Minimization of Tree Pattern Queries with Constraints

Ding Chen and Chee-Yong Chan
Dept. of Computer Science, School of Computing
National University of Singapore
{chending, chancy}@comp.nus.edu.sg

ABSTRACT

Tree pattern queries (TPQs) provide a natural and easy formalism to query tree-structured XML data, and the efficient processing of such queries has attracted a lot of attention. Since the size of a TPQ is a key determinant of its evaluation cost, recent research has focused on the problem of query minimization using integrity constraints to eliminate redundant query nodes; specifically, TPQ minimization has been studied for the class of forward and subtype constraints (FT-constraints). In this paper, we explore the TPQ minimization problem further for a richer class of FBST-constraints that includes not only FT-constraints but also backward and sibling constraints. By exploiting the properties of minimal queries under FBST-constraints, we propose efficient algorithms to both compute a single minimal query as well as enumerate all minimal queries. In addition, we also develop more efficient minimization algorithms for the previously studied class of FT-constraints. Our experimental study demonstrates the effectiveness and efficiency of query minimization using FBST-constraints.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - *Query processing*

General Terms

Algorithms, Performance

Keywords

XML, XPath, Tree pattern queries, Query minimization, Integrity constraints, Chase, Simulation

1. INTRODUCTION

Tree pattern queries (TPQs) provide a very natural and easy formalism to query XML data, and constitute a very useful and large fragment of queries expressible using XML query languages such as XPath [17] and XQuery [18]. Since the size of the query (in terms of the number of query steps) is a key determinant of its evaluation cost [9], there has been a lot of interest in the minimization of TPQs given knowledge of data constraints [19, 5, 14, 8]. The work in this area can be characterized along two main dimensions: the class of queries being supported (i.e., query fragment) and the types of constraints being considered. The various fragments of XPath queries explored so far can be denoted by XP^F [8], where $F \subseteq \{ /, //, [], * \}$ represents the set of query features supported including *child axis* "/", *descendant axis* "//", *nested predicates* "[]", and *wildcards* "$*$".

In terms of data constraints, besides the simplest case of query minimization without considering constraints, most of the work has focused primarily on *forward constraints* and *subtype constraints* [5, 14]. There are two types of forward constraints, namely, *required child (RC)* constraints and *required descendant (RD)* constraints. An RC (RD resp.) constraint is of the form $x \rightarrow y$ ($x \twoheadrightarrow y$ resp.) which states that for every element of type x, it has a child (proper descendant resp.) element of type y. A subtype constraint is of the form $x \leq y$ which states that every element of type x is also of type y.

In this paper, we examine the minimization of TPQs for the query fragment $XP^{\{/,//,[]\}}$ with respect to a richer class of data constraints that include not only forward and subtype constraints but also *backward* and *sibling* constraints.

Backward constraints, which are the "opposites" of forward constraints, can be classified into two types, namely, *required parent (RP)* constraints and *required ancestor (RA)* constraints. An RP (RA resp.) constraint is of the form $x \leftarrow y$ ($x \twoheadleftarrow y$ resp.) which states that for every element of type y, it has a parent (proper ancestor resp.) element of type x. For an example of a RP constraint, if an element 'b' appears as a sub-element only for the element 'a', then we have $a \leftarrow b$. For an example of a RA constraint, if an element 'd' appears as a sub-element only for elements 'b' and 'c', and both elements 'b' and 'c' appear as sub-elements only for element 'a', then we have $a \twoheadleftarrow d$.

A sibling constraint is of the form $a \overset{c}{\dashrightarrow} b$ which states that for every element of type a, if it has a child element of type c, then the a element must also have a child element of type b. Note that for a sibling constraint $a \overset{c}{\dashrightarrow} b$ to hold, it is not necessary to have $a \rightarrow b$ and $a \rightarrow c$. For example, the following are two possible DTD type definitions for an element 'a' that will result in the sibling constraint $a \overset{c}{\dashrightarrow} b$: (($c$?, b+)$*$, d) and ((b, c) | d).

