Evaluating Irredundant Maximal Contained Rewritings for XPath Queries on Views

Rui Zhou\textsuperscript{1}, Chengfei Liu\textsuperscript{1}, Jianxin Li\textsuperscript{1}, Junhu Wang\textsuperscript{2} and Jeffrey Xu Yu\textsuperscript{3}

\textsuperscript{1}Swinburne University of Technology, Australia
\textsuperscript{2}Griffith University, Australia
\textsuperscript{3}Chinese University of Hong Kong, China

Email: \{rzhou, cliu, jianxinli\}@swin.edu.au; J.Wang@griffith.edu.au; yu@se.cuhk.edu.hk

We review the problem of finding contained rewritings for XPath queries using XPath views. Contained rewriting (CR) is proposed to cater for data integration scenarios, 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, we are usually required to find a maximal contained rewriting (MCR) for a query to provide the best possible answers. An MCR is a set of CRs, and may contain redundant CRs. Obviously, evaluating redundant CRs on materialized views is unnecessary. In this paper, we first address how to find the irredundant maximal contained rewriting (IMCR), i.e. all the irredundant CRs. We show that the existing approach ignores a type of situations, and turns out to be not sufficient. As a result, the only safe solution is a brute-force pairwise containment check for all the CRs. We then propose some heuristics to speed up the brute-force comparisons. When a materialized view is given, we propose how to evaluate the IMCR on the materialized view, which, to our knowledge, is the first work on optimizing the evaluation of a set of produced CRs on the materialized view by considering the inherent structural characteristics of the CRs. Our experiments show the effectiveness and efficiency of our algorithms.

Keywords: Algorithm; IMCR; Query Rewriting; View; XML

Received ; revised ; accepted

1. INTRODUCTION

Answering queries using views \cite{1} (also known as rewriting 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 functionality \cite{2}. It is a classic problem, and resides 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.

In the literature, fruitful research achievements have been made on rewriting XML queries using XML views \cite{3, 4, 5, 6, 7, 8, 9, 10}. All of these works except \cite{6, 8} focus on equivalent rewriting (ER), which means, given a view and a query, to evaluate the rewritten query (called compensation query in \cite{3, 5}) 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. While in \cite{6}, the authors focus on contained rewriting (CR). 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, the view itself may not be 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 CR does not refer to a compensation query, it is defined as a merged query of a compensation query and the view. Note however that, a MCR may contain redundant CRs, i.e. CRs that
are contained in other CRs. This means the answers produced by the redundant CRs can be covered by the answers produced by other CRs that are also belonging to the MCR set. Obviously, it is unnecessary to evaluate those redundant CRs on the materialized view. We call an MCR with no redundant CRs an IMCR (irredundant maximal contained rewriting).

Examples of CR, MCR, IMCR and compensation patterns are shown in Table 1, formal definitions will be given in Section 2. In the table, \( CR_1 \) and \( CR_2 \) are contained rewritings for query \( q_1 \) using view \( v \), with compensation patterns \( /b \) 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 \( q_2 \), 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 \( CR_3 \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>( q_1 )</th>
<th>( CR_1 )</th>
<th>( CR_2 )</th>
<th>( v )</th>
<th>( q_2 )</th>
<th>( CR_1 )</th>
<th>( CR_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( //a/b )</td>
<td>( /b )</td>
<td>( //a/b )</td>
<td>( a )</td>
<td>( //a/b )</td>
<td>( /b )</td>
<td>( //a/b )</td>
</tr>
</tbody>
</table>

The aim of the paper is, given a view \( v \) and a query \( q \), (a) to efficiently find the IMCR of \( q \) using \( v \) and (b) to efficiently evaluate the IMCR on the materialized view of \( v \) to obtain the query result.

As to problem (a), it has been mentioned in previous work [6], but is not thoroughly studied. The previous strategy is to find the IMCR by finding useful embeddings (a technique we will review in Section 2) from \( q \) to \( v \), 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 on produced embeddings) are all unlikely to be sufficient. Therefore, the only safe solution is probably a brute-force pairwise containment check for all the CRs. We then propose some heuristics to speed up this general approach.

As to problem (b), to our knowledge, it has not been well studied, especially on how to optimize the evaluation of multiple CRs. The intuitive solution is to evaluate each CR from the IMCR set on the materialized view individually, and then union the results of these CR queries. This approach is correct, but not efficient, because the number of CRs tends to be exponential in terms of the number of paths in \( q \). However, our observation is that these exponential number of CRs are composed by a linear number of sub-patterns bounded by \(|q|\), the size of the query \( q \). Therefore, taking advantage of the special structural characteristics of the irredundant CRs, we are able to develop very efficient algorithms.

