Minimization of Tree Pattern Queries

Sihem Amer-Yahia
AT&T Labs–Research
sihem@research.att.com

SungRan Cho
Stevens Institute of Technology
scho@attila.stevens-tech.edu

Laks V. S. Lakshmanan
Concordia University & IIT Bombay
laks@it.iitb.ernet.in

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

Tree patterns form a natural basis to query tree-structured data such as XML and LDAP. Since the efficiency of tree pattern matching against a tree-structured database depends on the size of the pattern, it is essential to identify and eliminate redundant nodes in the pattern and do so as quickly as possible. In this paper, we study tree pattern minimization both in the absence and in the presence of integrity constraints (ICs) on the underlying tree-structured database.

When no ICs are considered, we call the process of minimizing a tree pattern, constraint-independent minimization. We develop a polynomial time algorithm called CIM for this purpose. CIM's efficiency stems from two key properties: (i) a node cannot be redundant unless its children are, and (ii) the order of elimination of redundant nodes is immaterial. When ICs are considered for minimization, we refer to it as constraint-dependent minimization. For tree-structured databases, required child/descendant and type co-occurrence ICs are very natural. Under such ICs, we show that the minimal equivalent query is unique. We show the surprising result that the algorithm obtained by first augmenting the tree pattern using ICs, and then applying CIM, always finds the unique minimal equivalent query; we refer to this algorithm as ACIM. While ACIM is also polynomial time, it can be expensive in practice because of its inherent non-locality. We then present a fast algorithm, CDM, that identifies and eliminates local redundancies due to ICs, based on propagating "information labels" up the tree pattern. CDM can be applied prior to ACIM for improving the minimization efficiency. We complement our analytical results with an experimental study that shows the effectiveness of our tree pattern minimization techniques.

1. INTRODUCTION

Spurred by the popularity of XML and LDAP directories, which employ a core tree-structured model for representing and manipulating data, there is currently a resurgence of interest in tree data models. Not surprisingly, queries in such data models tend to be expressed in the form of tree shaped *patterns*. The idea is one finds all ways of "embedding" the pattern into the database, with the answer set constructed from the set of all embeddings found. For example, "*find all books with an editor and an author*" and "*find projects under departments whose name has 'networking' appearing in it and such that the departments fall under research*" are examples of queries naturally represented as tree shaped patterns.

Answering such queries requires matching a tree pattern against a tree-structured database. Since the efficiency of tree pattern matching depends on the size of the pattern, it is essential to identify and eliminate redundant nodes in the pattern and do so as efficiently as possible. A tree query may fail to be minimal for one of two reasons.

First, there may be inherent redundant "components" in the query like in classical conjunctive queries. As an example, consider the query "*find departments that contain a database project and that contain project managers managing a database project*". Intuitively, the requirement on the first database project is "subsumed" by the one on the second and can be dropped, leading to an equivalent minimal query. We call this *constraint independent minimization* (CIM) since minimization is achieved in the absence of any integrity constraints (ICs) on the database. Recall that for classical conjunctive queries, containment and minimization are both NP-complete problems [5]. Thus, our first major challenge is finding ways of minimizing tree queries efficiently in the absence of any ICs.

Just like any database, tree databases naturally come with application dependent ICs. For tree databases, constraints that require entries/elements to have child/descendant entries/subelements of specified types, as well as constraints that require type co-occurrences, are very natural. The second reason a query fails to be minimal is that even if it is irredundant, it may become redundant in the presence of ICs. For example, consider the query "*find the title and author of books that have a publisher*". If the IC "*every book has a publisher*" is known to hold, this query can be simplified to "*find the title and author of books*". The minimization we just performed is *constraint dependent minimization* (CDM) as it depends on the knowledge of ICs that hold for a database. Query minimization under ICs is traditionally achieved using semantic query optimization techniques [2]. Existing techniques for semantic query optimization base themselves on the notion of a residue using

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

which one rewrites a query into an equivalent query. While semantic query optimization can add or delete a subgoal (node and edge for us), for tree pattern minimization only deletion is relevant. Unfortunately, given a set of ICs, there are exponentially many ways in which a query can be rewritten (while removing subgoals). So a direct approach based on semantic query optimization is inappropriate. Thus, our second major challenge is finding ways of minimizing tree queries efficiently in the presence of ICs.

1.1 Contributions and Overview

In this paper, we address these two challenges and make the following contributions:

- We show that when no ICs are present, every tree pattern query has a unique equivalent minimal query. We develop a polynomial time algorithm, CIM, based on containment mappings, for obtaining the minimal equivalent query that takes worst-case time $O(n^4)$ in the query size (Section 4).

  CIM's polynomiality stems from two key properties: (i) a node cannot be redundant unless its children are, and (ii) the order of elimination of redundant nodes is immaterial.

- When ICs are restricted to required children, required descendants, and required co-occurrences, we show that the equivalent minimal query is unique (Section 5).

  In this case, we show the surprising result that augmenting a query with redundant nodes and edges in accordance with given ICs and then applying the CIM algorithm to the augmented query always produces the (global) minimal equivalent query (Section 5). This algorithm, which we call ACIM, is also polynomial, and takes worst-case time $O(n^6)$ in the query size.

- To mitigate the expense of ACIM, we develop an efficient algorithm called CDM that labels each tree pattern node with an *information content*, and propagates it up the tree pattern (Section 5). This algorithm produces a locally minimal equivalent query for a given query, in time $O(n^2)$ in the query size. The value of CDM comes from the fact that applying it as a prefilter before ACIM yields the minimal equivalent query, but much faster than ACIM applied alone.

- To investigate the effectiveness of the techniques proposed in this paper, we implemented our techniques using hash indices to ensure efficiency, and then performed a series of experiments. Our study demonstrates both the practicality of CDM and ACIM for realistic queries, and quantifies the speed-up obtained by first reducing a tree pattern using CDM before applying ACIM (Section 6).

Section 1.2 presents the related work. Section 2 contains the background material. Formal statements of the problems studied in this paper are given in Section 3. Sections 4 and 5 contain the core of our work (CIM, ACIM and CDM). Implementation details and experiments are presented in Section 6.

1.2 Related Work

The problem of minimizing a query with or without ICs is a fundamental problem in query optimization, and much work has been done on this topic for various data models. Tree pattern queries are essentially specialized conjunctive queries on a tree-structured domain. Containment and minimization of relational conjunctive queries are known to be NP-complete [5]. Saraiya [18] shows 2-containment, the restriction of the containment problem where there are at most two occurrences of any predicate in the query, can be solved in polynomial time, while 3-containment is NP-complete. Kolaitis et al. [11] show that containment for conjunctive queries with disequalities ($\neq$) is Π_2^p-complete and is co-NP-complete for 2-containment. Kolaitis and Vardi [12] establish a fundamental connection between conjunctive query containment and constraint satisfaction in AI. Florescu et al. [8] show that containment of conjunctive queries with regular path expressions over semistructured data is decidable, using some form of the chase technique. Under some restrictions they show that the problem is NP-complete. Chan [4] characterized containment and equivalence of conjunctive queries for OODBs and provided a minimization algorithm for a restricted class, called terminal conjunctive queries. His formulation focuses on type inheritance in OODBs and as such his results are not directly applicable to the problems studied in this paper. Levy and Suciu [13] show that equivalence and weak equivalence for conjunctive queries for OODBs are decidable and show that equivalence with grouping and aggregates is NP-complete. Millstein et al. [15] study the problem of containment relative to available data sources in the context of data integration and show that it is decidable.

Semantic query optimization has a long history and we only mention a few papers. Chakravarthy et al. [2] proposed a technique based on the notion of a residue of an IC against a query for optimizing non-recursive relational queries, Calvanese et al. [1] consider the problem of conjunctive query containment in an abstract setting that covers relational and OO models, under a class of special inclusion dependencies over complex expressions. They establish decidability results for this problem when the query has no regular expressions or no number restrictions and show the problem is undecidable when disequalities are allowed.