Despite the fact that backward and sibling constraints have been largely neglected (with respect to query minimization), they are actually rather common in XML data. Table 1 compares the number of forward and backward constraints that are extracted from five different DTDs[1]. Observe that there are actually more backward constraints than forward constraints in these DTDs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

[1]Note that the number of forward and backward constraints indicated refer to the number of "basic" constraints that cannot be de-

XML Database	Forward		Backward	
	RC	RD	RP	RA
GraphML [7]	0	0	6	5
DBLP [11]	0	0	8	27
PSDML [1]	26	0	57	8
XMark [15]	57	0	57	14
Mondial [3]	13	0	30	8

Table 1: Forward & Backward Constraints in XML DTDs

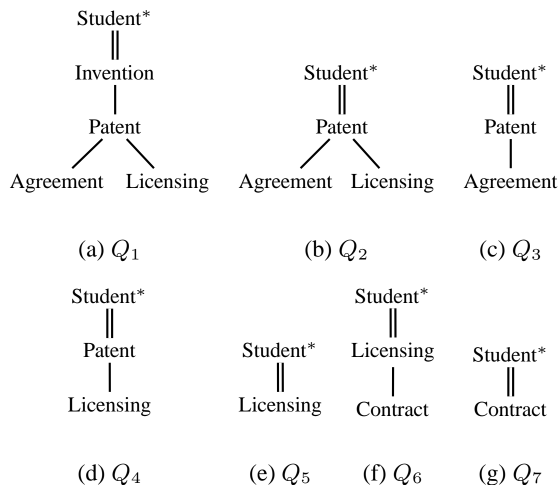


Figure 1: Minimization with Backward & Sibling Constraints

Furthermore, as pointed out by Bex, et al. [6] and Hinkelman [10], XML schemas (including industry-level standards) are generally too loosely defined with respect to the data that they actually represent. This means that XML data sets generally satisfy more constraints than what are explicitly specified in their schemas. We conducted a simple study to identify the sibling constraints from the data sets maintained at the XML Data Repository [11]. We found that there are 121, 6, and 3 sibling constraints, respectively, in the DBLP, Protein Sequence Database, and Mondial data sets[2], although there are only 0, 1 and 0 sibling constraints specified in their corresponding DTDs respectively. As an example, in the DBLP data set, the '$phdthesis$' element does not always have a '$publisher$' sub-element. However, if a '$phdthesis$' element has an '$isbn$' sub-element, then the '$phdthesis$' element must also have a '$publisher$' sub-element. Although the sibling constraint $phdthesis \stackrel{isbn}{\dashrightarrow} publisher$ is not explicitly captured by DBLP's DTD, this constraint is indeed satisfied by the DBLP data sets [11].

By considering a richer class of constraints, there are more opportunities for query minimization. As an example, consider the minimization of the TPQ Q_1 shown in Figure 1(a) w.r.t. the set of constraints $C = \{\ Patent \stackrel{Agreement}{\dashrightarrow} Licensing, Patent \stackrel{Licensing}{\dashrightarrow} Agreement, Patent \leftarrow Licensing, Invention \leftarrow Patent, Patent \rightarrow Title, Licensing \rightarrow Contract, Licensing \leftarrow Contract\ \}$. Each node in Q_1 represents an element type, and the special node that is marked with a * (i.e., *Student*) represents the output node of the query. Thus Q_1 will return all *Student* elements that satisfy a set of requirements specified by the edges. A single (double) edge connecting two nodes represents a parent-child (ancestor-descendant) relationship. Thus, two of the requirements specified by Q_1 are that each *Student* element must have a descendant *Invention* element, and each *Invention* element must have a child *Patent* element. Observe that Q_1 cannot be further minimized using only the forward constraints in C. However, using the RP constraint *Invention* ← *Patent*, Q_1 can be simplified to Q_2 (Figure 1(b)). Moreover, by applying the sibling constraint *Patent* $\stackrel{Agreement}{\dashrightarrow}$ *Licensing*, Q_2 can be further minimized to Q_3 (Figure 1(c)), which turns out to be a minimal query (w.r.t. C).

