Tree Pattern Query Minimization

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

SungRan Cho
Stevens Institute of Technology
scho@puma.mt.att.com

Laks V. S. Lakshmanan
University of British Columbia
laks@cs.ubc.ca

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. We develop a technique for query minimization based on three fundamental operations: augmentation (an adaptation of the well-known chase procedure), minimization (based on homomorphism techniques), and reduction. Under the classes of ICs mentioned above, we show that there is a unique minimal equivalent query that is obtainable by arbitrary application of these operations. 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 [14], 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 similarly to 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 [8]. 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 [5]. Existing techniques for semantic query optimization base themselves on the notion of a residue using 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.

- For the class of ICs consisting of required children, required descendants, and required co-occurrences, we develop a technique for query minimization based on three fundamental operations: augmentation, minimization (based on homomorphism techniques), and reduction; of these, augmentation is an adaptation of the well-known chase procedure, and adds redundant nodes and edges to a query. We show that there is a unique minimal equivalent query obtainable by arbitrary applications of the three operations above (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 minimal equivalent query (Section 5). This algorithm, which we call ACIM, is also polynomial time, 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 pattern tree node with an *information content*, and propagates it up the pattern tree (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 pre-filter 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 [8]. Saraiya [23] 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. [16] 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 [17] establish a fundamental connection between conjunctive query containment and constraint satisfaction in AI. Florescu et al. [13] show that containment of conjunctive queries with regular path expressions over semistructured data is decidable, using a form of the chase technique. Under some restrictions they show that the problem is NP-complete. Chan [7] 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 [18] 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. [20] 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. [5] 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. [4] 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. A logic-based approach is adopted in [22] 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, [21] uses a stratification technique in their implementation to reduce the search space during query optimization time.

Recently, researchers have begun investigating containment and equivalence of fragments of XPath expressions. XPath is one of the standards proposed by the W3C consortium for querying XML documents and has been incorporated in their latest standard, XQuery [3]. Wood [25, 26] showed that containment for the fragment of XPath involving descendant edges (// in XPath), wildcards, and branching is decidable. Deutsch and Tannen [11, 12] consider containment of various fragments of XPath expressions under various ICs, also expressible in XPath, and establish decidability, undecidability, and hardness results. Our results in this paper (which is a full version of [2]) are not covered by any of these works. In particular, we propose efficient algorithms for minimizing tree pattern queries arising in LDAP and XML applications, possibly in

the presence of required child, descendant and co-occurrence ICs, as opposed to simply studying containment or equivalence. Further, none of the works that are about query optimization for XML such as [19] and [9] consider IC-based optimization.

Finally, we note that the kind of tree pattern queries studied in this paper include both child- and descendant-edges, the latter being the transitive closure of the former. Thus, it is tempting to consider minimization of tree pattern queries as a special case of minimization of datalog query programs. However, this perspective does not lend itself to any useful insights into how and when efficient algorithms for minimization can be designed, since containment for datalog with recursion is in general undecidable [1]. It is more fruitful to view tree pattern queries as conjunctive queries with interpreted predicates, and this is what we do in this paper.

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 one or more associated types. Depending on the application, node order in trees may be important. Two main applications inspiring this model are XML and LDAP-style network directories. 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 considered important, and the edges may represent inherent hierarchies such as organizational or geographical.

Queries are essentially based on a *pattern tree*. Queries in languages such as XQuery [3], XML-QL [10] and Quilt [6] are based on a notion of a pattern tree using which they extract relevant portions of XML data. Queries on LDAP directories ask for entries that stand in specified structural relationships (parent, descendant, etc.) to other specified entries and are naturally represented as trees [15]. We express queries as tree patterns where nodes are types and edges are child/descendant relationships.

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 pattern tree 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. We do not consider order in our queries. In each query, one node is marked with a "*". For network directory applications, this is interpreted to mean only entries corresponding to the marked node are returned as the

```
<xsd:complexType name="Book">
   <xsd:element name="Title" type="xsd:String"/>
   <xsd:element name="Author" type="xsd:String"
                minOccurs="0" 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.
- $\tau_1 \text{ — } \boxed{\tau_2}$: 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

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.

Given a query tree pattern Q, and a tree database $\mathcal{D}$, a *match* of Q in $\mathcal{D}$ is identified intuitively by a mapping ψ from nodes in Q to nodes in $\mathcal{D}$, such that: (i) if node u in Q has type τ, then so does node $\psi(u)$ in $\mathcal{D}$, (ii) if v is a c-child (resp., d-child) of u in Q, then $\psi(v)$ is a child (resp. descendant) of $\psi(u)$ in $\mathcal{D}$, and (iii) if node u in Q is marked with a "*", then the result of the match identified by mapping ψ is the node $\psi(u)$ in $\mathcal{D}$. The result of a tree query Q against tree database $\mathcal{D}$, denoted $Q(\mathcal{D})$, is the (set) union of the results of all matches of Q in $\mathcal{D}$.

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 $\texttt{C}_i$ in it such that type $\texttt{C}_i$ is known to require a (descendant) subtype `B`, we conclude type `A` must have a descendant subtype `B`.

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

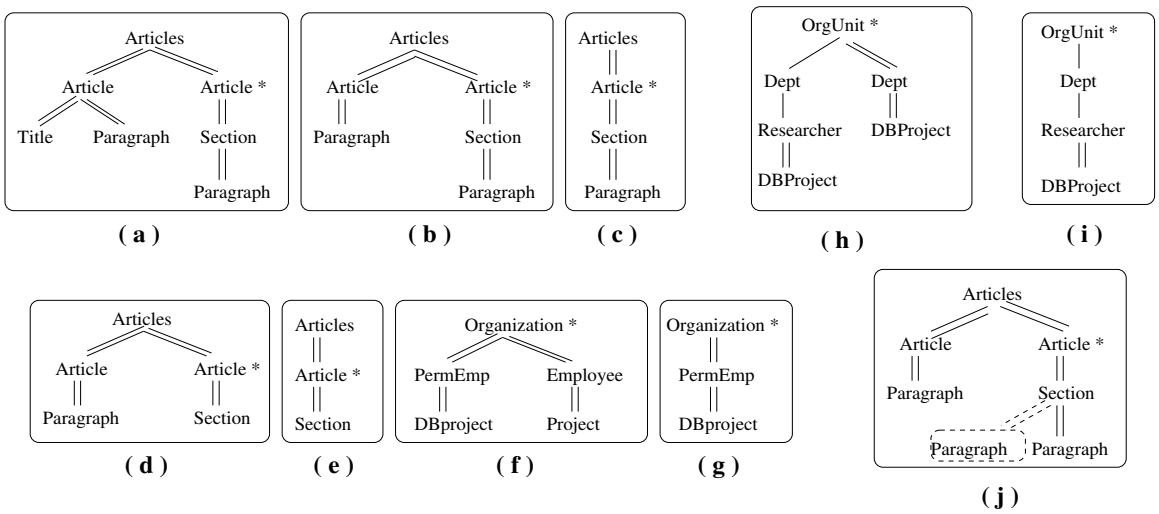


Figure 2: Examples of Tree Pattern Queries

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 with realistic examples.