Our paper is organized as follows:

- In Section 2, we will introduce background knowledge and notations including the concept of embedding and useful embedding, which are the main techniques to find contained rewritings. We will formally define CR, MCR and IMCR.
- In Section 3, we address how to find the IMCR. We give an example on which the previous strategy fails and we analyze the reason of the failure. We point out the only solution to find the IMCR is probably enumerating pairwise containment checks. We then propose a set of algorithms to speed up the checks.
- In Section 4, we look into the problem of evaluating an IMCR over a materialized view. The main idea is: observing the large number of CRs are produced by enumerating some component patterns, we are able to produce the same set of answers by only evaluating the component pattern. We develop a set of rules and heuristics to speed up the query processing.
- In Section 5, we conduct extensive experiments to test our algorithms. The results show that our pruning algorithms are very effective and efficient. When an IMCR needs to be evaluated on a materialized view, our heuristic algorithms perform very well.
- In Sections 6 and 7, related works and conclusions are given.

## 2. PRELIMINARIES

In this section, we introduce some background knowledge and notations. XML documents and XPath queries are modeled as trees and tree pattern queries, and evaluating an XPath query on an XML document is modeled as matching a tree pattern query to a tree. Then, query containment is formulated, and a homomorphism-based query containment test is introduced. Finally, we introduce useful embedding and formalize the concepts of CR, MCR and IMCR.

### 2.1. XML Trees and XPath Tree Pattern

Generally, an XML document is modeled as an unordered tree\(^4\) with nodes labeled 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 is the virtual document node, which has a single child that represents the root node.

\(^4\)Element order is ignored in this work.
element of the XML document. In other words, every XML tree starts with a virtual root node labeled $v^5$. We denote all possible trees over $\Sigma$ as $T_\Sigma$.

An XML document is a tree $t = (N_t, E_t, r_t)$, 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 $\text{label}(r_t) = r$;

XPath is the core subclass of XML query languages. We consider a subset of XPath featuring child axes (/), descendant axes (//) and branches ([]). It 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(l|/|/\).

An XPath query $q$ can be expressed as a tree pattern $(N_q, E_q, r_q, d_q)$, where

- $N_q$ is the node set, and for any node $n \in N_q$, $n$ has a label in an infinite alphabet $\Sigma$, denoted as $\text{label}(n)$;
- $E_q$ is the edge set, and $\forall e \in E_q, \text{type}(e) \in \{/|/|/\}$. We use the term “pc-edge” (“ad-edge”) to represent the type of an edge, “$|$” or “$/|$”;
- $r_q$ is the root node of the query; (If $q$ starts with “$|$” or “$/|$”, we add a virtual root node with a unique label $r$, so that every query corresponds to a unique tree pattern.)
- $d_q$ is the distinguished (also called return or answer) node of the query (identified with a circle in the figures throughout this paper).

A path in an XPath query $q$ is a set of nodes and edges residing from the root of the query $r_q$ to a particular leaf node of $q$. Obviously, the number of paths in $q$ equals to the number of leaf nodes in $q$. The paths containing the distinguished node are called distinguished paths.

An XPath view $v$ can be defined in a similar way as a 4-tuple, $(N_v, E_v, r_v, d_v)$. For ease of discussion, we introduce the following notions. Given a view $v = (N_v, E_v, r_v, d_v)$ and two nodes $n_1, n_2 \in N_v$: (i) $\text{pc}(n_1, n_2)$ holds in $v$ if $n_1$ is the parent node of $n_2$; (ii) $\text{ad}(n_1, n_2)$ holds in $v$ if $n_1$ is an ancestor node of $n_2$. Obviously, we have the following corollaries: (a) $(n_1, n_2)$ is a pc-edge. $\Leftrightarrow \text{pc}(n_1, n_2)$ is true. (b) $(n_1, n_2)$ is a ad-edge. $\Rightarrow \text{ad}(n_1, n_2)$ is true. (c) $\text{pc}(n_1, n_2)$ is true. $\Rightarrow \text{ad}(n_1, n_2)$ is true.

The result of evaluating an XPath query, equals to finding mappings from a tree pattern query $q$ to a tree $t$, which can be represented as $q(t) = \{f(d_q)|f$ is some mapping from $q$ to $t\}$. A mapping from a tree pattern $q = (N_q, E_q, r_q, d_q)$ to a tree $t = (N_t, E_t, r_t)$, $f : N_q \rightarrow N_t$, satisfies:

- Root preserving: $f(r_q) = r_t$;
- Label preserving: $\forall n \in N_q$, $\text{label}(n) = \text{label}(f(n))$;
- Structure preserving: $\forall (n_1, n_2) \in E_q$, if $(n_1, n_2)$ is a pc-edge, then $\text{pc}(f(n_1), f(n_2))$ holds in $t$; otherwise $\text{ad}(f(n_1), f(n_2))$ holds in $t$.

For any two tree pattern query $p$ and $q$, $p$ is said to be contained in $q$, denoted as $p \subseteq q$, if $\forall t \in T_\Sigma, p(t) \subseteq q(t)$. 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 $p \subseteq q$ (proved in [11]). 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 $f(d_q) = d_p$. 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 mappings from a pattern to a data tree.