Query minimization using backward and/or sibling constraints is a more challenging problem due to two new properties of minimal queries under such constraints. First, the minimal query is not necessarily unique; and moreover, the minimal queries do not necessarily have the same size. Second, a minimal query can contain element types that are not present in the input query. In contrast, the minimal query (with respect to only forward and subtype constraints) is always unique, and the element types appearing in the minimal query is a subset of those in the input query [5, 14].

Referring again to the example in Figure 1, Q_1 actually has three minimal queries: besides Q_3, Q_5 and Q_7 are also minimal queries of Q_1. Note that the sizes of Q_3 and Q_5 are different, and Q_7 contains the element type *Contract* that is absent in Q_1. The second minimal query Q_5 can be derived from Q_2 as follows: first, apply the sibling constraint *Patent* $\stackrel{Licensing}{\dashrightarrow}$ *Agreement* to minimize Q_2 to Q_4; next, apply the RP constraint *Patent* ← *Licensing* to simplify Q_4 to Q_5. The third minimal query Q_7 is obtained from Q_5 as follows: first, note that Q_5 is equivalent to Q_6 due to the RC constraint *Licensing* → *Contract*; next, note that Q_6 is equivalent to Q_7 due to the RP constraint *Licensing* ← *Contract*.

For notational convenience, we use the letters F, B, S, and T to represent, respectively, the class of forward, backward, sibling, and subtype constraints. In addition, we use α-constraints to denote the class of constraints of types in α, where $\alpha \subseteq \{F, B, S, T\}$; braces and commas in α are omitted for simplicity.

The key results of our paper are summarized in Figure 2. Figure 2(a) compares the key properties of minimal queries (columns 2 to 4) for different classes of data constraints (column 1): our new results are indicated in rows 2 to 4, while the results from previous work [5, 14] are indicated in row 1. The lattice structure in Figure 2(b) summarizes the time-complexity of query minimization for $XP^{\{/,//,[]\}}$ under different constraint classes (represented by the lattice nodes), where the lattice edges represent the containment relationship between constraint classes. Specifically, the time complexity shown for each class is for computing one minimal query of an input query Q w.r.t. a set of constraints C, where n denotes the number of steps in Q, and Σ denotes the set of distinct element types in C.

The rest of the paper is organized as follows. Section 2 covers background material, and related work is presented in Section 3. Section 4 introduces key concepts for query minimization. Section 5 presents properties of minimal queries under different classes of constraints. Section 6 presents efficient algorithms to generate a minimal query and to enumerate all the minimal queries under FBST-constraints. Section 7 presents a more efficient minimization algorithm for FT-constraints. Section 8 covers the minimization algorithms and results for the remaining subclasses of FBST-constraints. Section 9 presents an experimental evaluation of our proposed algorithms. Finally, Section 10 concludes the paper.

2. BACKGROUND

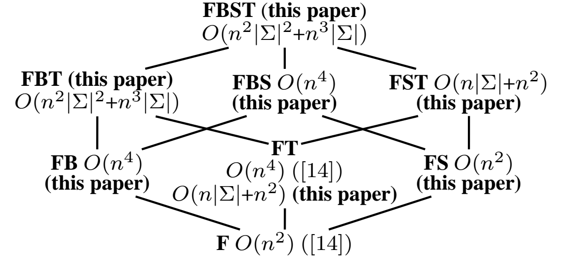
Tree pattern queries. As illustrated in Figure 1, tree pattern

rived from other constraints.

[2]Again here, we counted only the "basic" sibling constraints that cannot be derived from other constraints.