Techniques like predicate elimination and join minimization are used in cost-based optimization. This kind of optimization is based on algebraic rewritings which generate exponential search spaces. While such algebraic techniques can also be used in our setting, the search space of equivalent algebraic expressions remains exponential. None of the works that are about query optimization for XML such as [14] and [6] consider IC-based optimization. A logic-based approach is adopted in [17] where queries and constraints are represented in first-order logic. Chasing and backchasing are used to rewrite queries considering ICs. Because of the high complexity of these rewritings, [16] uses a stratification technique to reduce the search space during query optimization time.

2. BACKGROUND

2.1 Data Model and Queries

We consider a data model where information is represented as a forest of trees. Each node has an associated type. Depending on the application, node order in trees may be important. Two main applications inspiring this model are XML and LDAP-style network directories [9]. In XML,

data is modeled as a forest of ordered trees, each node corresponding to an element and the edges representing element-subelement relationships. For LDAP directories, sibling order is not important, and the edges may represent inherent hierarchies such as organizational or geographical.

Queries in languages such as XML-QL [7] and Quilt [3] are based on a notion of a *tree pattern* using which they extract relevant portions of XML data. Directory queries ask for entries that stand in specified structural relationships (parent, descendant, etc.) to other specified entries and are naturally represented as trees [10]. We express queries as tree patterns where nodes are types and edges are child/descendant relationships. We do not consider order in our queries. Section 3 contains concrete examples of queries.

2.2 Integrity Constraints

For tree structured databases, ICs that deal with the tree structure are very natural. Consider the example XML Schema specification shown in Figure 1(a). From this specification, we can infer that every `Book` element *must* have an immediate (i.e., child) subelement of type `Title`. We can infer that every `Book` element must have a `LastName` descendant subelement, if the schema specification says that every `Author` element must have a `LastName` child. More generally, whenever type `B` appears (as a subelement) in every XML Schema specification for type `A`, we can conclude every element of type `A` must have a child of type `B`, and hence a descendant of type `B`. In addition, suppose every specification for type `A` contains an element type C_i in it such that type C_i is known to require a (descendant) subtype `B`, we conclude type `A` must have a descendant subtype `B`. Inferring required parent and ancestor constraints is more involved, and requires examining all occurrences of a type in the schema.

Consider a directory database maintaining organizational white pages information. In this context, it is natural to expect that every `department` entry must have some `manager` entry below it, and that every `employee` entry must also belong to the type `person`.

The preceding examples illustrate the naturality and utility of integrity constraints corresponding to required children/descendants and required co-occurrences of types. Figure 1(b) shows the notation for these kinds of ICs.

3. PROBLEMS STUDIED

In this section, we formally state the problems studied in this paper and illustrate the issues involved in solving them, using realistic examples. The examples are depicted in Figure 2. Each query is shown as a tree with two kinds of edges: single edges (called *child edges*) represent direct containment (or immediate subelement) relationship between the parent and the child; double edges (called *descendant edges*) represent transitive containment relationship between the two nodes in question. Descendants are defined via transitive closure of the child relation, as usual. By way of terminology, whenever there is a child or descendant edge (u, v) in the tree pattern query, we say that v is a child of u. We stress this terminology should not be confused with the kind of edges, which are viewed purely syntactically for the purpose of this definition. When it is necessary to distinguish between different kinds of children of a node in a query, we speak of c-child and d-child with their obvious meanings. In each query, one node is marked with a "*". For directory applications, this is interpreted to mean only entries corresponding to the marked node are returned as the answer set. For example, in Figure 2(h), only `OrgUnit` entries would be included in the answer set. For XML applications, the "*" is interpreted to mean that the subtree rooted at the marked node is returned as part of the answer set. For example, in Figure 2(a), whole `Article` elements would be returned.

3.1 Constraint Independent Minimization

Given a tree query, we would like to minimize it. Let the size of a tree query stand for the number of nodes in it. A query Q_1 is contained in a query Q_2, $Q_1 \subseteq Q_2$, provided for every tree database $\mathcal{D}$, $Q_1(\mathcal{D}) \subseteq Q_2(\mathcal{D})$. Equivalence is two-way containment. A query Q is *minimal* if there is no equivalent query which has a proper subset of nodes of Q. Then, our minimization problem is as follows.

> **Problem** P_1: Given a tree pattern query Q, find an equivalent query of the smallest size.

As an illustration, consider Figure 2(h). This query asks for all entries of type `OrgUnit` (organizational unit) that immediately contain a department immediately containing a researcher who manages a database project, as well as a department descendant that contains some database project. Since the two query branches can map to the same database branch, it can be seen that this query is equivalent to the one shown in Figure 2(i). However, if Figure 2(h) were modified to put the "*" on the `Dept` node in the right branch, the queries in Figures 2(h) and 2(i) would not be equivalent.

Recall that classical conjunctive query containment and minimization are NP-complete. One source of complexity in this case is repeated occurrences of the same relation in the query. It is worth noting this source is also present in tree query containment in the form of repeated occurrences of the same type in a query. So it is not obvious how we can solve this problem in polynomial time.

3.2 Minimization Under Constraints

The class of ICs we are interested in were described in Section 2. Given a set of constraints $\mathcal{C}$ taken from this class, we say a query Q_1 is contained in a query Q_2, $Q_1 \subseteq_{\mathcal{C}} Q_2$, provided for every tree database $\mathcal{D}$ that satisfies the constraints $\mathcal{C}$, $Q_1(\mathcal{D}) \subseteq Q_2(\mathcal{D})$. Again, equivalence under ICs is defined as two-way containment under ICs. Minimality under ICs is defined in the obvious way. The second problem we study in this paper is as follows.

> **Problem** P_2: Given a tree pattern query Q and a set of constraints $\mathcal{C}$, find a query that is equivalent to Q under $\mathcal{C}$ and is of the smallest size among all such queries.

3.3 Illustrative Examples

We now discuss a few examples that illustrate constraint dependent minimization as well as the complex ways in which it can interact with constraint independent minimization. As a first illustration, consider the query in Figure 2(f). It asks for all organizations that have an employee managing a project and that have a permanent employee managing a database project. If we know that `PermEmp` must co-occur with `Employee` and that `DBproject` must co-occur with `Project`, we can see that the "`Organization--Employee`

```
<xsd:complexType name="Book">
  <xsd:element name="Title" type="xsd:String"/>
  <xsd:element name="Author" type="xsd:String"
               minOccurs="1" maxOccurs="5"/>
  <xsd:element name="Chapter" type="xsd:Chapter"/>
               ...
</xsd:complexType>
```

(a) Example XML Schema Specification

- $\tau_1 \longrightarrow \boxed{\tau_2}$: every type τ_1 node must have a child of type τ_2.
- $\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$: every type τ_1 node must have a descendant of type τ_2.
- $\tau_1 - \boxed{\tau_2}$: every type τ_1 node must also be of type τ_2.

(b) Notation for some ICs

Figure 1: Example XML Schema Specification and Notation for some ICs

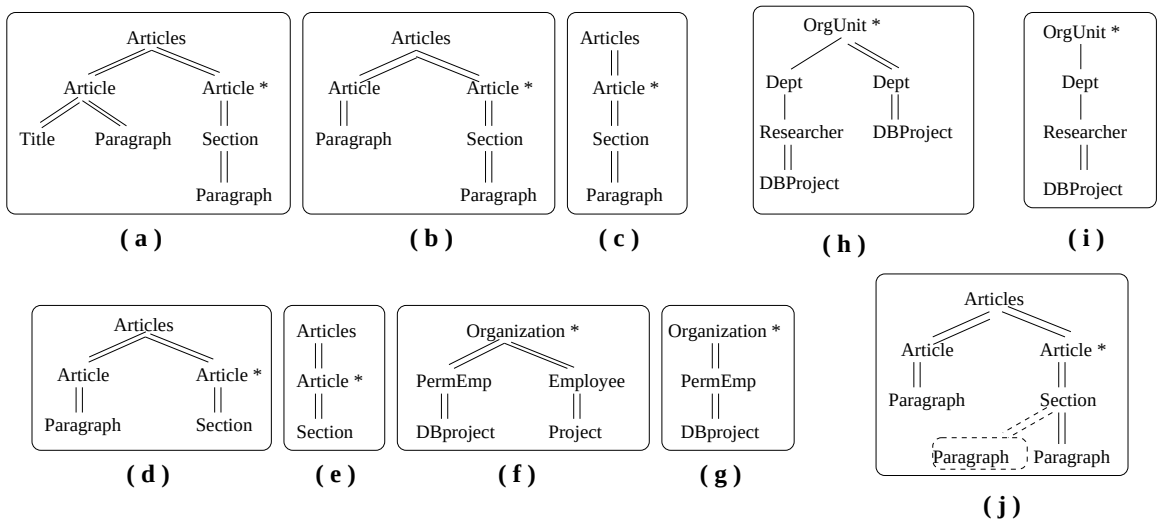


Figure 2: Examples of Tree Pattern Queries

`--Project`" path is redundant and can be eliminated to obtain the smaller equivalent query of Figure 2(g). The latter query cannot be reduced further and is thus minimal.