### 2.2. XPath Query Rewriting Using View

We explain equivalent rewriting [5] and maximal contained rewriting [6] formally as follows: given an XPath query $q$, an XPath view $v$, and an XML database $t$ representing a large XML document, let the materialized view of $v$ on $t$ be $t_v = v(t)$, to find an equivalent rewriting for $q$ using $v$ is to find a compensation pattern $q_c$ such that $q_c(t_v) = q(t)$. Meanwhile, the pattern produced by merging the root node of $q_c$ with the answer node of $v$, denoted as $q_c \oplus v$, is called an equivalent rewriting of $q$. Fig. 1(a) shows an example. $v$ (/a/b) is a query, $q_1$ (/a/b/c) is a query, $q_c$ (b/c) is the compensation query, and $ER$ is the equivalent rewriting of $q_1$ using $v$. Different from equivalent rewriting, maximal contained rewriting usually requires to find a set of compensation patterns $Q_c = \{q_{c_1}, \ldots, q_{c_j}\}$ satisfying $Q_c(T_v) \subseteq q(T)$, where $Q_c(T_v) = q_1(T_v) \cup \ldots \cup q_{c_j}(T_v)$, and there does not exist another compensation pattern set $Q'_c$ such that $Q_c(T_v) \subset Q'_c(T_v) \subseteq q(T)$. Each $q_{c_j} \oplus v$ ($j \in [1, i]$) is called a contained rewriting $(CR)$ of $q$, and it can be produced by merging the root node of $q_{c_j}$ with the answer node of $v$, similar to constructing an equivalent rewriting. Fig. 1(b) shows an example for a maximal contained rewriting. $q_2$ and $v$ are the query and the view, $\{q_{c_1}, q_{c_2}, q_{c_3}, q_{c_4}\}$ are the compensation patterns, and $\{CR_1, CR_2, CR_3, CR_4\}$ are the CRs.

To rewrite 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, i.e. discover the compensation pattern. The solution is to find an embedding from the query to the view, and see whether the unembedded parts (or

---

5The purpose of adding a virtual root is to facilitate pattern match. For example, given an XPath pattern //a/b/, we need to add a virtual root to express the XPath query as a tree pattern. Accordingly we need a virtual root for an XML data tree so that the evaluation of an XPath query can be modelled as finding mappings from the XPath tree pattern to the XML tree.
unembedded conditions) can be further checked in the view extensions. If so, the embedding is valid to produce a contained rewriting. A valid embedding is called useful embedding. Embedding and useful embedding are defined below.

Given a query \( q = (N_q, E_q, r_q, d_q) \) and a view \( v = (N_v, E_v, r_v, d_v) \), 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: \( \forall n \in N'_q, e(n) \in N'_v \land \text{label}(n) = \text{label}(e(n)) \);
- structure preserving: \( \forall (n_1, n_2) \in E_q \land n_1, n_2 \in N'_q \), if \( (n_1, n_2) \) is a pc-edge, \( \text{pc}(e(n_1), e(n_2)) \) holds in \( v \); otherwise \( \text{ad}(e(n_1), e(n_2)) \) 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, \( n, n' \in N'_q \Rightarrow \forall n', n'' \in N_q \) and \( \text{ad}(n', n) \) holds in \( q \), 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 \( e \), \( N'_q \subseteq N'_v \). An embedding implies that part of the conditions of query \( q \) have been satisfied in the view \( v \), so if the remaining 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 \( q \). Such embedding is called a useful 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 \( p_{th_i} \) of \( q \), the last node on \( p_{th_i} \) that is embedded on or above \( d_v \) is called an anchor node of embedding \( e \) with respect to path \( p_{th_i} \). An embedding is called a useful embedding, if both of the following conditions hold:

1. every anchor node \( n_a \) satisfies either of (a), (b):
   - (a) \( e(n_a) = d_v \);
   - (b) \( \text{ad}(e(n_a), d_v) \) holds in \( v \) and let \( n_c \) be the child node of \( n_a \) on \( p_{th_i} \) in \( q \), \( (n_a, n_c) \) is an ad-edge.
2. for the anchor node \( n_a \) on the distinguished path \( q \), either \( n_a = d_q \) or \( \text{ad}(n_a, d_q) \) holds in \( q \).

FIGURE 1. Equivalent rewriting and Contained rewriting

FIGURE 2. An example of useful embedding

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 with "\(^{}/\)" on the corresponding path (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, \( e_1 \) is an embedding, but not a useful embedding, because its anchor node, the \( a \)-node, does not satisfy either condition (1)(a) or (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 \( p_{th_1} \), the anchor node \( b \)-node maps onto the distinguished node of the view \( v \) (satisfying condition (1)(a)); on path \( p_{th_2} \), 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 \( p_{th_2} \) with "\(^{}/\)" (satisfying condition (1)(b)); the anchor node \( b \)-node on the distinguished path \( p_{th_1} \) is above the distinguished node in the query \( q \) (satisfying condition (2)). As a result, \( e_2 \) is a useful embedding. The corresponding CR produced by \( e_2 \) is also given on the rightmost side of Fig. 2. In the CR, \( b[//d]/c \) is called a compensation pattern to the view. In the following, we will use clip-away tree (CAT) to represent compensation pattern in alignment with a pioneer work [6].