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, denoted $Q_1 \subseteq Q_2$, provided for every tree database $\mathcal{D}$, $Q_1(\mathcal{D}) \subseteq Q_2(\mathcal{D})$. Equivalence, denoted $Q_1 \equiv Q_2$, is two-way containment. A query is *minimal* if there is no equivalent query which has a proper subset of nodes of the original query. Then, our minimization problem is as follows.

Problem P_1: Given a tree 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, since (for example) some departments may not have a researcher.

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, denoted $Q_1 \subseteq_{\mathcal{C}} Q_2$, 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, denoted $Q_1 \equiv_{\mathcal{C}} Q_2$, 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 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.

We attack this problem by developing operations (augmentation, minimization, and reduction), and then characterize minimal equivalent queries obtainable via application of these operations.

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.

1. 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 intuitively see that the `Organization--Employee--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.

2. 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 CIM.

3. Next, consider the query of Figure 2(b). Suppose we know the IC `Section` $\longrightarrow\!\!\!\!\rightarrow$ `Paragraph` holds, this 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 CIM and CDM may be important in obtaining a minimal equivalent query.

4. 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 CIM to reason that the `Articles--Article--Paragraph` is subsumed by the `Articles--Article--Section--Paragraph` path, 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 form 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

4.1 Minimal Equivalent Query

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 mapping from a query Q' to a query Q is a mapping $h : Q' \rightarrow Q$ from the nodes of Q' to the nodes of Q such that:

- h preserves node types: $\forall u$, u and $h(u)$ have the same type, and $h(u)$ has the "*" label whenever u has the "*" label;
- h preserves structural relationships: whenever v is a c-child (resp., d-child) of u in Q', $h(v)$ is a c-child (resp., descendant) of $h(u)$ in Q.

  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.

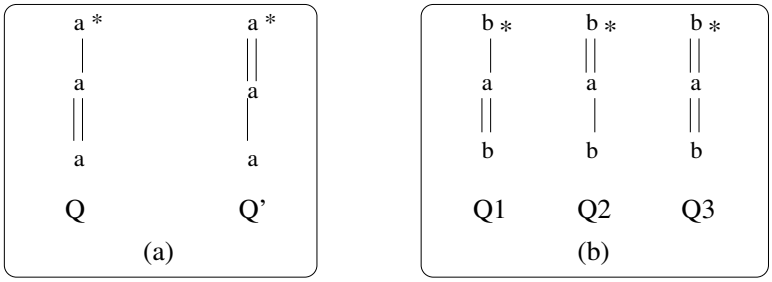


Figure 3: Counterexamples to the necessity of containment mappings on databases over a small alphabet.

In the case of relational databases, the homomorphism theorem of Chandra and Merlin [8] says the existence of a containment mapping (a homomorphism from the containing query to the contained query) is a necessary and sufficient condition for containment. While the existence of a homomorphism continues to be sufficient for tree queries, it is not always necessary, as the following example shows.[1]

Example 4.1 Consider the queries Q and Q' shown in Figure 3(a). There is no containment mapping from Q to Q' that preserves edge types. Yet they are equivalent, on databases whose nodes have only one type. The reason is that all paths between nodes must pass through intermediate nodes, if any, whose node types are all the same, say a. Thus, there is essentially no distinction between child edges and descendant edges. Suppose the node types are drawn from a binary alphabet, say $\{a, b\}$. Surprisingly, even now, existence of a containment mapping is not necessary for containment. Figure 3(b) shows three queries Q_1, Q_2, Q_3. It can be shown that on databases whose node types are restricted to be a or b, $Q1 \subseteq Q2$, and $Q_2 \equiv Q_3$. However, any mappings between these queries would map a child edge to a descendant edge, violating the condition of a containment mapping. ∎

In the sequel, we shall assume that input databases of interest have their nodes types drawn from an alphabet of unbounded size. This is a reasonable assumption, and one that covers almost all practical cases. For LDAP, node types correspond to object classes, whereas for XML they correspond to tags. Both of these are extensible. We first reprove the Chandra and Merlin homomorphism theorem for tree queries, under this assumption.

Theorem 4.1 *Let Q and Q' be any tree queries. Then Q is contained in Q' iff there is a homomorphism from Q' to Q.*

Proof: Sufficiency is straightforward. Suppose then that Q is contained in Q'. We construct an input database D_Q from the query Q as follows. For every node x in Q, D_Q contains a corresponding node x', with the same node type as x. Whenever Q contains a child edge (x, y), D_Q contains the edge (x', y'). Whenever Q contains a descendant edge (x, y), D_Q contains the edges (x', z) and (z, y') where z is a new node that appears nowhere else. The node type of z is chosen to be different from that of every node in Q'.

[1] We would like to thank Dan Suciu for alerting us to this phenomenon.