As a second example, consider the query of Figure 2(a). In the absence of ICs, it cannot be minimized further. If we know the IC `Article` $\longrightarrow\!\!\!\!\!\rightarrow$ `Title`, then the `Title` node becomes redundant and the query can be simplified to that in Figure 2(b). This query is *not* minimal: it can be further simplified to Figure 2(c), which *is* minimal. We did not use any IC in the latter step: it is akin to an application of constraint independent minimization.

Next, consider the query of Figure 2(b). If we know that the IC `Section` $\longrightarrow\!\!\!\!\!\rightarrow$ `Paragraph` holds, this query can be simplified to Figure 2(d). The latter query cannot be simplified further, either by applying this IC, or by using constraint independent means. However, it is *not* minimal: indeed, it can be shown that the query of Figure 2(e) is equivalent to this query and it contains a proper subset of the nodes and edges of this query. One way to obtain this query is to first apply a constraint independent minimization to Figure 2(b) to obtain Figure 2(c) and then use the above IC to minimize it further to Figure 2(e). This example shows the order in which we apply constraint independent and constraint dependent minimizations may be important in obtaining the minimal equivalent query.

As a last example, consider the query of Figure 2(d). In the absence of ICs, this query is minimal. If the only IC we are given is `Section` $\longrightarrow\!\!\!\!\!\rightarrow$ `Paragraph`, we cannot directly perform any minimization. Yet, we can *augment* this query by adding an extra `Paragraph` node to it and making it a descendant of `Section`, yielding an equivalent query. From this, we can perform constraint independent minimization to reason that the "`Articles--Article--Paragraph`" path is subsumed by the path "`Articles--Article--Section--Paragraph`", yielding Figure 2(c), which can be further minimized to Figure 2(e).

To summarize, in the absence of any ICs, we can make use of the classical technique based on containment mappings. But the source of high complexity for containment in the classical case is still present for our situation, and mere application of containment mappings will *not* yield an efficient algorithm. When ICs of the forms: required children, descendants, and co-occurrences are known to hold, we can use them to eliminate nodes from a given tree query. However, this step interacts with constraint independent minimization in subtle ways: (i) the two steps may need to be applied in some sequence; (ii) the final outcome may depend on the order of this application; and (iii) in certain cases, we need to temporarily augment the query to be able to perform the requisite minimization. Finally, when ICs are present or absent, it is not even clear that a query of the least size that is equivalent to a given query is unique.

4. CONSTRAINT INDEPENDENT MINIMIZATION

We wish to minimize a tree query when no ICs are known. The first question is whether there is necessarily a unique minimal equivalent query. The main tool we employ is that of containment mappings, which are essentially query homomorphisms. Adapted to tree queries, a containment map-

ping from a query Q_2 to a query Q_1 is a mapping $h : Q_2 \rightarrow Q_1$ from Q_2's nodes to Q_1's nodes such that: (i) h preserves node types: $\forall u$, u and $h(u)$ have the same type, and $h(u)$ has the "*" label iff u has the "*" label; (ii) h preserves structural relationships: whenever v is a c-child (resp., d-child) of u in Q_2, $h(v)$ is a c-child (resp., descendant) of $h(u)$ in Q_1. One can prove that, like in the homomorphism theorem of Chandra and Merlin [5], for two tree queries Q_1, Q_2, in the presence of sufficiently many node types, the query Q_1 is contained in Q_2 iff there is a containment mapping $h : Q_2 \rightarrow Q_1$. Note that while a child edge must be mapped to a child edge, a descendant edge may be mapped to any chain of child and descendant edges.

Let Q be a tree query and Q' a tree query which contains a subset of the nodes in Q. Then a homomorphism from Q to Q' is actually a homomorphism from Q to itself. Such a homomorphism is called an *endomorphism*. A natural way to approach minimization is to identify redundant nodes and eliminate them. A node v of Q is *redundant* provided the query obtained by deleting v (and possibly other nodes) is equivalent to Q. Notice that a node marked "*" can never be redundant. We first have:

PROPOSITION 4.1. *Let Q be a tree query. A node v of Q is redundant iff there is an endomorphism on Q that is not identity on v.* ∎

For a leaf v of a tree query Q, denote by $Q - \{v\}$, the tree obtained by deleting the leaf v and the incident edge from Q. Given a tree query Q, we define an *elimination ordering* as a sequence of nodes of Q $(v_1, \ldots, v_i, \ldots, v_k)$ such that v_i is a redundant leaf in $Q - \{v_1, \ldots, v_{i-1}\}$. Such a sequence is a *maximal elimination ordering* (MEO) if $Q - \{v_1, \ldots, v_k\}$ does not contain any redundant leaf. Our next result shows that any query obtained via an MEO cannot be further simplified through such node elimination.

LEMMA 4.1. *Let Q be a tree query and Q' be a query obtained from Q via an MEO $(v_1, \ldots, v_k)$. Then Q' is a minimal query equivalent to Q.* ∎

Note that the above lemma does *not* tell us that an equivalent query of the least size can always be obtained via a MEO, since it is possible there are other minimal equivalent queries not obtainable via an MEO. The next lemma settles that issue.

LEMMA 4.2. *Let Q be a tree query and Q' be an equivalent query containing a proper subset of the nodes of Q. Then there is an elimination ordering such that Q' can be obtained from Q via that elimination ordering.* ∎

It follows from the above lemma that any minimal equivalent query for a given query can be obtained via an MEO. Thus, for the purpose of finding an equivalent query of the least size, it suffices to consider only the equivalent queries obtainable via MEOs. But can different MEOs result in minimal equivalent queries of different size? The next lemma answers that question.

LEMMA 4.3. *Let Q be a tree query and let Q' and Q'' be any two minimal equivalent queries obtained via two different MEOs. Then Q' and Q'' are isomorphic.* ∎

Combining the above lemmas, we get the following result.

THEOREM 4.1. *Let Q be any tree query. Then it has a minimal equivalent query which is unique up to isomorphism.* ∎

This theorem only says that by following an arbitrary MEO, we can obtain a minimal equivalent query (which is the unique query with the least size). It does not necessarily imply this can be achieved in polynomial time. Notice that testing whether a leaf v is redundant involves testing whether it can be potentially mapped to some other node by some endomorphism. This is not a local test since the ancestors of this leaf as well as their descendants must be consistently mapped. In the following, we develop an efficient technique for testing whether there *exists* such an endomorphism that is not identity on the leaf v.

Suppose we wish to test whether a specific leaf v of Q is redundant. We begin by associating the set $images(u)$ with each node of Q: for the leaf v, $images(v)$ is set to the set of leaves of Q, other than v itself, such that their type is identical to that of v; for every other node u, set $images(u)$ to be the set of nodes of Q whose type coincides with that of u. We prune these sets in one bottom-up sweep as follows. Mark all leaves. Whenever there is an internal node p, all of whose children are marked, prune a node x from the image set $images(p)$ whenever one of the following holds: (i) p has a c-child c, but none of the nodes in $images(c)$ is a c-child of x, or (ii) p has a d-child c, but none of the nodes in $images(c)$ is a descendant of x. After checking for the prune-ability of all nodes in the image set $images(p)$, p is marked as well. We repeat this iteratively bottom-up. At the end, the set $images(r)$ associated with the root r may or may not be empty after the pruning. We can show the following result.