We define another concept, component pattern: given a useful embedding, let \( a_n \) be an anchor node in \( q \) with respect to some path and let \( a_n \) be the immediate successor of \( a_n \) on that path, we call such a constructed pattern a component pattern: the pattern has a root node with \( \text{label}(d_v) \), and the root node connects \( q_{ub}(a_c) \) with the same type of edge as edge \( (a_n, a_c) \), here \( q_{ub}(a_c) \) is the subpattern in \( q \) rooted at node \( a_c \). A component pattern may be a predicate component pattern (not containing node \( d_q \)) or a distinguished component pattern (containing node \( d_q \)). In the CR in
Fig. 2, \( b[//d] \) is a predicate component pattern, \( b/c \) is the distinguished component pattern. We will use \( p \) to denote a component pattern, and \( P = \{ p_1, \ldots, p_n \} \) to denote a CAT, since a CAT can be regarded as several component patterns fusing at the root. (A CAT may contain only one component pattern.)

3. FINDING THE IMCR

In this section, our target is to find the IMCR of a query using a view. 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, and then (2) do 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, even though we ignore the possible case that one CR can be contained in a union of some others, we still need to check query containment for an exponential number of CR pairs. Therefore, in the first step, 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 3.1, we will show how to minimize the candidate CR set, and in Section 3.2 we will provide a theoretical result that allows to efficiently eliminate the remaining redundant CRs.

3.1. Minimizing the candidate CR set

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

3.1.1. Representing a CR

Some redundant CRs are introduced into the candidate CR set, because different useful embeddings can produce the same contained rewriting. Fig. 3 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 \((CR_1 \text{ and } CR_2)\) of two different embeddings are the same.

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\(^6\) of \( n_a \) on the path is a link node w.r.t. embedding \( e \). In Fig. 3, 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 to see 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 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.

\(^6\)The child node must exist according to the anchor node definition.
To find the link node set of a useful embedding is not difficult. An existing algorithm [6] can be modified to serve the goal. It is not the focus of this paper, so we simply describe the amendments as follows: (1) We do not allow an anchor node to map onto a node below $d_a$, therefore we will take the first node on a particular path that cannot be mapped on or above $d_a$ as a link node. (2) The subtree rooted at $d_a$ does not need to be considered in the algorithm, because no nodes under $d_a$ can be an anchor node, since $d_a$ can never map onto a node above $d_a$. We can trim the subtree rooted at $d_a$ before the algorithm starts. When the algorithm finishes, every link node is leading a component pattern, e.g., $p_1, p_2$ in Fig. 4. The complexity of constructing component patterns is $O(|q|v^2)$. The upper bound of the number of component pattern is $\min\{|N_q| - 1, kl\}$. (We will give a detailed explanation of the upper bound in Section 4.1.) In the following, we will show how to eliminate part of redundant CRs using CATs and component patterns.

3.1.2. A sufficient condition to eliminate part of 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. 5. $CR_1$ is redundant and is contained in $CR_2$, 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 that redundant CRs are produced because we did not embed as many nodes as possible. However, it is indeed component pattern containment introduced 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 [6]. 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. 6 gives an example. $CR_1$ and $CR_2$ are two CRs for rewriting $q$ using $v$. Here, component patterns $p_1$ and $p_2$ are on different paths, and $p_2$ is contained in $p_1$ (explained later). 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 $p_2$. (Using Condition 1 in Lemma 1 in [6], $CR_2$ is wrongly identified as irredundant. Interested readers can refer to the work [6] to verify the result.)

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 [11]. 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.

In the above examples in Fig. 5 and 6, one common observation is that CR containment can be attributed to CAT containment. In Fig. 5, each CAT is a single component pattern, and the result is obvious. While in Fig. 6, 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 3.1.

**Lemma 3.1.** 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 $P \subseteq Q$. 

---

**The Computer Journal, Vol. ??, No. ??, ???**
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 \( CAT_2 \not\subseteq CAT_1 \). 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.

**Lemma 3.2.** 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.** We will prove the lemma by using homomorphism technique. Recall Theorem 3 in [11], 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, therefore 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 \( p_i \subseteq q_j \). \( \square \)

### 3.1.3. Algorithms to eliminate part of redundant CRs
Following Lemma 3.2, 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 \( p_i \subseteq p_j (p_i \subseteq p_j \wedge p_j \subseteq p_i) \). Each entry of \( M \) is filled only once when necessary. For example, given four CATs \( P_1 = \{p_1, p_2\}, P_2 = \{p_1, p_3\}, P_3 = \{p_4, p_5\}, P_4 = \{p_3, p_4\} \), to check containment relationship \( P_1 \subseteq P_3, p_1 \subseteq p_2 \) needs to be checked once. (Let us assume \( p_2 \not\subseteq p_1 \).) 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 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 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 \( \emptyset \), 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 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 \( P \).
Algorithm 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: \hspace{1em} retrieve $c(p_i)$;
4: \hspace{1em} $S := S - c(p_i)$;
5: for each $s_j$ in $S$ such that $M[r(p_i)][r(s_j)]$ is not defined do
6: \hspace{2em} if $p_i \subseteq s_j$ then
7: \hspace{3em} $M[r(p_i)][r(s_j)] := 1$ and $S := S - \{s_j\}$;
8: \hspace{2em} else
9: \hspace{3em} $M[r(p_i)][r(s_j)] := 0$;
10: \hspace{1em} if $S = \emptyset$ then
11: \hspace{2em} return true;
12: \hspace{1em} return false;