Call every such node z a "new" node. Suppose u is the distinguished node of Q and u' the corresponding node in D_Q. Let f be a mapping from Q to the database D_Q that sends a node x to its corresponding node x' in D_Q. Clearly, f is an embedding of Q into D_Q. It follows that $u' \in Q(D_Q)$. Since $Q \subseteq Q'$, we must have $u' \in Q'(D_Q)$. Let g be an embedding of Q' into D_Q verifying this. Clearly, g cannot map any node of Q' to any of the "new" nodes of D_Q, since their node types do not match any node in Q'. Also, notice that whenever (x, y) is a child edge of Q', $(g(x), g(y))$ must be an edge of D_Q. The mapping f^{-1} is well defined since f is 1-1. It is easy to see that the mapping $g \circ f^{-1}$ is indeed a containment mapping from Q' to Q. ∎

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 is equivalent to Q. Notice that a node marked "*" can never be redundant. We now 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.*

Proof: The "Only if" direction is straightforward and follows from Theorem 4.1. For the "If" direction, suppose there is an endomorphism η on Q such that $v \notin \eta(Q)$. Again, it follows from Theorem 4.1 that v is redundant in Q. Suppose, however $v \in \eta(Q)$, but $\eta(v) \neq v$. We will show that in this case, there must exist another endomorphism μ such that $v \notin \mu(Q)$, from which the "If" direction will follow as above.

Define μ to be identical to η except for the following differences. Let u be any node that is mapped to an ancestor (not necessarily proper) of v by η. For every such u, set $\mu(u) = u$. Furthermore, for every descendant w of such a node u, set $\mu(w) = w$. To see that μ so defined is an endomorphism, notice that μ trivially preserves node types. Let (a, b) be a c-edge. If μ agrees with η on both a and b, or disagrees on both, $(\mu(a), \mu(b))$ must be a c-edge as well. So suppose $\mu(a) \neq \eta(a)$. This is possible only when $\eta(a)$ is an ancestor of v or there is a node x such that $\eta(x)$ is an ancestor of v and a is a descendant of x. In either case, $\mu(a) = a$, and furthermore, the construction ensures that μ sends every descendant of a, including b, to itself, so $(\mu(a), \mu(b)) = (a, b)$, trivially preserving the c-edge. Suppose $\mu(b) \neq \eta(b)$. Again, $\eta(b)$ must be an ancestor of v, or b must be a descendant of a node x that is mapped to an ancestor of v by η. In the former case, $\eta(a)$ must be an ancestor of v as well. In the latter case, $\eta(a)$ must be either an ancestor of v or v must be a descendant of a node that is mapped to an ancestor of v by η. In all cases, μ must be identity on both a and b.

Next, consider a d-edge (a, b). If μ agrees with η on both a and b, then $(\mu(a), \mu(b))$ must be an ancestor-descendant pair in the query tree. If it disagrees on both, then $(\mu(a), \mu(b)) = (a, b)$. By an argument similar to that used for c-edges, we can show that whenever μ disagrees with η on either a or b it must necessarily disagree on the other.

In sum, we have shown that for an arbitrary edge (a, b), $(\mu(a), \mu(b))$ is a c-edge whenever (a, b) is a c-edge and it is an ancestor-descendant pair whenever (a, b) is a d-edge, showing μ is an endomorphism. The proof is complete, on noting that μ does not map any node to v. ∎

For a leaf v of a tree query Q, we use $Q - \{v\}$ to denote 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 each 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.*

Proof: By the definition of maximality of the elimination ordering, we know Q' does not contain any redundant leaves. Suppose it has an internal node u that is redundant. This implies there is an endomorphism on Q' that is not identity on u. In this case, this endomorphism cannot be identity on any of the descendants of u either, implying all its descendants must be redundant as well. This contradicts the fact that Q' has no redundant leaf. ∎

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 a query Q'' with no more nodes than Q', not necessarily distinct from Q', equivalent to Q', for which there is an elimination ordering such that Q'' can be obtained from Q via that elimination ordering.*

Proof: Let V be the set of nodes in Q but not in Q'. Suppose V is downward closed, in that whenever a node u is in V, then any descendant of u is in V too. Then the required query Q'' is identical to Q'. Suppose V is not downward closed. Let $u \in V$ and v be a descendant of u in Q, i.e., there is a path involving child and/or descendant edges from u to v, such that v is not in V. Since Q' is equivalent to Q, there is a containment mapping from Q to Q', i.e., an endomorphism on Q, verifying this. Any endomorphism that is not identity on u cannot be identity on v either, showing v can be eliminated while preserving equivalence. Let $V' = V \cup \{v \mid v \text{ is a descendant of a node in } V\}$ and let Q'' be the query obtained from Q by deleting all nodes in V'. Clearly, Q'' is equivalent to Q and Q'. Let $\mu : Q \rightarrow Q''$ be a containment mapping verifying this equivalence. Order the nodes in V' into a sequence $(v_1, .., v_k)$ such that v_i is a leaf of $Q - \{v_1, \ldots, v_{i-1}\}$. It follows that this is an elimination ordering, since $\mu|_{\text{nodes}(Q) - \{v_1, .., v_{i-1}\}}$ verifies that v_i is redundant in $Q - \{v_1, \ldots, v_{i-1}\}$. ∎

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.*

Proof: Let $\mu_1 : Q \rightarrow Q'$ and $\mu_2 : Q' \rightarrow Q$ be the containment mappings that verify the equivalence of Q and Q'. The composition $\eta = \mu_2 \circ \mu_1$ is an endomorphism on Q. Since Q is minimal, η must be an identity mapping. If either μ_1 or μ_2 is not 1-1, it is easy to see that η cannot be an identity mapping. Thus, μ_1 and μ_2 are both 1-1, showing Q and Q' are of the same size. By the definition of containment mappings, μ_i preserves node types. Furthermore, whenever (x, y) is a c-edge of one query, $(\mu_i(x), \mu_i(y))$ is a c-edge of the other. It remains to show that whenever (x, y) is a d-edge of one query, $(\mu_i(x), \mu_i(y))$ is a descendant of the other. Suppose not. Let (x, y) be a d-edge of Q and that $(\mu_1(x), \mu_1(y))$ be a c-edge of Q'. In this case, $\mu_2(\mu_1(y)) \neq y$. For that would entail mapping a c-edge of Q' to a d-edge of Q. This contradicts the fact that η is an identity mapping. This shows μ_1 is an isomorphism. By symmetry, so is μ_2. ∎

Combining the above lemmas, we get the following result.

Theorem 4.2 *Let Q be any tree query. Then it has a minimal equivalent query which is unique up to isomorphism.*

Proof: Follows from the above lemmas. ∎

This theorem only says that by following an arbitrary MEO, we can obtain a minimal equivalent query (which is the query with the least size, unique up to isomorphism). 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.