THEOREM 4.2. *Let v be a leaf of a tree query Q. Then v is redundant iff at the end of the pruning procedure the set $images(r)$ associated with the root r is non-empty.* ∎

The above theorem immediately yields a polynomial time algorithm for testing whether a leaf of a query is redundant. Let maxImage be the maximum size of the $images(u)$ set for any node u. The initialization of the initial $images(u)$ sets for all nodes of Q takes $O(n \times \text{maxImage})$, where n is the size of Q. Then for each edge of Q, we incur time proportional to maxImage^2 during the bottom-up sweep during which the sets are pruned. The final check for emptiness of $images(r)$ takes $O(1)$ time. Thus, checking redundancy of a given leaf can be done in $O(n \times \text{maxImage}^2)$ time. Since the redundancy of a leaf may have to be repeatedly checked, a naive implementation results in an MEO in $O(n^3 \times \text{maxImage}^2)$ time. The performance of this algorithm can be improved with a more efficient implementation, as given in Figure 3.

The key enhancements of this algorithm compared with the naive implementation outlined above are the following. (1) Once a node is identified to be non-redundant, it never need be tried again for redundancy since it can never become redundant. Thus, we invoke the redundancy check routine (not shown in Figure 3) only a linear number of times as opposed to quadratic. (2) It is unnecessary to prune the $images(p)$ sets for arbitrary unmarked nodes in the pruning phase. We only need to consider ancestors of the leaf v for pruning, and prune their children if appropriate. In the worst case, the complexity of the pruning phase can still be proportional to the number of edges of Q, but in practice, we expect the performance to be much better. (3) Whenever

```
Algorithm redundant-leaf;
Input: a tree query Q, and a leaf v of Q;
Output: YES if v is redundant, NO if v is
        not redundant;

Method:
1. images(v) = {x | x is a node of Q, x ≠ v,
       x and v are of the same type };
2. for each node u of Q: u ≠ v {
   images(u) = {x | x is a node of Q, x and
          u are of the same type };
   }
3. mark leaf v;
4. for (p = parent(v);; p = parent(p)) {
   4.1. minimize-images(p);
   4.2. if (images(p) = ∅) return (NO);
   4.3. if (p = images(p)) return (YES);
   /* once p is the root of the tree,
          one of 4.2 or 4.3 will be true */
   }

Algorithm minimize-images;
Input: a tree query Q, and a node p of Q;
Action: Prunes the images(u) sets for all
        descendant nodes u of p;

Method:
1. if (p is a leaf) mark p and return;
2. for each child c of p
        if (unmarked(c)) minimize-images(c);
3. for each x ∈ images(p) {
        for each child c of p {
            if (∄d ∈ images(c) : d is a descendant of x)
                {images(p) = images(p) − {x}};
        }
   }
4. mark p and return;
```

Figure 3: An Efficient Implementation of CIM

the set $images(p)$ for a node becomes empty, it stops: in this case, v cannot be redundant. It also stops, whenever $images(p) = \{p\}$ for some internal node. In the latter case, let c be the child of p that is an ancestor of v. We can show that the entire subtree of Q rooted at c is redundant. Sort the nodes in this subtree in any topological sort. This gives rise to a partial elimination ordering. We can then proceed with further redundancy detection. The correctness of the above algorithm follows from the theory developed earlier. Its time complexity is $O(n^2 \times \text{maxImage}^2)$. Since maxImage can itself be $O(n)$ in the worst case, the constraint independent minimization algorithm is $O(n^4)$.

5. MINIMIZATION UNDER CONSTRAINTS

Suppose we are given a query Q and a set of ICs $\mathcal{C}$. The first question is whether there is a unique equivalent query of the least size. We will show in this section that when only required child, descendant and co-occurrence constraints are considered, Q always has a unique minimal equivalent query. In addition to answering the first question, we tackle the issue of how to obtain the minimal equivalent query, and do so efficiently. For relational queries, there is a classical technique called *chase* which can be used to rewrite the query by adding the effects of the given ICs. Redundancies that were not (syntactically) visible before may become visible after the chase and an application of the homomorphism technique can be used to subsequently eliminate redundancies. The question is whether a similar technique would work for minimizing tree queries as well. We begin addressing this issue first. Uniqueness of the minimal equivalent query will be shown later.

5.1 Chase Reviewed

Let us first review the chase technique [19], adapted to tree queries. Let Q be a tree query and $\mathcal{C}$ a set of required child, descendant and co-occurrence constraints. Then, $chase_{\mathcal{C}}(Q)$, the *chase* of Q w.r.t. $\mathcal{C}$, is defined as follows:

- whenever $\mathcal{C}$ contains an IC of the form $\tau_i \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_j}$ and Q contains a node u of type τ_i, add a node of type τ_j and make it a d-child of u; a similar action is performed for constraints of the form $\tau_i \longrightarrow \boxed{\tau_j}$;
- whenever Q contains a node u of type τ_i and $\mathcal{C}$ contains the co-occurrence constraint $\tau_i - \boxed{\tau_j}$, also associate type τ_j with node u.

The first question is can we apply the CIM algorithm developed in the previous section to $chase_{\mathcal{C}}(Q)$ and obtain a minimal equivalent query. The following example illustrates the difficulty involved. Consider the query given in Figure 2(b). Suppose the IC `Section` $\longrightarrow\!\!\!\!\!\rightarrow$ `Paragraph` is known to hold. Chasing the query with this IC will add a second d-child of type `Paragraph` to the `Section` node. Applying the MEO-based technique of the previous section, we see that the left branch of the tree as well as *one* of the two d-children of the `Section` node will be eliminated. The final query obtained is the one in Figure 2(c), which is *not* minimal, since the `Paragraph` leaf can be eliminated, leading to Figure 2(e). This example shows that a direct application of chase followed by CIM will in general not yield a minimal equivalent query. A second issue with this approach is that a blind application of chase can make the result of the chase arbitrarily bigger than the original query: in particular, its depth can increase arbitrarily under chase.

5.2 Augmentation

In the following, we develop a technique for obtaining a minimal query equivalent to a given query under ICs. We also address the concern of query tree size blowup through the chase. Let Q be a query and $\mathcal{C}$ a set of ICs consisting of required child, descendant, and co-occurrence constraints. We make three major changes to the chase technique described above. First, we assume that $\mathcal{C}$ is a logically closed set of ICs. The closure can be obtained in a straightforward way, and has size at most quadratic in the size of the original ICs; details are omitted here for reasons of space.

The second change to chase is that we only apply ICs involving node types that existed prior to the chase, and to nodes that existed prior to the chase. In particular, if u is a node that was added by the chase, we do not apply any ICs to it. Similarly, if $\tau_i \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_j}$ is an IC, and u is a node of type τ_i, but there is no node of type τ_j in the original query, then we do not apply this IC to the chase.

Finally, note that nodes/edges added by the chase are redundant and so should be eventually removed. To facilitate this, the third change to chase is that we mark all such nodes and edges as temporary. Call the modified chase procedure *augmentation*.

We advocate the following procedure, called *Algorithm ACIM*, for minimizing a query under ICs: (i) Augment the query w.r.t. the closure of the given set of ICs; (ii) Apply the CIM algorithm of the previous section, ensuring that temporary nodes inserted by the augmentation phase are *not* checked for redundancy; and (iii) remove all temporary nodes added by step (i). Before proving the optimality of this algorithm, let us first illustrate it.

Consider the query of Figure 2(b), together with the IC `Section` $\longrightarrow\!\!\!\!\rightarrow$ `Paragraph`. This is already logically closed. Augmenting the query with this IC leads to the query of Figure 2(j), where nodes/edges added by augmentation are distinguished using dotted lines or boxes. Next, applying an MEO-based minimization leads to the left branch and the (unboxed) `Paragraph` node to be eliminated. Finally, the remaining dotted `Paragraph` node can be eliminated since it is temporary, yielding the query of Figure 2(e). In this example, this is indeed the minimal equivalent query.