Algorithm 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: \hspace{1em} retrieve $\tau(q_j)$;
3: \hspace{1em} if $\tau(q_j) \cap P = \emptyset$ then
4: \hspace{2em} temResult := false;
5: \hspace{1em} for each $p_i \in P$ such that $M[r(p_i)][r(q_j)]$ is not defined do
6: \hspace{3em} if $p_i \subseteq q_j$ then
7: \hspace{4em} $M[r(p_i)][r(q_j)] := 1$ and temResult := true;
8: \hspace{4em} break;
9: \hspace{3em} else
10: \hspace{4em} $M[r(p_i)][r(q_j)] := 0$;
11: \hspace{1em} if temResult = false then
12: \hspace{2em} return false;
13: \hspace{2em} return true;

• 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 1 stop earlier, and also benefit Algorithm 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 2 stop earlier.

Observe Algorithm 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 2 does the opposite. Therefore, if $P$ is indeed contained in $Q$, Algorithm 1 may stop earlier, and hence perform better than Algorithm 2. If $P \not\subseteq Q$, Algorithm 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 3, where overlapping parts with Algorithm 1 and 2 are described with natural language to save space. We compare the three algorithms in Section 5.

An important atomic operation in these algorithms is to check containment between two component patterns, see line 6 in both Algorithm 1 and 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, we have a more efficient way to find whether $p$ is contained in $q$. We introduce a property as Lemma 3.3 below.

**Lemma 3.3.** 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 3.3 is not difficult to prove. With Lemma 3.3, we can check component pattern containment on the same path easily. Here, $ad(n_p, n_q)$ and whether (parent($n_p$), $n_p$) is an ad-edge can be checked in constant time if we have an index built for query $Q$. In the implementation, we used region encoding [12] to encode the nodes in $Q$. We can check whether two nodes $n_p$ and $n_q$ satisfy $ad(n_p, n_q)$ by checking whether their encoded regions intersect. Region codes were generated when query pattern and view pattern were traversed to find component patterns. Alternatively, other encoding methods, such as dewey encoding [13], can be used. By making annotations on the edges, we can check whether

---

The Computer Journal, Vol. ??, No. ??, ???
3.2. Further eliminating redundant CRs

As is mentioned in the previous section, CAT containment is a sufficient but not necessary condition to find redundant CRs. Therefore, although the candidate CR set is reduced, we still need a strategy to identify those real irredundant ones. The challenge is that one CR may be contained by a union of others, making the computation difficult.

In the previous study [6], irredundant CR is defined as “not contained in any other CR”. This definition only needs pairwise checks between the CRs, which made the computation simple. However, if a union of CRs can contain some CR, according to commone sense, the contained CR should be regarded as redundant as well. The question is: given CR1, CR2 and CR3, is it possible that CR1 \( \not\subseteq \) CR2 and CR1 \( \not\subseteq \) CR3, but CR1 \( \subseteq \) CR2 \( \lor \) CR3? Fortunately, within query subset \( XP(////) \), 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 (Theorem 3.1). As a result, the previous definition happens to be equivalent to ours. We now highlight the property as a theorem and prove it.

**Theorem 3.1.** For an XPath pattern \( p \) and a union of XPath patterns \( Q = \{q_1, q_2, \ldots, q_n\} \) in \( XP(////) \), we have: \( p \subseteq Q \), if and only if \( \exists q_i \in Q, \text{ such that } p \subseteq q_i \).

**Proof.** 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 [11], containment relationship for XPath patterns remains the same for their boolean pattern counterparts. For two boolean patterns \( p \) and \( q \) in \( XP(////) \), \( p \subseteq q \) if and only if there exists a homomorphism from \( N_p \) to \( N_q \). In other words, if \( p \not\subseteq q \), there must exist a node \( n_q \) in \( N_q \), such that we cannot find any homomorphism \( h \) that has a corresponding node \( h(n_q) \) in \( N_p \), satisfying label preserving and structure preserving conditions w.r.t. nodes \( n_q \) and \( h(n_q) \). We call such node \( n_q \) 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 3.1, 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 \( Q(t) = q_1(t) \lor q_2(t) \lor \cdots \lor q_n(t) \), 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.e. \( z \in \Sigma - \bigcup_{i=1}^{n} \Sigma_q \)), where \( \Sigma_q \) is the alphabet of \( q_i \).

Since \( \Sigma \) is infinite (when there is no schema available) and \( \Sigma_q \) is finite (because the number of labels in a query is limited), this transformation is always possible. After the transformation, it is straightforward to see 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 \( n_q \) in \( q_i \), such that for the transitional node, we cannot find a corresponding node \( f(n_q) \) 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 \( h(n_q) \) in \( p \) would exist for the homomorphism. This result contradicts with the assumption that \( n_q \) 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 \( p(t) \Rightarrow Q(t) \). The contrapositive statement of the necessary condition is proved.

Theorem 3.1 holds for all XPath patterns in \( XP(////) \). Since the CRs are XPath patterns generated by some special operations, Theorem 3.1 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 in the candidate CR set to eliminate the redundant ones.