4.2 Efficiently Determining Redundancy

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 u 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 have the following result.[2]

Theorem 4.3 *Let v be a leaf of a tree query Q. Then v is redundant iff at the end of the above pruning procedure the set $images(r)$ associated with the root r is non-empty.*

Proof:
($\Leftarrow$:) Suppose the leaf v of query Q is redundant. Then there must be an endomorphism η on Q that is not an identity on v. Then clearly $\eta(u)$ must be in $images(u)$, for every leaf u of Q. Let p be the parent of v and $c_1, \ldots, c_n$ be the children of p (which includes v). Since $\eta(c_i)$ is a child (or descendant, as appropriate) of $\eta(p)$, $i = 1, \ldots, n$, the entry $\eta(p)$ would not be deleted from $images(p)$ by the algorithm. A simple inductive argument would show that $\eta(r)$ would not be deleted from $images(r)$ either. This shows $images(r)$ is non-empty at the end of the algorithm.

($\Rightarrow$:) Suppose $images(r)$ is non-empty at the end of the algorithm. We shall show that v must be redundant. First, observe that for any node p and for any x in $images(p)$:

(a) if u is a c-child of p then there exists a node y in $images(u)$ such that y is a c-child of x.

(b) if u is a d-child of p then there exists a node y in $images(u)$ such that y is a d-child of x.

The observation follows directly from the pruning procedure, for x would otherwise have been pruned from $images(p)$.

We can define an endomorphism η, top-down, as follows. First, set $\eta(r) = r$ (where r is the root). Also, r is in $images(r)$ of necessity. Then at each node p, recursively, if we already defined $\eta(p) = x$, where x is some node in $images(p)$, we can extend η on descendants of p as follows. If u is a c-child of p then define $\eta(u)$ to be some arbitrary y in $images(u)$ such that y is a c-child of x. Follow a similar procedure for d-children. Such y exists by the observation above. Since the query is a tree, η will be well-defined. (Because every node has a unique parent, thus it cannot be mapped to more than one node). Moreover, η is an endomorphism because it preserves c- and d-edges. Also, $\eta(v) \neq v$, because v is not in $images(v)$. ∎

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

[2] We would like to thank the anonymous reviewers for the proof of this result.

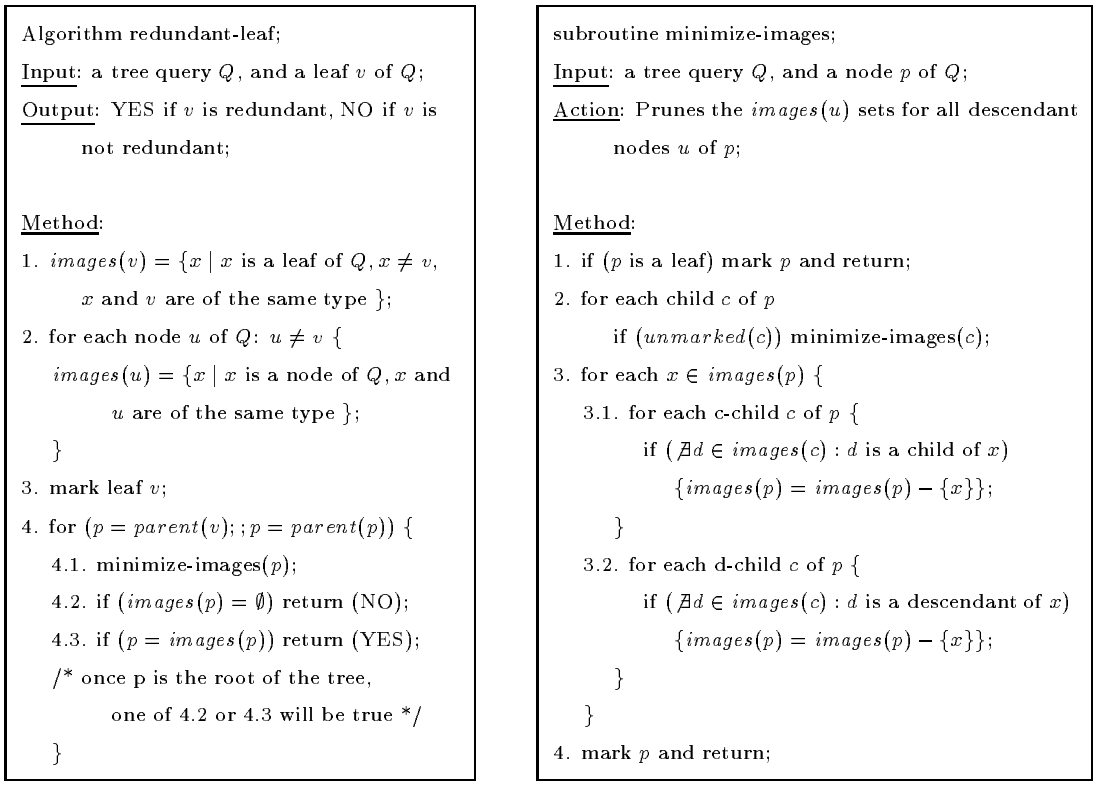


Figure 4: An Efficient Implementation of CIM

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 in practice with a more efficient implementation, as given in Figure 4.

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 4) 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. Theoretically, 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 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, we have the following result.

Theorem 4.4 *The worst-case time complexity of Algorithm CIM 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. While this question still remains open, in this section, we define three natural operations – augmentation, minimization, and reduction – and show that when only required child, descendant and co-occurrence constraints are considered, there is a unique minimal equivalent query that is obtainable by applications of the above operations to Q. 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 [24], 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 a constraint 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 constraint `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, resulting in the query given in Figure 2(j). 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.

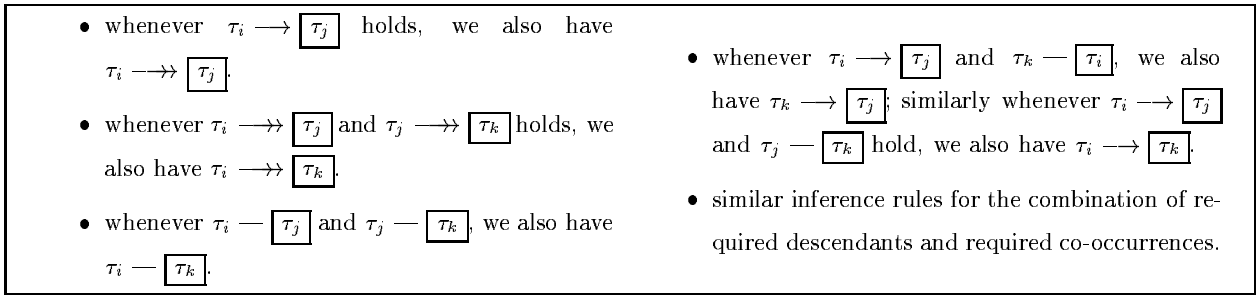


Figure 5: Inference Rules

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 with the application of the inference rules shown in Figure 5. It is easy to verify that these inference rules are sound and complete for the given class of ICs.

