It is usually difficult to cover the whole XPath axis set, but for particular applications, it may be worthwhile to include other XPath axes. As far as we know, in the current studies on the topic, few works have touched other axes. Furthermore, a more popular XML query language XQuery may be more thoroughly studied. A possible solution is to model XQuery query as tree pattern queries, and use existing methods in the state-of-the-art works, because most of the state-of-the-art works assume the queries to be tree pattern queries. XQuery is after all more complex than XPath, for example, XQuery supports aggregate functions, but XPath does not, so there may be some differences when we deal with XQuery queries in the context of answering queries using views.

Another interesting aspect is to investigate the rewriting problem in the presence of complex XML schema, such as more complex recursive schema. The current discussion in Chapter 6 touches simple recursive schema, and does the reasoning by examples. More thorough discussion is needed for complicated cases, such as that two cycles in the schema may have intersection parts. These complicated cases are always difficult, and may result in chasing the pattern in $XP^{(\text{/}/\text{[]})}$ rather challenging.
The last, but not the least, direction is to explore materialized view maintenance. It is not directly related to answering queries using views. However, the quality of view maintenance affects the performance of the rewriting both on the accuracy of using a view and the effectiveness of using a view.
Bibliography


[TXg] The XMark group. XMark An XML Benchmark Project. In *http://www.xml-benchmark.org/*.


Appendix A

Author’s Publications