Note that, Theorem 3.1 holds when \( p \) and \( Q \) have an infinite alphabet. If the alphabet is finite, Theorem 3.1 does not hold, because ‘//’ can be unfolded by enumerating all the possibilities. For example, assume the alphabet is \( \{a, b\} \), let \( p = /a/b \), \( Q = \{q_1, q_2, q_3\} \) where \( q_1 = /a//b \), \( q_2 = /a//b \) and \( q_3 = /a//b \), \( p \) is contained in the union of \( q_1, q_2 \) and \( q_3 \), but \( p \) is not contained in any single pattern of \( Q \).

4. EVALUATING THE IMCR ON MATERIALIZED VIEWS

In some applications, such as query caching, query optimization, to find the IMCR is only an intermediate step, not the final goal. We need to utilize the cached query results or intentionally precomputed query results, i.e. materialized views, to quickly answer new queries from the users. We will target the problem in this section.

4.1. Basic Algorithm

We regard a materialized XPath view as a set of subtrees, whose roots have the same label as the distinguished node of the view, obtained by matching the view pattern to the base document. To evaluate a CAT on these trees, we can equally match the CAT
pattern on each tree and combine the results. For simplicity, we only consider one tree of the materialized view in this paper, for computation on other trees are the same. After obtaining the CATs using useful embedding, the problem turns to be: given a set of CAT queries (representing the compensation patterns) and a materialized data tree $T_v$, how to efficiently evaluate the CAT queries over the 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 one by one on the materialized view will result in repeated computation of the same component patterns. Although the number of CATs is exponential, the number of component patterns is polynomial. We now explain the bound for the number of component patterns, because $\min\{|N_q| - 1, kl\}$ for the number of component patterns: (1) Notice that component patterns share the same labeled root node, and distinguish each other starting from the single child of the root, e.g. in Fig. 2, component patterns $b/\emptyset/d$ and $b/c$ share the same b-node, and distinguish each other at the d-node and c-node. This observation implies that each node in $N_q$ (the node set of the query) except $r_q$ determines one possible component pattern, and there are at most $|N_q| - 1$ component patterns. Meanwhile, $kl$ 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 $q$. In consequence, the maximum number of component patterns $\min\{|N_q| - 1, kl\}$ could be far less than the maximum number of CATs $k^l$.

The idea of our basic algorithm, shown in Algorithm 4, is to break down the CATs into component patterns, and evaluate the component patterns. As a result, each component pattern is examined only once. We now introduce the basic algorithm, and then develop some optimization techniques on the basic algorithm in Section 4.2. We first evaluate all the predicate component patterns on the materialized view $T_v$, and then, for each CAT $P$, if all the predicate component patterns of $P$ are satisfied on $T_v$, we evaluate the distinguished component pattern of $P$ on $T_v$ and add the result into the final result set $R$. 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.

4.2. Optimizing Techniques

In the Basic Algorithm, 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 them. In the following, when we say component pattern $p$ is satisfied on $T_v$, we mean: $p(T_v) = true$ if $p$ is a predicate component pattern; or $p(T_v) \neq \emptyset$ if $p$ is a distinguished component pattern.

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 1. In Fig. 7, 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 $P_3$ and CAT $P_2$.

A CAT can be regarded as a conjunction of its

---

Algorithm 4 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: $R := \emptyset$;  
4: for each CAT $P$ such that $\forall p_i \in P, p_i$ is predicate component pattern and $p_i(T_v) = true$ do  
5: evaluate the distinguished component pattern $\hat{p} \in P$;  
6: $R := R \cup \hat{p}(T_v)$;  
7: return $R$;

**FIGURE 7.** CATs of the IMCR
component patterns, and each component pattern is a condition. Any unsatisfied condition will prevent the CAT from producing answers, no matter whether other component patterns are satisfied or not. Pruning Rule 2 is based on the following proposition:

**Proposition 4.1.** Let $\text{CAT}_1$ and $\text{CAT}_2$ be two CATs having the same distinguished component pattern, then, for any data tree $T$, either $\text{CAT}_1(T) = \text{CAT}_2(T)$ or one of $\text{CAT}_1(T)$ and $\text{CAT}_2(T)$ is empty.

**Proof.** The result of evaluating a CAT on a tree $T$ is the same as the result of evaluating the distinguished component pattern of the CAT on $T$, when all predicate component patterns of the CAT are satisfied on $T$, because the predicate component patterns only serve as conditions imposed on $T$. Therefore, if another CAT $Q$ has the same distinguished component pattern as CAT $P$, then no matter what predicate component patterns $P$ and $Q$ possess, if both $P$ and $Q$ have answers on $T$, their answer sets will be the same. □

**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 2.** In Fig. 7, if CAT $P_1 = \{p_1, p_3\}$ has been evaluated (implying $p_3(T_v) = \text{true}$), CAT $P_2 = \{p_1, p_4, p_5\}$ can be discarded, since both CATs share the same distinguished component pattern $p_1$, no matter whether $p_1(T_v)$ is empty or not.

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. Again, we use $c(p)$ to denote the set of component patterns that contain $p$, and $\pi(p)$ to denote 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 3.** In Fig. 7, 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 $\pi(p)$ cannot be satisfied on the view.