The second change to chase is that we apply only 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 known to be 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:

1. Augment the query w.r.t. the closure of the given set of ICs;
2. Apply the CIM algorithm of the previous section, ensuring that temporary nodes inserted by the augmentation phase are *not* checked for redundancy; and
3. 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 only IC `Section` $\longrightarrow\!\!\!\!\!\rightarrow \boxed{\texttt{Paragraph}}$. 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 as *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 to an equivalent query of the least size. Since augmentation can often facilitate the identification of redundant nodes/edges, we can also include A in the language of strategies.

Various strategies to 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.*

Proof: If μ is the identity mapping, set μ' to be identity on Q'. Suppose μ is a non-trivial endomorphism. In this case, it must map some leaf u of Q to another leaf v. In all cases, we start by setting μ' to be identical to μ on the nodes of Q. Let w be the closest ancestor of u in Q on which μ is identity. Clearly, w must be an ancestor of v too. If the subtrees of Q and of Q' rooted at w are identical, the lemma follows trivially: for every new node of Q', set μ' to be an identity. So suppose not. Let x be any node in the aforementioned subtree of Q that has a new child in Q'. If x is w, again μ' can be an identity on all new children/descendants of x. Let x be a proper descendant of w in Q. In this case, for every new child y of x in Q', there must be a corresponding new child y' of $\mu(x)$ in Q, such that their classes match. This argument extends to arbitrary (new) descendants of x. Then setting $\mu'(y) = y'$ will lead to a consistent endomorphism.

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) $RM \sqsubseteq AMR$.*

Proof: Identities (i) and (ii) follow from the fact that reduction and minimization never add nodes/edges. From Lemma 5.1, it follows that every node/edge eliminated by a direct application of minimization will certainly be eliminated by an application of minimization to the augmented query. Furthermore, any redundant nodes/edges eliminated by R after the M, if present in $AM(Q)$, will certainly be eliminated by the following R, which will also eliminate all temporary nodes/edges added by A. Identity (iii) follows from this.

It remains to prove the last one. To see it, we claim that any node/edge that is redundant in $R(Q)$ will also be redundant in $A(Q)$. Here, by redundancy, we mean there is an endomorphism that is not identity on that node. Furthermore, any node eliminated by R in Q, if present in $AM(Q)$ would be eliminated by the following R. From these, the last identity follows. Let us now prove the above claim by induction on the number n of nodes of Q eliminated by R. Consider the base case of $n = 1$. Suppose a d-child v of a node u was eliminated by R. The case of c-child is analogous. Let the types of u, v be τ_1, τ_2 respectively. The IC in $\mathcal{C}$ used for this reduction must be $\tau_1 \longrightarrow\!\!\!\!\rightarrow \boxed{\tau_2}$. Clearly, $A(Q)$ would contain a temporary d-child v' for u, of type τ_2. Let w be any leaf of $R(Q)$ that is redundant. There must be some other leaf w' to which w can be consistently mapped by an endomorphism, say μ, on $R(Q)$. Such an endomorphism can be extended to an endomorphism μ' on $A(Q)$ as follows. If μ is identity on u, then μ' is identity on v and v'. So suppose $\mu(u) = u'$, where u' is another leaf of $R(Q)$. In this case, u' must be of type τ_1 and augmentation would add a d-child, say v'', of type τ_2 to u' too. In this case, set $\mu'(v) = \mu'(v) = v''$ and $\mu'(v'') = v''$. It is easy to see μ' is an endomorphism. When $n > 1$, the argument is similar to the base case. This proves the claim, as required. ∎

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, $AMRAMR(Q) = AMR(Q)$.*

Proof: First, we observe that $AR = R$ and $RA = A$. The reason for the first equality is that when reduction is performed on an augmented query, all temporary nodes added by the augmentation will certainly be deleted. Furthermore, every node of Q that would be deleted by R would surely be deleted by R after augmentation as well. The reason for the second equality is similar. Now, clearly, $AMRAMR = AMAMR$, since $RA = A$. Let μ be any non-trivial endomorphism on $AMA(Q)$. Clearly, its restriction to $AM(Q)$ must be a non-trivial endomorphism on $AM(Q)$. But, by definition, there can be no non-trivial endomorphism on $AM(Q)$, since an application of M renders any query minimal w.r.t. containment mappings. Thus, there cannot be any non-trivial endomorphism on $AMA(Q)$ either, which implies $AMAM(Q) = AMA(Q)$. Since Q was arbitrary, we have $AMAM = AMA$. Now, $AMAMR = AMAR = AMR$. ∎

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 obtained by an arbitrary number of applications of A, M and R.*

Proof: If ϕ contains no occurrence of A, it must be of the form $(MR)^n$, or $(MR)^n M$, or $(RM)^n$, or $(RM)^n R$, for some $n \geq 0$. Note that consecutive M's and consecutive R's can be simplified to one M and one R respectively. Now, for each of the above cases, we can replace each substring of the form MR, or RM, or M, or R, by the string AMR and obtain a string of the form $(AMR)^k$, where k is n or $n+1$, such that the original string is dominated by $(AMR)^k$. As an example, $(MR)^n M \sqsubseteq (AMR)^{n+1}$. Since AMR is idempotent, the latter string is dominated by AMR.