Data constraints	Number of minimal queries	Element types in minimal queries	How do minimal queries differ?
F / FT	One	Subset of original query's types	-
FB / FBT	Possibly multiple minimal queries of the same size	Can contain element types that are not present in original query	ad-leaf nodes
FS / FST			pc-leaf nodes
FBS / FBST	Possibly multiple minimal queries of different sizes		pc/ad-leaf nodes

(a) Properties of Minimal Queries



(b) Time Complexity of TPQ minimization
(n = size of query, Σ = set of element types in constraints)

Figure 2: Summary of Key Results (F = forward, B = backward, S = sibling, T = subtype)

queries (TPQs) are represented as trees, where the nodes of a TPQ Q are labelled by element types from a finite alphabet Σ. The type of a node u is denoted by $\tau(u)$, and the root node of Q is denoted by *root*(Q). The size of Q, denoted by $|Q|$, refers to the number of nodes in Q. Each query Q has a unique *output node*, denoted as $op(Q)$, and its element type label is distinguished with a * mark. The nodes in Q are connected by two types of edges: parent-child edges (pc-edges) and ancestor-descendant edges (ad-edges). Consider an edge $e = (u, v)$ with parent node u and child node v. If e is a α-edge, where $\alpha \in \{pc, ad\}$, we say that v is a α-child of u and u is the α-parent of v. Moreover, if v is a leaf node, v is also known as a α-leaf node.

An embedding of a TPQ Q onto a tree database db is defined as a mapping β from the nodes of Q to the nodes of db such that the following conditions are satisfied:

1. Preserve node types: for each node $u \in Q$, either u and $\beta(u)$ are of the same type, or $\beta(u)$ is of a subtype of u;

2. Preserve pc/ad-edge relationships: if v is a pc-child (ad-child resp.) of u in Q, then $\beta(v)$ is a child (descendant resp.) of $\beta(u)$in db.

The evaluation of a TPQ Q on db requires finding all the embeddings of Q in db, and the answer to Q is given by the set of database nodes $\beta(op(Q))$.

Minimal queries. A node u of a TPQ Q is *redundant* if u is a non-output node and the query obtained by deleting u from Q is equivalent to Q. Here, deleting a leaf node u simply removes u and its incident edge; while deleting an internal node u requires removing u and connecting u's parent node (if it exists) to each of u's child nodes with an ad-edge.

Given two TPQs Q and Q', and a set of integrity constraints (ICs) C, Q and Q' are *equivalent* w.r.t. C, denoted by $Q \equiv_C Q'$, if and only if Q and Q' have the same answer on all tree databases that satisfy C. Q' is a *minimal query* of Q w.r.t. C iff (1) $Q \equiv_C Q'$ and (2) $Q \not\equiv_C Q''$ for every Q'' that is obtained by deleting some node(s) from Q'.

Notations. In this paper, we use Σ to denote the set of distinct element types in the constraints, and n to denote the size of Q.

3. RELATED WORK

Sihem et al. were the first to study the TPQ minimization problem for the query fragment $XP^{\{/,//,[]\}}$ [5]. The state-of-the-art minimization algorithms for this fragment have time complexities of $O(n^2)$ and $O(n^4)$, respectively, for the case without constraints and the case with FT-constraints [14]. Moreover, for both cases, every TPQ has a unique minimal query [5, 14]. To the best of our knowledge, TPQ minimization for $XP^{\{/,//,[]\}}$ using backward or sibling constraints has not been explored.

Query minimization for the full query fragment $XP^{\{/,//,[],*\}}$ in the absence of constraints was shown by Flesca et al. to be NP-hard [8]. For fragment $XP^{\{/,[],*\}}$, query minimization in the absence of constraints has polynomial time complexity, and each XPath query has a unique minimal query [19].

A related direction is query containment [12, 13, 20]. The containment problem under DTDs for a smaller XPath fragment $XP^{\{/,[]\}}$ was shown to be *co*NP-complete [13].

4. REASONING WITH CONSTRAINTS

This section introduces the key concepts for TPQ minimization under FBST-constraints.

4.1 Constraint Closure

Given a set of FBST-constraints C, let *closure*(C) denote the set of FBST-constraints that must hold w.r.t. C. The following set of inference rules (R1 to R22) can be used to compute $closure(C)$ as follows: first, initialize $closure(C)$ to be C, and then iteratively add new constraints that are generated by the rules to $closure(C)$ until no further rules can be added.