5.3 Optimality of ACIM

We develop some notions and notation to help prove the optimality of this technique. Given a query Q and a set of ICs $\mathcal{C}$, we can eliminate a leaf u of type τ of Q, provided its parent v is of type τ' and $\mathcal{C}$ contains or implies the IC $\tau' \longrightarrow\!\!\!\!\rightarrow \boxed{\tau}$ or $\tau' \longrightarrow \boxed{\tau}$, depending on the type of the edge (v, u). By *reduction*, we mean a repeated application of this step until no longer possible. Note that reduction preserves equivalence under ICs, and always eliminates a descendant before eliminating its ancestors in a query. Let us call an application of the MEO-based algorithm to a query *minimization*. In the sequel, we denote the three major steps augmentation, reduction, and minimization by the letters A, R, and M. Since both R and M prune nodes and edges from a query tree, an equivalent query of the least size must be obtainable via some sequence of applications of the steps R and M. We need to determine a definite sequence of steps and show it leads an equivalent query of the least size. Since augmentation can facilitate the identification of redundant nodes/edges, we also include A in the language of strategies.

Various strategies for optimizing a query thus correspond to strings over the alphabet $\{A, M, R\}$, with possible repetitions. Our first result is that doing augmentation first does not prevent us from any minimization we might be able to do on the original query.

LEMMA 5.1. *Let Q be a query, $\mathcal{C}$ a logically closed set of ICs, and let $Q' = A(Q)$, i.e., the query obtained by applying augmentation to Q w.r.t. $\mathcal{C}$. Suppose $\mu : Q \rightarrow Q$ is an endomorphism on Q. Then there is an endomorphism $\mu' : Q' \rightarrow Q'$ on Q' such that μ' is an extension of μ.* ∎

Since μ' is an extension of μ, μ' will not be identity on any node of Q on which μ is not, and thus every redundant node of Q is also redundant in Q'.

Our next lemma establishes certain basic identities about the extent of minimization achieved by strings in the alphabet $\{A, R, M\}$. Recall that each string represents an algorithm. We use the notation $\alpha \sqsubseteq \beta$ to mean for any query Q, the result $\alpha(Q)$ of applying α to it contains every node and edge that is present in the result $\beta(Q)$ of applying β to Q. In this case, we say that α is dominated by β.

LEMMA 5.2. *The following identities hold, where α is any string over $\{A, M, R\}$: (i) $\alpha \sqsubseteq \alpha M$; (ii) $\alpha \sqsubseteq \alpha R$; (iii) $MR \sqsubseteq AMR$; and (iv) $RM \sqsubseteq AMR$.* ∎

Our next lemma shows that the strategy represented by the string AMR is idempotent.

LEMMA 5.3. *AMR is idempotent, i.e., for any query Q, $AMRAMR(Q) = AMR(Q)$.* ∎

Our last lemma shows that AMR is an optimal strategy in that no other string over $\{A, M, R\}$ is superior to it.

LEMMA 5.4. *Let ϕ be an arbitrary string over $\{A, M, R\}$. Then $\phi \sqsubseteq AMR$. In words, for any query Q, AMR produces the equivalent query of the least size among all equivalent queries.* ∎

Lemma 5.4 says two things. First, it says AMR is the best strategy among all strategies composed of A, M, R applied in any order, any number of times. Second, since every equivalent query of size smaller than Q can be obtained this way, it says the minimal equivalent query is unique. How does this result affect Algorithm ACIM developed earlier? Recall that ACIM does not exactly correspond to a string over $\{A, M, R\}$, since it uses a notion of temporary nodes and never considers them for redundancy checking, although it eliminates them in the end. Actually, ACIM is nothing but a clever implementation of AMR! To see this, consider any node deleted by R in $AM(Q)$. If this node corresponds to a temporary node inserted by A, there is nothing to show. Suppose it is a node u of Q. In this case, it is straightforward to see that the result of augmentation will make u redundant w.r.t. containment mappings. So, again this node will be eliminated during the minimization step. We finally have:

THEOREM 5.1. *Let Q be a tree query and let $\mathcal{C}$ be a set of ICs consisting of required child and required descendant constraints. Then there is a unique query Q' which is equivalent to Q and is minimal. Furthermore, Algorithm ACIM will always produce this minimal query.* ∎

Algorithm ACIM is illustrated in Section 3.3. We next analyze its complexity. We know that Algorithm CIM takes $O(n^2 \times \text{maxImage}^2)$, where n is the number of nodes in the query that is input to CIM. Since we perform an augmentation, the number of nodes could be much larger than in the original query Q. However, augmentation does not add new types to the query and increases the query tree pattern depth by at most one. Hence, the size of the augmented query can be at most $O(n^2)$. This can increase the maximum size of *images* to $O(n^2)$ as well. However, the temporary nodes added by augmentation are themselves never considered for removal during the minimization phase of CIM. As a result, the (worst-case) complexity of ACIM is $O(n^6)$.

Note that, in ACIM, we do not take advantage of opportunities to prune away nodes that are redundant in the presence of ICs. Can we remove all such redundant nodes *before* ACIM is applied? How quickly can we remove redundant nodes? These questions are addressed next.

5.4 Local Pruning

While Algorithm ACIM always yields the minimal equivalent query under ICs, and is in polynomial time, in practice, the size of the augmented query can be substantially larger than the original query leading to a performance that may sometimes not be acceptable. We ask whether there is any way we can improve the performance. In particular, can we

quickly identify all query tree nodes that are redundant under ICs, and eliminate them before feeding it to ACIM? In this section, we develop precisely such an algorithm, called *CDM*. CDM can act as an efficient pre-filter before ACIM is applied. We also show that CDM followed by ACIM always leads to the minimal equivalent query. The gain in the overall efficiency of constraint-dependent minimization under this approach comes from the fact that all locally redundant nodes are quickly eliminated and, as with ACIM, the temporary redundant nodes added by augmentation are themselves never considered for redundancy checking and are all eliminated in one shot at the end.

The basic idea behind CDM is the following. Iteratively identify any redundant leaves of the query tree and eliminate them until no longer possible. Suppose Q is a query and $\mathcal{C}$ is a logically closed set of ICs. There are four ways in which a leaf v can be found to be redundant:[1] (i) leaf v of type τ' is a c-child of node u of type τ and $\mathcal{C}$ contains the IC $\tau \longrightarrow \boxed{\tau'}$; or (ii) leaf v of type τ' is a d-child of node u of type τ and $\mathcal{C}$ contains the IC $\tau \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau'}$; or (iii) leaf v of type τ' is a c-child of node u, node u has another c-child of type τ, and $\mathcal{C}$ contains the IC τ — $\boxed{\tau'}$; or (iv) leaf v of type τ' is a d-child of node u, which has a descendant w [2] of type τ, and $\mathcal{C}$ contains one of the ICs $\tau \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau'}$ or τ — $\boxed{\tau'}$. Say that a leaf of a query is *locally redundant* precisely when one of the above conditions holds. While this procedure is incomplete in that it may not yield the minimal equivalent query, it has the advantage of being efficient and having the local minimality property.

The rules above by themselves do not yield an efficient test, since they need information that is not available at a node or its neighbors (see rule (iv)). Algorithm CDM is essentially an efficient implementation of the above procedure, by way of maintaining an "information content" at each node. The information content at a node is all the information relevant to applying ICs that will help detect whether its children are redundant. As a preview, in Figure 2(b), the essential information at the left `Paragraph` leaf is that it is of type `Paragraph`, while at its parent, the essential information is not only the parent's type but also the fact that it is constrained (by the query) to be an ancestor of a `Paragraph` node. If we know that the IC `Section` $\longrightarrow\!\!\!\!\!\rightarrow$ `Paragraph` holds, then at the `Section` parent, based on its information content, we can deduce that its `Paragraph` child is redundant.