Suppose now ϕ contains at least one occurrence of A. Consider the left-most occurrence and split ϕ as $\alpha A \beta$, where α is A-free. If β is empty, $\phi \sqsubseteq \alpha$, which reduces to the above case, so suppose β is not empty. If β contains no occurrence of R, then $\phi \sqsubseteq \phi R$, so without loss of generality, assume β contains an R. Consider the left-most occurrence of R in β and split ϕ as $\alpha A \gamma R \delta$. If γ is empty, AR reduces to R, and the A following α is redundant. If all A's in ϕ are redundant in this sense, we are back to the previous case, so assume this A is not redundant. So γ must be non-empty. γ cannot start with A and it does not contain R by assumption. So, γ must be of the form $M\eta$. Any A in η is redundant and can be eliminated. Consequently, any M in η also becomes redundant (owing to the preceding M) and can be eliminated. So, without loss of generality, we can assume our ϕ is of the form $\alpha AMR\delta$, where α is A-free. Then since AMR dominates every single letter, it follows that $\phi = \alpha AMR\delta \sqsubseteq (AMR)^{|\alpha|+|\delta|+1}$, which, by idempotence of AMR, is dominated by AMR. ∎

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 over $\{A, M, R\}$ of size smaller than Q can be obtained this way, it says the minimal equivalent query over $\{A, M, R\}$ 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, descendant and co-occurrence constraints. Then there is a unique query Q', obtainable by arbitrary applications of the operations A, R, and M to Q, which is equivalent to Q, and is minimal. Furthermore, Algorithm ACIM will always produce this minimal query.*

Proof: Follows from the preceding lemmas. ∎

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. Consequently, we have the following result.

Theorem 5.2 *The (worst-case) complexity of Algorithm 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 in the next subsection.

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 the query 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:[3] (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 $\tau - \boxed{\tau'}$; or (iv) leaf v of type τ' is a d-child of node u, which has a descendant[4] w of type τ, and $\mathcal{C}$ contains one of the ICs $\tau \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau'}$ or $\tau - \boxed{\tau'}$. Say that a leaf of a query is *locally redundant* precisely when one of the above conditions holds. While this procedure is incomplete in

[3] 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.

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

that it may not yield the minimal equivalent query, it has the advantage of being efficient and having the local minimality property, i.e., no leaf in the result is locally redundant.

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 right `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, $a\tilde{\tau}''$ 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.

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: Information content assignment and propagation are done as follows. As already pointed out, for each leaf, its information content is the type associated with it. For example, see Figure 6, top left. There is just one node of type t, and the information content t is associated to it.

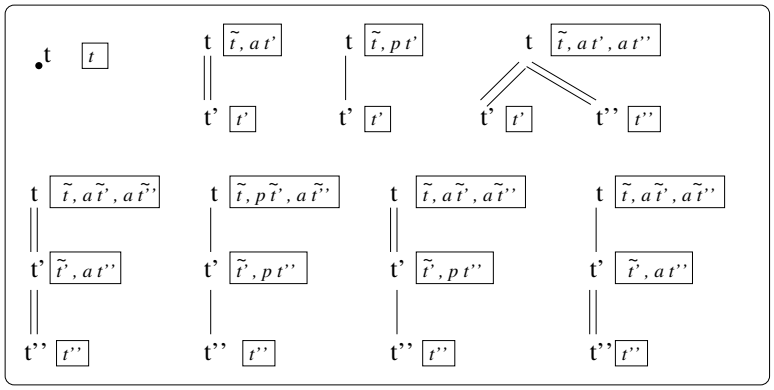


Figure 6: Information Propagation Examples

Edge	Parent Node	d-child/c-child Node	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 7: Information Propagation Rules

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 7. 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. Figure 6 contains several representative patterns that may be found in queries and illustrates how propagation of information content occurs.

Example 5.1 [Information Content Propagation]

Figure 8 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. The top left part of the figure (STEP 1) 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 t_6, its unconstrained type.[5] 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 7).

[5] Information contents appear in boxes.

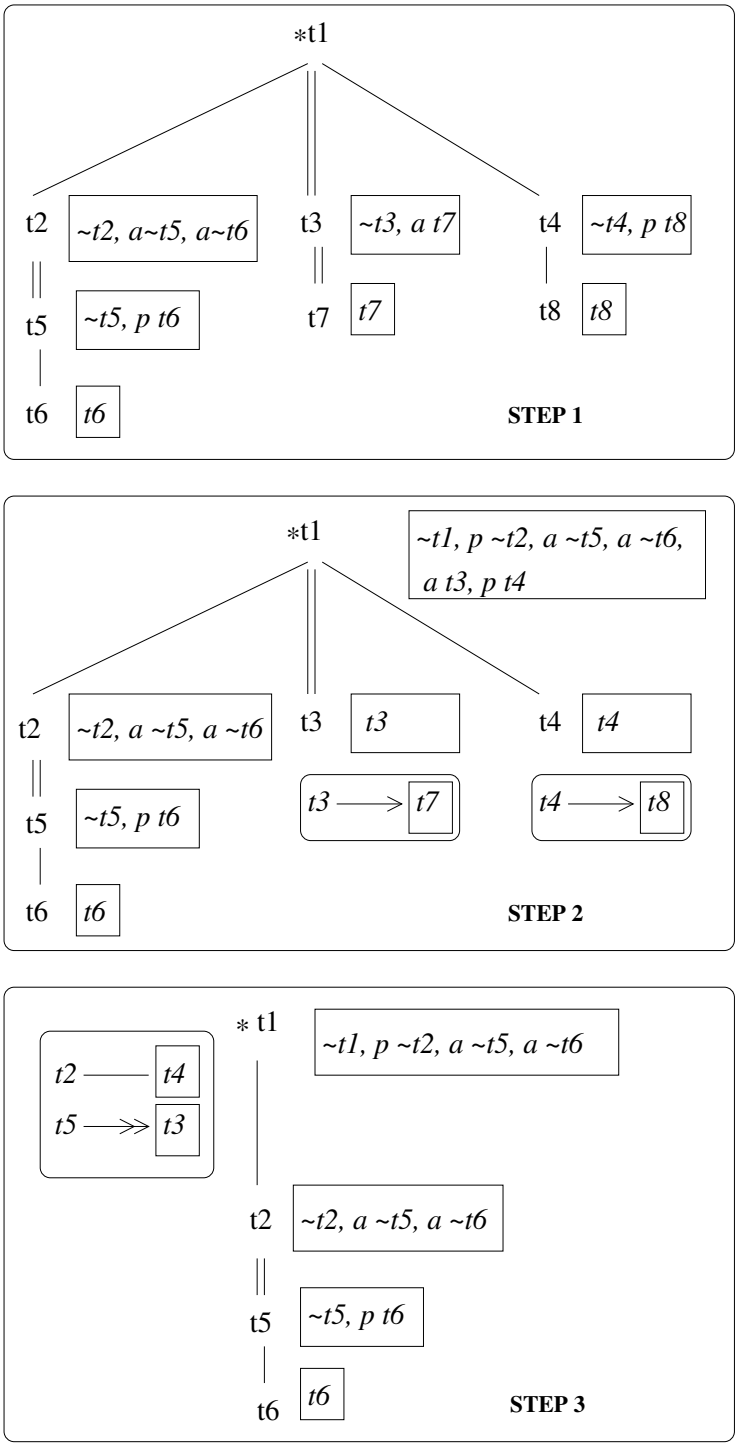


Figure 8: A CDM Example; ICs shown at point of application

Similarly, the d-parent of this node is of type t_2, and accordingly it gets the information content $\tilde{t}_2, a\tilde{t}_5, a\tilde{t}_6$ (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 of course 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 9. 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 $\tau_1 \longrightarrow\!\!\!\!\!\rightarrow \boxed{\tau_2}$.