R1. *if* $\tau_1 \rightarrow \tau_2$, *then* $\tau_1 \twoheadrightarrow \tau_2$
R2. *if* $\tau_1 \twoheadrightarrow \tau_2$ *and* $\tau_2 \twoheadrightarrow \tau_3$, *then* $\tau_1 \twoheadrightarrow \tau_3$
R3. *if* $\tau_1 \leq \tau_2$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_1 \leq \tau_3$
R4. *if* $\tau_1 \leq \tau_2$ *and* $\tau_2 \rightarrow \tau_3$, *then* $\tau_1 \rightarrow \tau_3$
R5. *if* $\tau_1 \leq \tau_2$ *and* $\tau_2 \twoheadrightarrow \tau_3$, *then* $\tau_1 \twoheadrightarrow \tau_3$
R6. *if* $\tau_1 \rightarrow \tau_2$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_1 \rightarrow \tau_3$
R7. *if* $\tau_1 \twoheadrightarrow \tau_2$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_1 \twoheadrightarrow \tau_3$
R8. $\tau_i \leq \tau_i$ *for every* $\tau_i \in \Sigma$
R9. *if* $\tau_2 \leftarrow \tau_1$, *then* $\tau_2 \twoheadleftarrow \tau_1$
R10. *if* $\tau_2 \twoheadleftarrow \tau_1$ *and* $\tau_3 \twoheadleftarrow \tau_2$, *then* $\tau_3 \twoheadleftarrow \tau_1$
R11. *if* $\tau_1 \twoheadrightarrow \tau_2$ *and* $\tau_3 \leftarrow \tau_2$ *and* $\tau_1 \not\leq \tau_3$, *then* $\tau_1 \twoheadrightarrow \tau_3$
R12. *if* $\tau_1 \rightarrow \tau_2$ *and* $\tau_3 \twoheadleftarrow \tau_2$ *and* $\tau_1 \not\leq \tau_3$, *then* $\tau_3 \twoheadleftarrow \tau_1$
R13. *if* $\tau_1 \leq \tau_2$ *and* $\tau_3 \leftarrow \tau_2$, *then* $\tau_3 \leftarrow \tau_1$
R14. *if* $\tau_1 \leq \tau_2$ *and* $\tau_3 \twoheadleftarrow \tau_2$, *then* $\tau_3 \twoheadleftarrow \tau_1$
R15. *if* $\tau_2 \leftarrow \tau_1$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_3 \leftarrow \tau_1$
R16. *if* $\tau_2 \twoheadleftarrow \tau_1$ *and* $\tau_2 \leq \tau_3$, *then* $\tau_3 \twoheadleftarrow \tau_1$
R17. *if* $\tau_1 \rightarrow \tau_2$, $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$, *then* $\tau_1 \rightarrow \tau_3$
R18. *if* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$, $\tau_1 \overset{\tau_3}{\dashrightarrow} \tau_4$ *then* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_4$
R19. *if* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$, $\tau_3 \leq \tau_4$, *then* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_4$
R20. *if* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$, $\tau_4 \leq \tau_1$, *then* $\tau_4 \overset{\tau_2}{\dashrightarrow} \tau_3$
R21. *if* $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$, $\tau_4 \leq \tau_2$, *then* $\tau_1 \overset{\tau_4}{\dashrightarrow} \tau_3$
R22. *if* $\tau_1 \rightarrow \tau_2$, *then* $\tau_1 \overset{\tau_i}{\dashrightarrow} \tau_2$ *for every* $\tau_i \in \Sigma$

In each rule, τ_1, τ_2, τ_3, and τ_4 represent distinct element types. We write $\tau_1 \not\leq \tau_3$ to mean that τ_1 is not a subtype of τ_3. Rules R1 to R8 were used earlier in [14] for query minimization with FT-constraints. Rules R9 to R16 are new rules to handle backward constraints, while rules R17 to R22 are new rules to handle sibling constraints.