We use the following notation for information content. The information content at any node is composed of one or more *information arguments*, which can be one of the following, where τ, τ_i denote types as usual. The information argument τ at a node means it is of type τ, without being constrained by any descendants. In contrast, $\tilde{\tau}$ means the node is associated with type τ, but it is constrained by the presence of descendants. In addition, we also denote structural obligations of a node in the form $a\tau, a\tilde{\tau}, p\tau$, and $p\tilde{\tau}$, with the following interpretation. $a\tau$ at node u means u is constrained (by the query) to be an ancestor of some node v of type τ, such that v itself has no descendants and no ancestors between u and v. $a\tilde{\tau}$ means the same obligation holds except the type τ at node v is either constrained, or v has an ancestor between u and v. The arguments $p\tau$ and $p\tilde{\tau}$ have similar interpretations. As an example, suppose the query contains a node u of type τ with a d-child v of type τ', which in turn has a d-child w of type τ''. Then we would associate the information arguments τ'' with w, $\tilde{\tau}', a\tau''$ with v, and $\tau, a\tilde{\tau}', a\tilde{\tau}''$ with u. Here, $\tilde{\tau}'$ at v indicates this node is constrained by being required to be an ancestor of some node of type τ. Other arguments can be similarly explained.

Edge	Parent	d/c-Child	Propagated Information
d	τ_1	$\tau_2 \mid \tilde{\tau}_2$	$\tilde{\tau}_1,\ a\tau_2 \mid a\tilde{\tau}_2$
d	τ_1	$a\tau_2 \mid a\tilde{\tau}_2$	$\tilde{\tau}_1,\ a\tilde{\tau}_2$
d	τ_1	$p\tau_2 \mid p\tilde{\tau}_2$	$\tilde{\tau}_1,\ a\tilde{\tau}_2$
c	τ_1	$\tau_2 \mid \tilde{\tau}_2$	$\tilde{\tau}_1,\ p\tau_2 \mid p\tilde{\tau}_2$
c	τ_1	$a\tau_2 \mid a\tilde{\tau}_2$	$\tilde{\tau}_1,\ a\tilde{\tau}_2$
c	τ_1	$p\tau_2 \mid p\tilde{\tau}_2$	$\tilde{\tau}_1,\ a\tilde{\tau}_2$

Figure 4: Information Propagation Rules

5.5 Algorithm CDM: An Overview

We start by labeling each leaf with an information content and then propagate it up the tree in a bottom-up sweep. We make use of propagation rules, which will be explained shortly. Alternating with the propagation is a minimization step. Once the information content has been propagated to a non-leaf node, we inspect the information content at that node to determine whether any of its children is redundant. Nodes determined redundant during this pass are marked as being redundant. As with Algorithm CIM, if a node is marked "*", then it cannot be removed since it is part of the answer.

Information Content Propagation: For each leaf, its information content is the type associated with it. Information content for internal nodes is computed by propagating it from the leaves up the c- or d-edges as appropriate, in accordance with the rules in Figure 4. We explain a couple of rules. The first rule says if the query contains a node u of type τ_1 having a d-child v with associated information argument τ_2, then we propagate the information content $\tilde{\tau}_1, a\tau_2$ to node u. If the argument associated with v was $\tilde{\tau}_2$ instead, we would change the content at u to $\tilde{\tau}_1, a\tilde{\tau}_2$. Rule 2 is very similar, except that the information argument associated with the d-child v happens to be $a\tau_2$ or $a\tilde{\tau}_2$. The propagation rule is the same, since if v is constrained to be an ancestor of some node of some type, this obligation certainly extends to u, which is the d-parent of v. Whenever an internal node has more than one child, the information content propagated from all its children are merged.

EXAMPLE 5.1. **[Information Content Propagation]** *Figure 5 shows a complete example of a query together with ICs, illustrating the propagation as well as minimization steps. We focus only on the propagation aspect now. Step 1 of the figure shows a query asking for instances of type t_1 satisfying the tree pattern. It also shows propagation along each of the three branches. For example, the left-most leaf is labeled by its unconstrained type t_6.*[3] *In accordance with the propagation rules, its c-parent of type t_5 gets the information content $\tilde{t}_5, pt_6$ (rule 4, Figure 4). Similarly, the d-parent of*

[1] In the following, when we say a node v is of type τ, we mean that it is the original type associated with that node, not added in as a result of augmentation.

[2] Not necessarily a direct d-child: there could be a sequence c- or d-edges on the path from u to w.

[3] Information contents appear in boxes.

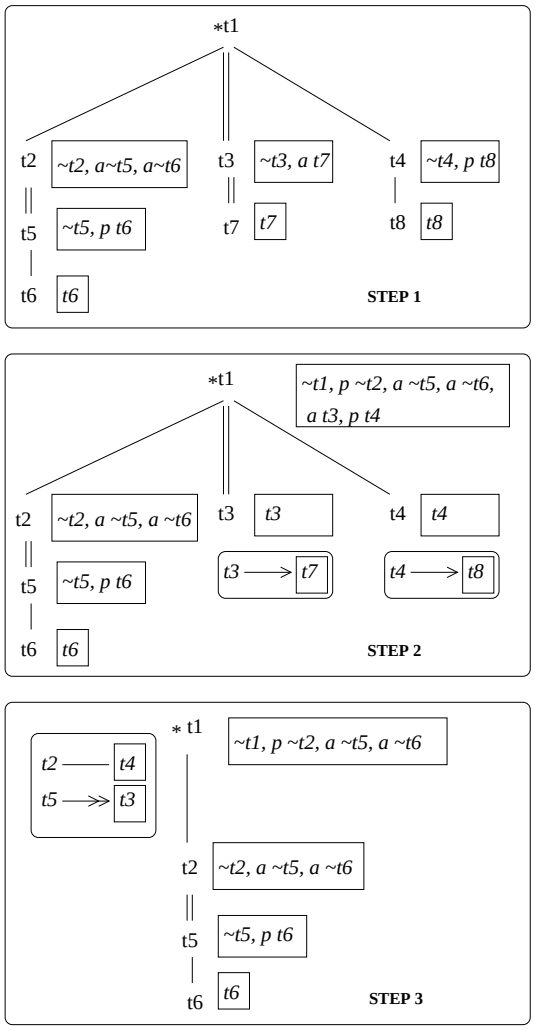


Figure 5: A CDM Example; ICs shown at point of application

this node is of type t_2, and accordingly it gets the information content $\tilde{t}_2, at_5, at_6$ (rule 3). Propagation up the other branches is similar. ∎

Minimization: Like propagation, minimization can be expressed in the form of rules. The objective of minimization is to mark redundant nodes. In addition, we may sometimes need to change the status of some information arguments from "constrained" to "unconstrained". The minimization rules are given in Figure 6. Rule 1 says whenever a node is of type τ_1 and has an obligation (from the query) to be an ancestor of another node of type τ_2, the latter node can be made redundant whenever $\mathcal{C}$ contains the IC $\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$. Rule 2, for obligation for parenthood, is similar, except one needs the IC $\tau_1 \longrightarrow \boxed{\tau_2}$ to effect minimization in this case. The remaining rules deal with the case where a node has two obligations, out of which one can be inferred to be redundant by virtue of an appropriate IC. As an example, rule 4 says when a node has the obligation to be an ancestor or parent of a node which has constrained type τ_1 and it has an obligation to be an ancestor or parent of a node of type τ_2, the latter obligation is redundant whenever $\mathcal{C}$ contains the IC $\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$.

Arg1	Arg2	Constraint	Minimization
$\tilde{\tau}_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
$\tilde{\tau}_1$	$p\tau_2$	$\tau_1 \longrightarrow \boxed{\tau_2}$	make τ_2 node redundant
$a\tau_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
$a\|p\tilde{\tau}_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
$a\|p\tau_1$	$a\|p\tau_2$	$\tau_1 — \boxed{\tau_2}$	make τ_2 node redundant
$a\|p\tilde{\tau}_1$	$a\|p\tau_2$	$\tau_1 — \boxed{\tau_2}$	make τ_2 node redundant

Figure 6: Minimization Rules

The minimization procedure alternates with propagation. Whenever propagation to a node is complete, starting from the parent of leaves, the minimization procedure kicks in. At that node, we apply any applicable minimization rules, rendering children redundant in the process. In addition, whenever all children of a node are marked redundant, the information argument $\tilde{\tau}$ at the node, if any, is changed to τ.