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. See Figure 8, top left (STEP 1), which shows propagation completed along the three branches. The middle part 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 9), we note the c-child t_4 is redundant and can be removed. Similarly, the d-child t_3 can also be eliminated (Figure 8, 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 9 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 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 (which performs only local minimization) has a much better complexity than Algorithm ACIM (which performs global minimization).

Before leaving this section, we conclude with the following results.

Theorem 5.3 *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.*

Proof: The equivalence of Q' to Q is straightforward, as the soundness of Algorithm CDM is obvious. Suppose Q' has a leaf v that is locally redundant. This means one of the four rules in the beginning of Section 5.4 must hold. We illustrate just one case: rule (iv). Suppose leaf v of type τ' is a d-child of node u, which has a descendant w of type τ, and $\mathcal{C}$ contains one of the ICs $\tau \longrightarrow\!\!\!\!\rightarrow \boxed{\tau'}$ or $\tau - \boxed{\tau'}$. Further, suppose the IC that $\mathcal{C}$ contains is $\tau - \boxed{\tau'}$. In this case, the information content of w would initially include τ. It is easy to show that this would contribute to the information argument $a\tilde{\tau}$ at w, which would also include the argument $a\tau'$. The last rule in Figure 9 says the node v must be redundant — a contradiction. Other cases are similar. ∎

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

Theorem 5.4 *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 $\mathcal{C}$.*

Proof: The main intuition is that whatever nodes/edges are eliminated by the prior application of CDM would all be added "back" as temporary variants. Thus, if applying ACIM to the result of applying CDM does not find a leaf to be redundant, then this node will never be found to be redundant by applying ACIM directly to Q. ∎

One of the main contributions of this section has been the development of Algorithm CDM for finding a query equivalent to a given query that 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

Arg1	Arg2	Constraint	Minimization
1. $\tilde{\tau}_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
2. $\tilde{\tau}_1$	$p\tau_2$	$\tau_1 \longrightarrow \boxed{\tau_2}$	make τ_2 node redundant
3. $a\tau_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
4. $a\|p\tilde{\tau}_1$	$a\tau_2$	$\tau_1 \longrightarrow\!\!\!\!\rightarrow \boxed{\tau_2}$	make τ_2 node redundant
5. $a\|p\tau_1$	$a\|p\tau_2$	$\tau_1 - \boxed{\tau_2}$	make τ_2 node redundant
6. $a\|p\tilde{\tau}_1$	$a\|p\tau_2$	$\tau_1 - \boxed{\tau_2}$	make τ_2 node redundant

Figure 9: Constraint-Dependent Minimization Rules

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 Architecture: Use of Hash Tables

The architecture of our system is depicted in Figure 10. All the minimization work is done at the client side and the server only receives the minimized tree pattern to evaluate. A parse tree is first generated from the user query pattern. The core of our system is the minimization module that takes a parse tree and generates a minimized tree. It interacts with the schema repository that contains the set of constraints satisfied by the data instance on which the query is to be evaluated.

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.

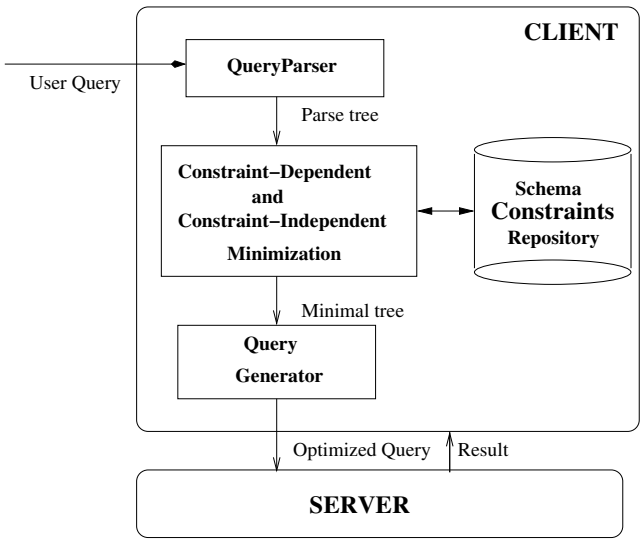
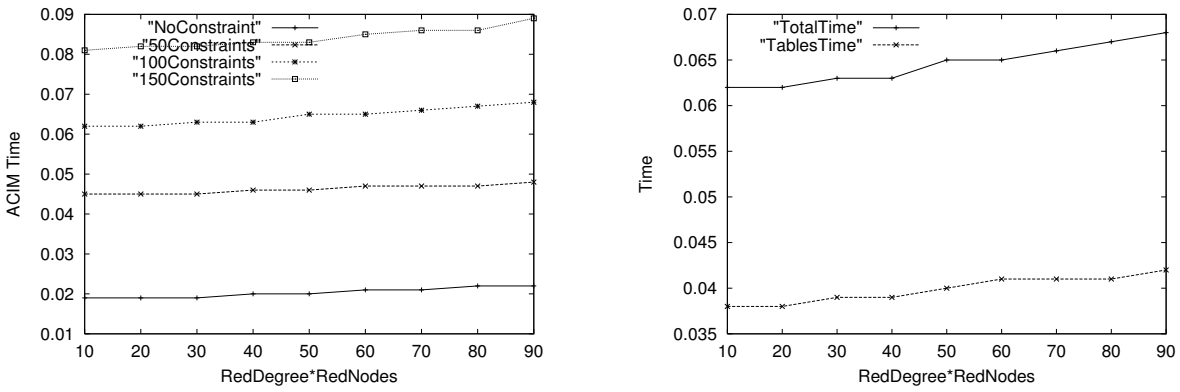


Figure 10: System Architecture

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 table. 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 vary the number of redundant nodes from 1 to 90 and the degree of redundancy from 1 to 40. We run ACIM with no constraints, 50, 100 and 150 constraints relevant to the query. The graphs of Figure 11(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 in time appears to be linear in the number of constraints.