**Example 4.** In Fig. 7, if $p_4$ is not satisfied, we know none of $p_1$, $p_2$ and $p_3$ could be satisfied, then all CATs $P_1$, $P_2$, $P_3$, $P_4$ do not have answers on the materialized view.

All of the four rules introduced above try every means to prune some patterns, and hence we can evaluate as fewer component patterns as possible. 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, but it is unlikely for us to find this order without knowing in advance whether a component pattern is satisfied in the materialized view or not. To this end, we design some heuristics to find a reasonably good evaluation order.

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 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. 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 the pruning rules. Heuristic 1 is more akin to 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 discarded. Heuristic 2 implies an eager strategy to find some answers as early as possible by inspecting component patterns in the same CAT. It is akin to 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, other CATs in the same group will be freed of examination. We want to stress that the heuristics are orthogonal to the pruning rules. Any heuristic to determine an order of evaluating the component patterns can be integrated into the algorithm shown in Algorithm 5.
We now go through Algorithm 5 step by step. In the beginning, the result set $R$ is set to $\emptyset$. 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, 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 $\land$ all the predicate component patterns of a CAT containing $p$ as the distinguished component pattern are satisfied then

$$R := R \cup p(T_v);$$

else

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

use Rule 1 to prune other CATs;

return $R$;

4.3. Discussion

In the above process, the IMCR is generated first, and then for each CAT of an irredundant CR, evaluate it on the materialized view. Since any MCR will produce the same set of answers as the IMCR, we do not require the input to be strict CATs from the IMCR. Sometimes, users do not care whether it is the IMCR or a MCR being evaluated, as long as the answers returned is correct. Therefore, if the materialized view is not large, which means to evaluate a few redundant CRs on the view may not be more expensive than to check the redundancy of all the CRs, we can simply inject a MCR into the system to evaluate. The algorithms also work with CATs from a MCR. When evaluating a MCR, We can 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 $c(p)$, $\mathbf{7}(p)$ can be added after line 2 in Algorithm 5.

5. EXPERIMENTS

We build a prototype system IMCRE (IMCR Evaluator) to find the IMCR for a query given a view. IMCRE also supports to evaluate a generated MCR 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 [10]), 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 $\alpha_1$; (2) the edge connecting the child to itself is labeled as // with probability $\alpha_2$; (3) a descendant node is selected with probability $\alpha_3$ directly connecting to its parent; (4) the maximum fanout $f$ is fixed and set within a limit. We do not generate value predicates in the pattern, because checking value predicates can be easily integrated into the system.

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 therefore the fanout of the result pattern will be increased.

Datasets We test our algorithms on two datasets, XMark [14] and BIOML [15]. The former is widely used in the literature, and the latter is famous for its recursive feature. The BIOML DTD is tailored with only “chromosome” and its descendant elements. The datasets are generated with IBM XML Generator [16]. We use the eXist [17] database as the underlying engine to store and query the documents. Each materialized
view is generated by evaluating the view pattern on the documents and saving back into the eXist database.

5.1. 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 refers to the total number of CRs found by useful embeddings, PCR refers to the leftover CRs after performing CAT containment pruning, ICR refers to 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 by varying parameters $\beta_1$ or $\beta_2$. In Fig. 8, 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 that, on each path, there will be more component patterns, because any node will produce a component pattern as long as it is leading an ad-edge and also can be mapped onto the distinguished path between $r_v$ and $d_v$ of the view. The 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. 9. The number of CRs is increasing, because the 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 decreasing, the number of paths is increasing. As a result, the total number of CRs is still increasing. While at $\beta_1 = 0.8$, the decrease is due to the 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.

5.2. 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 then 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 1), qFirst (Algorithm 2), Hybrid (Algorithm 3). Experiments in Fig. 10 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 1 and Algorithm 2 the size of $Q$ is usually smaller than the size of $P$. Hybrid is superior to any other algorithm, which confirms our assumption.

We also 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.

5.3. Performance on Evaluating the IMCR on Materialized Views

Average Case Study In this study, we test the performance of the four different algorithms. 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. The size of view extensions ranges from 15k to 532k for XMark data, and 9k to 864k for BIOML data. In the NAIVE algorithm, each CAT of an MCR is evaluated on the materialized views. In BASIC algorithm, only component patterns are evaluated on the materialized views, but all the component patterns are evaluated. In the optimized algorithms, we use the proposed four rules to prune unnecessary component patterns, and we
also use heuristic information to schedule the evaluation order. Specifically speaking, in HEU1, we use the first two heuristics with heuristic 2 going prior to heuristic 1. In HEU2, we first group the CATs by their distinguished component patterns (heuristic 3), and then apply heuristic 2 and heuristic 1.

The result is shown in Fig. 11, the 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. This demonstrates that 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 suffers in updating component pattern statistics for each distinguished pattern group, and therefore it may spare some time on maintaining the auxiliary information.