EXAMPLE 5.2. **[Information Content Minimization]** *Let us revisit Example 5.1. Step 2 shows various ICs that can be applied using minimization rules, marking (and eliminating) redundant nodes. For example, the node with (unconstrained) type t_7 can be inferred to be redundant and removed, by virtue of the information content at its parent — $\tilde{t}_3, at_7$ — and the IC $t_3 \longrightarrow \boxed{t_7}$. A similar remark applies to the node with type t_8. At this time, we also update the information argument $\tilde{t}_3$ to t_3 and similarly for $\tilde{t}_4$. Next, we propagate the information content to the root we examine the information content of the root and find that it has $p\tilde{t}_2$ and pt_4. Using the IC $t_2 — \boxed{t_4}$ and minimization rule 6 (in Figure 6), we note the c-child t_4 is redundant and can be removed. Similarly, the d-child t_3 can also be eliminated (Figure 5, PART 3). The resulting query does not have any local redundancy.* ∎

Let us analyze the complexity of Algorithm CDM. A naive analysis shows every pair of nodes is compared at most once, so it is $O(n^2)$, n being the number of nodes in the query. We next undertake a more careful analysis. First, notice that propagation of information content takes time proportional to the number of nodes. Next, the six rules in Figure 6 are essentially captured by the four rules at the beginning of Section 5.4. Of these, the first three involve comparison of a parent with its c- or d-children, or of a d-child with another d-child, or of two c-children. This takes $O(\Sigma_{i \text{ not a leaf}}(f(i)+1)^2)$, where $f(i)$ is the fanout of node i. Let *maxf* be the maximum fanout of any node. Since $\Sigma_{i \text{ not a leaf}}(f(i)) = O(n)$, the above expression simplifies to $O(n \times maxf)$. As for a the last rule, for checking its redundancy, a node w may have to be compared with the descendants of each of its ancestors. Let *maxd* be the maximum depth of the query tree. Then the number of ancestors of w is at most *maxd*. So, the amount of work done for a specific node w is at most $maxd \times maxf$. The overall work done for this step is given by $O(n \times maxd \times maxf)$. Hence, the overall work done by Algorithm CDM can be seen to be no more than $O(\min(n \times maxd \times maxf, n^2))$. For a balanced binary tree with n nodes, for example, this term is $O(n \log(n))$. The min operator signifies the fact that when the tree shape is such that $n \times maxd \times maxf > n^2$, the algorithm still never does more than $O(n^2)$ work, since no pair is compared more than once. One such situation is when $maxf = O(n)$ and $maxd = O(n)$. Notice that Algorithm

CDM has a much better complexity than Algorithm ACIM.

We conclude this section with the following results.

THEOREM 5.2. *Let Q be a tree query and $\mathcal{C}$ a logically closed set of ICs. Let Q' be the result of applying Algorithm CDM to Q, Then Q' is locally minimal, i.e. Q' is equivalent to Q and no leaf in Q' is locally redundant.* ∎

The next theorem says applying CDM prior to ACIM as a pre-filter does not compromise the optimality of ACIM.

THEOREM 5.3. *Let Q be a tree query and $\mathcal{C}$ a logically closed set of ICs. Then, applying Algorithm CDM followed by Algorithm ACIM always yields the unique minimal query equivalent to Q under $\mathcal{C}$.* ∎

One of the main contributions of this section has been the development of Algorithm CDM for finding a query equivalent to a given query which is locally minimal. This, when fed to Algorithm ACIM, will still enable the latter to find the unique (globally) minimal query equivalent to the given query under the given constraints. The main advantage of CDM is as an efficient pre-filter before ACIM takes over. We expect the efficiency gain for this approach compared with directly applying ACIM to come from the fact that all nodes that are removed by CDM will never have to be processed by the more expensive ACIM.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the various algorithms presented in the paper, and experimentally compared their performance for minimization of tree pattern queries, both without and with ICs. The experiments study in detail the minimization time by first separating the study of ACIM and CDM, and then combining them to see how they interact. In all experiments, time is reported in seconds.

6.1 Use of Hash Tables

Constraints are organized in a hash table for efficient retrieval during the minimization process. Given an information content at a node, CDM considers each pair of arguments in this information content and uses them as a key to access the hash table that contains the constraints relevant to the given pair. In the same manner, given a leaf node, ACIM uses it as a key to retrieve relevant constraints and perform augmentations. The ancestor/descendant table as well as the images table are also stored as hash tables. In order to avoid the additional overhead required by the ACIM algorithm (because of the constrained augmentation), augmentations are not physically added to the initial query. They are maintained only as redundant nodes in the images and the ancestor/descendant tables.

6.2 ACIM

ACIM time depends on the number of redundant nodes in a query pattern, the degree of redundancy (which is the number of times a node is redundant), the query size and finally, the number of constraints that might generate additional redundancy in the query pattern. We ran two sets of experiments. The first set shows the variation of ACIM time with a growing number of constraints starting from 0. The second one shows, for a fixed number of constraints (100), the proportion of time ACIM spends in building the images and the ancestor/descendant tables. We verified that ACIM response time is a function of the *total* number of redundant nodes, irrespective of whether this total was obtained by fixing the degree of redundancy and varying the number of redundant nodes, or by fixing the number of redundant nodes and varying the degree of redundancy. Thus, we report the variation of ACIM time as a function of the total number of redundant nodes.

Varying Redundancy and Constraints: We consider a query with 101 nodes. We varied the number of redundant nodes from 1 to 90 and the degree of redundancy from 1 to 40. We ran ACIM with no constraints, 50, 100 and 150 constraints relevant to the query. The graphs of Figure 7(a) show the variation of ACIM time, as a function of the total number of redundant nodes. Essentially, ACIM time stayed about the same for a given number of constraints, when varying the total number of redundant nodes, while keeping the query size fixed. Further, the larger the number of relevant constraints, the more is the time taken by ACIM; this increase appears to be linear in the number of constraints.

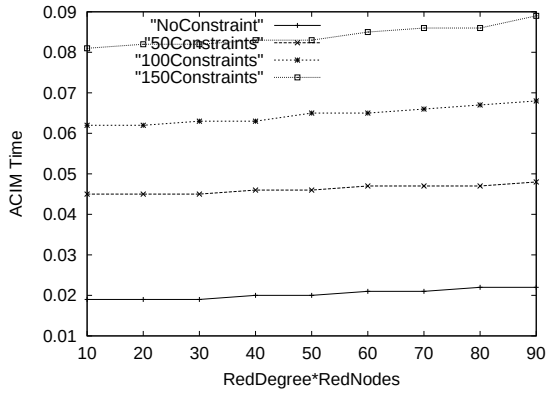
Total Time and Tables Time: We report the total time it takes to execute ACIM and the fraction of this time spent to build the images and the ancestor/descendant tables. The query we considered has 101 nodes and the number of constraints relevant to this query is 100 (thus all nodes except the root node were redundant). The graph of Figure 7(b) shows that the time spent in building these tables is around 60% of the total ACIM time. The same results hold for other numbers of constraints.

6.3 CDM

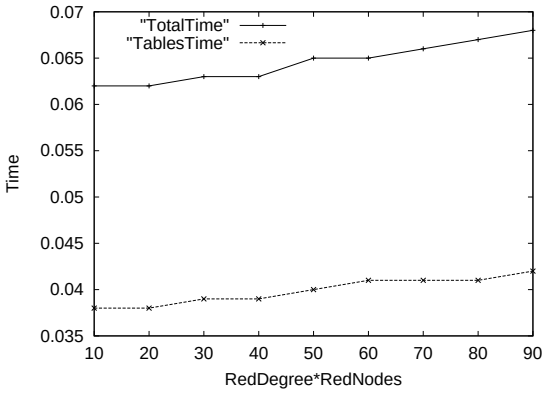
We ran two sets of experiments. The first set shows that the CDM algorithm does not depend on the total number of available constraints in the constraint repository, because of its use of hashing techniques. The second set of experiments aims to understand how CDM time varies with varying query sizes, shapes and node fanouts.

Varying Constraints: Figure 8(a) shows that, for a fixed query size (containing 127 nodes) and a growing number of relevant constraints (from 0 to 150), CDM time remains constant. This shows that our CDM algorithm does not depend on the number of schema constraints. This is due to the fact that, given the two arguments in the information content of a node, the algorithm uses them as a (combined) key to access the hash table and verify whether a constraint involving both of them is present. This check is an access to the hash table where constraints reside and does not depend on the number of constraints in this table.