(a) Varying Redundancy and Constraints

(b) Total Time and Tables Time

Figure 11: Studying ACIM

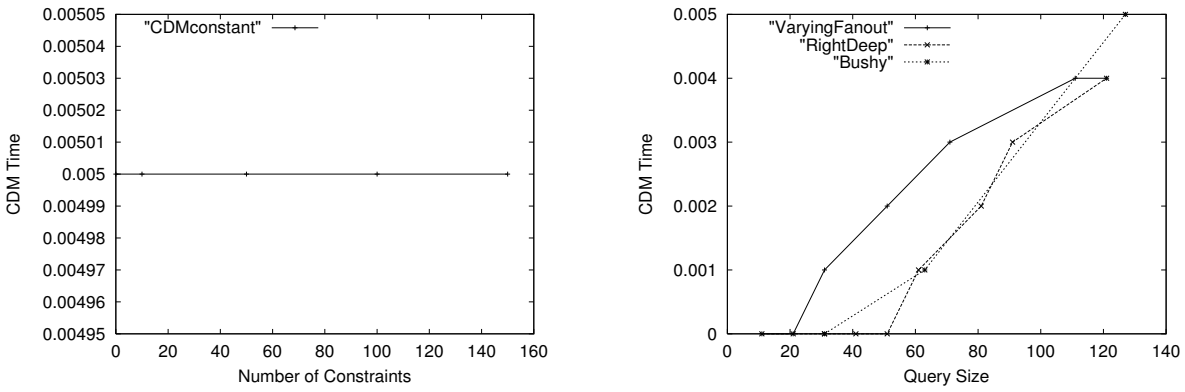
Total Time and Tables Time: We report the total time it takes to execute ACIM and the fraction of this time spent to build the images and the ancestor/descendant table. The query we consider has 101 nodes and the number of constraints relevant to this query is 100 (thus all nodes except the root node were redundant). The graph of Figure 11(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 12(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 12(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.



(a) Varying Constraints

(b) Varying Query Size

Figure 12: Studying CDM Time

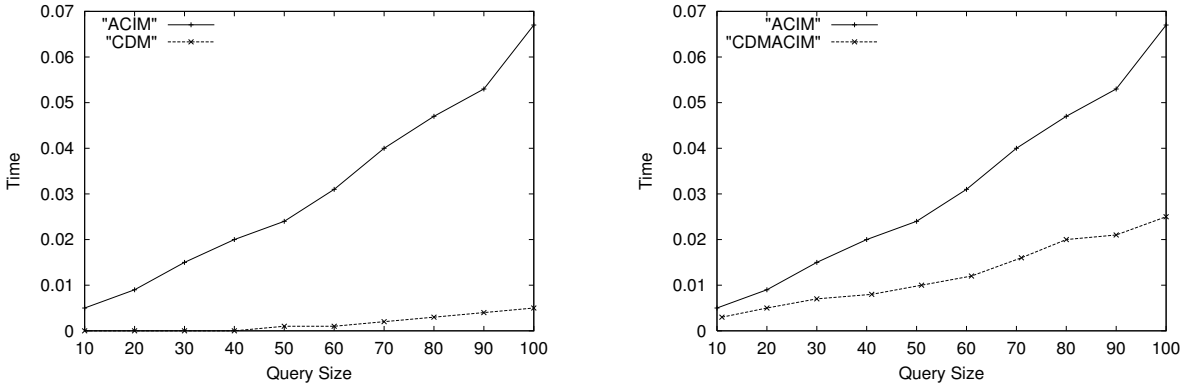
Because of the way the query is generated (all edges are redundant), the only node that remains after query minimization is the root node. The shape of a query does not have any impact on the CDM time. Right-deep and bushy tree pattern queries have very similar performance results. The experiments show 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 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 performances. 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 13(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



(a) ACIM and CDM with a Varying Query Size

(b) Varying Number of Redundant Nodes

Figure 13: Comparing/Combining ACIM and CDM

before running ACIM (on a potentially smaller query). We report on an experiment (in Figure 13(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 network 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. One question is how far can we enlarge the class of tree queries (e.g., viewed as a fragment of XPath) or the class of ICs considered while staying in PTIME, as far as containment and minimization are concerned. A fundamental difficulty that arises with larger classes of ICs is that intermediate nodes could become redundant without their descendants necessarily becoming so. A second source of difficulty comes from the fact that minimal equivalent queries may not be unique. What if integrity constraints involve value-based conditions? Our ongoing work addresses some of these

questions.

Acknowledgements

We would like to thank Dan Suciu and the reviewers of this paper for their insightful comments and suggestions, which helped improve this paper substantially.

References

[1] S. Abiteboul. Boundedness is undecidable for datalog programs with a single recursive rule. Information Processing Letters 32(6): 281-287 (1989).

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree patterns queries. *SIGMOD 2001.*

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C Working Draft. Available from http://www.w3.org/TR/xquery. 2001.

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

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

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

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

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

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

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

[11] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath fragments. *KRDB 2001.*

[12] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. *DBPL 2001.*

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

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

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

[16] 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.*

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

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

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

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

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

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

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

[24] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Macmillan Technical Publishing, Indianapolis, Indiana, 1999. Computer Science Press, Rockville, Maryland, 1989.

[25] P. T. Wood. On the equivalence of XML patterns. *DOOD 2000.*

[26] P. T. Wood. Minimizing simple XPath expressions. *WebDB 2001.*