**Best and Worst 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 variously as possible. It is reasonable that our heuristic algorithms perform better in the average case. One may wonder how far our heuristic algorithms can achieve and what is the worst performance our heuristic algorithms will reach. We examine the best and the 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 patterns. The query time is shown in Fig. 12. On the BIOML dataset, the BASIC algorithm beats the NAIVE in two-thirds time, and the HEU1 needs only 10% query time of the BASIC algorithm. Similar observation is obtained on the XMark dataset, but the result is not that dramatic.

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. 13, the HEU1 algorithm performs almost the same as the BASIC, because both of them have evaluated all the component patterns. The part of updating heuristic statistics in the HEU1 algorithm does not apparently degrade, though it may take some extra time. Both the BASIC and the HEU1 are a little costly than the NAIVE. The reason may be that the NAIVE algorithm has evaluated less number of patterns, although each pattern has a larger size.

### 6. RELATED WORK

Answering queries using views has been extensively studied for a long time. Halevy [1] has done a survey 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 have been developed as well, e.g. MiniCon [18], bucket [19], inverse-rules [20, 21], to tackle the problem in the relational context.

In the XML context, two types of rewritings, equivalent rewriting and contained rewriting, have been studied for XPath [3, 4, 5, 7], XQuery [9, 10], and tree patterns [6, 8]. The works [9, 3, 4] propose to use materialized views to speed up query evaluation in the query caching scenario, where to find an equivalent rewriting for a query with given views is a key subtask. Heuristic approaches are used in these works. Xu and Ozsoyoglu [5] have provided a theoretical study on equivalent rewriting based on query containment [11] and query minimization [22]. It has shown that finding an equivalent rewriting for queries in $XP[1,*,//]$ including branches, wild cards and descendant axes, is coNP-hard. And for three
subclasses (combining any two of the three features), \( XP[[i],\ast ] \), \( XP[[i],\ast ] \) and \( XP[[i],\ast ] \), the problem is in PTIME. These conclusions are based on the previous study of XPath containment in [11]. Afrati et al. [7] have extended Xu and Ozsoyoglu’s result in fragment \( XP[[i],\ast ] \). They have discovered a coNP-complete upper bound for some sub-fragments of \( XP[[i],\ast ] \), and also show to find an equivalent rewriting for \( XP[[i],\ast ] \) is decidable in the general case. Onose et al. [10] 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 [10]. Tang et al. [23] studied the materialized view selection problem, which is to select fewer materialized views to answer a query equivalently. Wu et al. [24] and Chen et al [25] proposed two different view implementations, and how to use the materialized views to evaluate queries. View definition nodes are also materialized in [24, 25], while our work does not have this requirement. Lakshmanan et al. [6] addressed how to find contained rewritings, but did not cover how to efficiently evaluate the compensation patterns over materialized views. A recent work [26] discussed how to perform the evaluation. It is based on an approach to examine each path pattern and then verify the rewriting, and each path pattern needs to be built into an automata. While, in our approach, we use useful embedding to find the compensation patterns, and try to evaluate as few component patterns as possible.

The work [27] also addressed a similar problem: that is how to use views to answer tree pattern queries. However, the focus points and techniques are different. Firstly, the work [27] aimed to find answers without finding contained rewritings. Their idea is to annotate view answers and then evaluate queries against the annotated view answers. They propose not to find contained rewritings. On the contrary, our work aims to find contained rewritings first, and then evaluate contained rewritings on genuine view answers. Secondly, as a result of the first difference, the work [27] discussed how to eliminate redundant views. While in our work, since we find contained rewritings first, we discuss how to eliminate redundant rewritings. To summarize, the work [27] focused on processing the views, but we focus on processing the queries.

A few works consider constraints to rewrite queries [6, 8, 28]. In [6], schema information are considered as five types of constraints. The view is first chased by the constraints implied within the schema, and then the rewriting can be discovered by finding useful embeddings from the query to the chased view. In [8], the constraints is expressed as structural summary, precisely speaking, Dataguide [29] constraints. In [28], views are defined by annotating a DTD, not simply XPath views. Queries are rewritten into regular XPath queries that are evaluated on the source. XML query rewriting using multiple views are studied in the works [30, 31]. The work [32] studies query rewriting using query set specifications, which can express families of queries. There are works on rewriting queries using view based on RDF data [33, 34].

7. CONCLUSIONS

In this paper, we have investigated the problem of rewriting XPath queries using XPath views, especially on evaluating the irredudant maximal contained rewriting (IMCR). The problem consists of two parts: finding all the irredudant CRs and evaluating them on materialized views. Compared to the previous works, we have given a more precise definition to redundant CR and pointed out the insufficiency of the existing method on finding irredudant 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 irredudant ones. Our approach is proved to be correct. When the irredudant 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 irredudant 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.

ACKNOWLEDGEMENTS

Rui Zhou, Chengfei Liu, Jianxin Li are supported by the grant of Australian Research Council Discovery Project No. DP110102407 and DP120102627. Junhu Wang is supported by the grant of Australian Research Council Discovery Project No. DP1093404. Jeffrey Xu Yu is supported by the grant of Research Grants Council of Hong Kong SAR, No. 418512. We also would like to thank anonymous reviewers for their helpful comments.

REFERENCES


Evaluating Irredundant Maximal Contained Rewritings for XPath Queries on Views