Varying Query Size: Figure 8(b) reports three graphs. Two of them are very similar and show the variation of the CDM time with a growing query size. Given a set of constraints (in this case, fixed to 110), we generate queries for which all the constraints are relevant. The only marked node is the root node. Because of the way the query is generated (all edges are redundant), the only node that remains after query minimization is the root node. The shape of a query does not have any impact on the CDM time. Right-deep and bushy tree pattern queries have very similar performance results. The experiments show that the CDM algorithm grows in a linear fashion, for a fixed fanout. The third graph shows the evolution of CDM with an increasing node fanout. In general, CDM behaves in a quadratic fashion with respect to

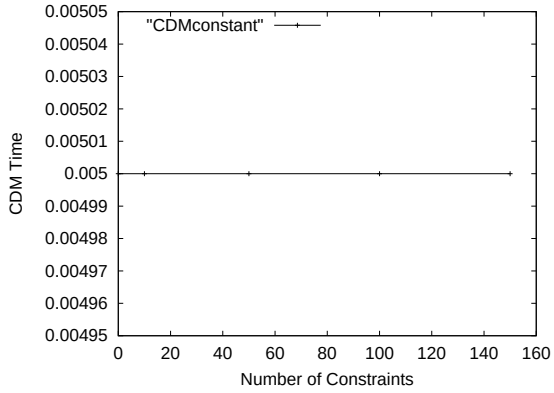


(a) Varying Redundancy and Constraints

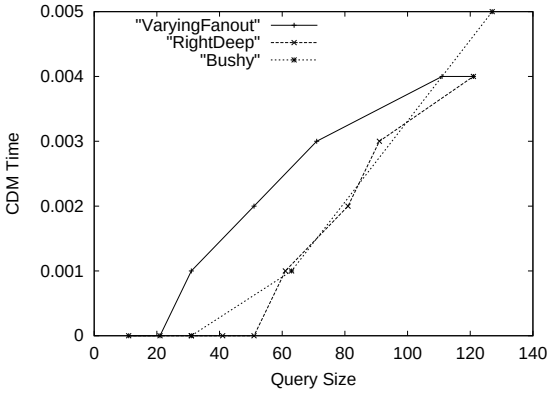


(b) Total Time and Tables Time

Figure 7: Studying ACIM

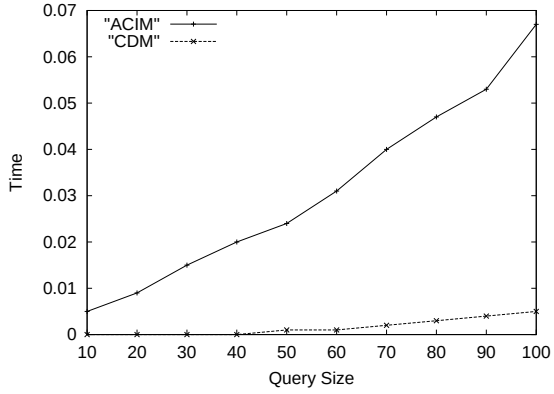


(a) Varying Constraints

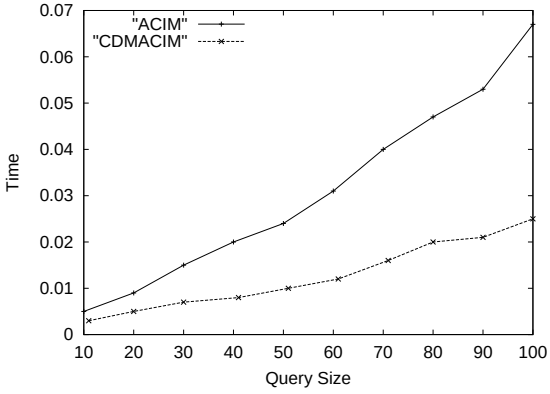


(b) Varying Query Size

Figure 8: Studying CDM Time



(a) ACIM and CDM with a Varying Query Size



(b) Varying Number of Redundant Nodes

Figure 9: Comparing/Combining ACIM and CDM

the node fanout. However, CDM time remains small (within a few milliseconds) for large queries (containing more than 120 nodes).

6.4 Combining Both Minimizations

In order to understand how ACIM and CDM algorithms compare, we ran two sets of experiments. In the first set, we build a query where the number of nodes removed by CDM is the same as the number of nodes removed by ACIM and we increase the query size while preserving this property. We ran ACIM and CDM on the same query separately and compare their performance. The second set of experiments aims to compare the direct application of ACIM with the use of CDM as a pre-filter to ACIM.

ACIM versus CDM: In order to compare the ACIM and CDM algorithms, we built a query where ACIM and CDM would remove the same set of nodes if applied separately. CDM outperforms ACIM substantially. The graphs in Figure 9(a) show that the time to perform CDM is significantly smaller than the time to perform ACIM with a growing query size. Further, the time difference between the two algorithms grows with a growing query size.

CDM as a Pre-filter: This experiment shows the benefit of using CDM before ACIM. Since ACIM is more complex than CDM, we expect that it is beneficial to run CDM first and remove all local redundancies before running ACIM (on a potentially smaller query). We report on an experiment (in Figure 9(b)) where CDM removes half the nodes that ACIM can remove. Our results show that applying CDM as a pre-filter always outperforms the direct application of ACIM, and that the advantage increases with increasing query size.

7. CONCLUSIONS AND FUTURE WORK

Tree patterns form a natural basis with which to query tree databases such as XML and LDAP style directories. Query answering in this context can be considerably improved by reducing the pattern size. Doing so is closely related to conjunctive query minimization: a problem that is in general NP-complete for classical relational database queries. In this paper, we showed that for tree pattern queries, in the absence of ICs, this problem can be solved in polynomial time. Indeed, there is a unique minimal equivalent query for a given query. This happy situation extends also to the case when minimization must be performed under the presence of required child, descendant, and co-occurrence ICs — constraints that are fairly natural for tree-structured databases. In addition to providing efficient algorithms for minimization with and without ICs, we also established their practicality using an experimental study.

There are several interesting directions for further work. First, tree pattern queries may involve value-based conditions, e.g., that the price of a book always be less than $100, in addition to the structure-based conditions. How can one extend the techniques developed in this paper to this case? Our intuition is that, in this case, we should still be able to apply the techniques developed in this paper, with the following modification: when we consider endomorphisms (for ACIM), a node u cannot be mapped to a node v unless the conditions at node v logically entail those at node u. The main impact of incorporating value-based conditions should be an increase in the complexity owing to reasoning about implication of value-based conditions.

Another challenging direction is to consider larger classes of tree-structured constraints: for example, those that forbid certain types of children or descendants, or require/forbid certain types of parents and ancestors. In this case, there may be no unique minimal equivalent query, since a node might be redundant without its children nodes being redundant. We conjecture, though, that all minimal equivalent queries would be of the same size.

8. REFERENCES

[1] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. *PODS 1998.*

[2] S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. *Foundations of DD and LP 1988.*

[3] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. *WebDB 2000.*

[4] E. P. F. Chan. Containment and minimization of positive conjunctive queries in OODBs. *PODS 1992.*

[5] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. *STOC 1977.*

[6] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. *SIGMOD 2000.*

[7] A. Deutch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. A query language for XML. *WWW 1999.*

[8] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. *PODS 1998.*

[9] T. Howes, M. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services.* Macmillan Technical Publishing, Indianapolis, Indiana, 1999.

[10] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. *SIGMOD 1999.*

[11] P. G. Kolaitis, D. L. Martin, and M. N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. *PODS 1998.*

[12] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *PODS 1998.*

[13] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects. *PODS 1997.*

[14] J. McHugh and J. Widom. Query optimization for XML. *VLDB 1999.*

[15] T. D. Millstein, A. Y. Levy, and M. Friedman. Query containment for data integration systems. *PODS 2000.*

[16] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? *SIGMOD 2000.*

[17] L. Popa and V. Tannen. An equational chase for path-conjunctive queries, constraints, and views. *ICDT 1999.*

[18] Y. P. Saraiya. Polynomial-time program transformations in deductive databases. *PODS 1990.*

[19] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.