Rule R11 follows from the tree structure property: if a node u_1 of type τ_1 has a descendant u_2 of type τ_2, and u_2 in turn has a parent u_3 of type τ_3, then u_3 must be a descendant of u_1 provided that τ_1 is not a subtype of τ_3. Otherwise, if $\tau_1 \leq \tau_3$, then u_3 may not be a descendant of u_1 as u_3 and u_1 can be the same node. A similar reasoning applies to rule R12. The rest of the rules are straight-forward.

4.2 Constraint Graph

The FBS-constraints in $closure(C)$ can be categorized into *trivial* and *non-trivial* constraints defined as follows. Every RC and RP constraint is non-trivial. An RD or RA constraint is trivial if it can be inferred from other constraints using rules that do not involve subtype constraints; otherwise it is non-trivial. A sibling constraint is trivial if it can be inferred using rule R22; otherwise it is non-trivial.

Constraint graph. The non-trivial FBS-constraints in $closure(C)$ can be represented succinctly by a *constraint graph*, denoted by $G_C = (V_C, E_C)$, where each node in V_C represents some element type in Σ and each edge in E_C represents a non-trivial constraint. Specifically, if $\tau_1 \rightarrow \tau_2, \tau_1 \leftarrow \tau_2, \tau_1 \twoheadrightarrow \tau_2, \tau_1 \twoheadleftarrow \tau_2$, or $\tau_1 \overset{\tau_3}{\dashrightarrow} \tau_2$, is a non-trivial constraint, then G_C contains, respectively, an edge $\tau_1 \rightarrow \tau_2$ (a *c-edge*), $\tau_1 \leftarrow \tau_2$ (a *p-edge*), $\tau_1 \twoheadrightarrow \tau_2$ (a *d-edge*), $\tau_1 \twoheadleftarrow \tau_2$ (an *a-edge*), or $\tau_1 \overset{\tau_3}{\dashrightarrow} \tau_2$ (an *s-edge*).

Thus, there are five types of edges in G_C: a-, c-, d-, and p-edges (all represented by solid arrows) denote RA, RC, RD, and RP constraints, respectively; and s-edges (represented by labelled dashed arrows) denote sibling constraints. We refer to τ_3 as the *edge label* of an s-edge $\tau_1 \overset{\tau_3}{\dashrightarrow} \tau_2$.

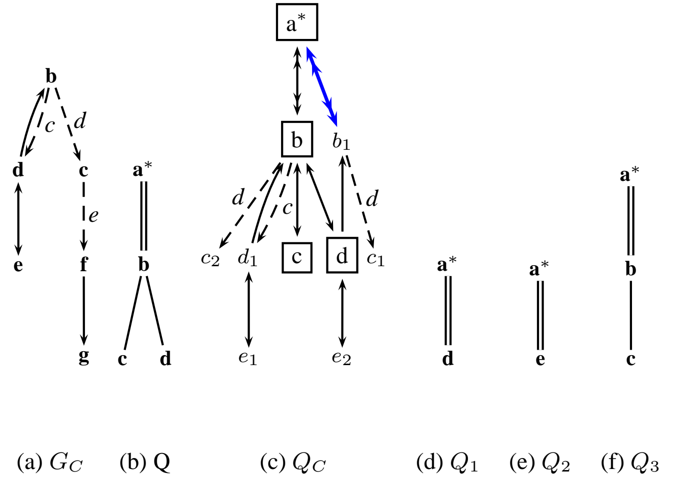
In all the above five edge types, we say that τ_1 is the *parent* of τ_2; or equivalently, τ_2 is the *child* of τ_1. Note that in the edge specification, τ_1 is on the left side of the arrow and τ_2 is on the right side of the arrow. Given two nodes τ_1 and τ_k in G_C, we say that τ_1 is an *ancestor* of τ_k (or equivalently, τ_k is a *descendant* of τ_1) if there is a sequence of nodes $\tau_1, \tau_2, \cdots, \tau_k$ in G_C such that τ_i is the parent of τ_{i+1} for $i \in [1, k)$. Graphically, each edge in G_C is depicted with the parent node shown above the child node.

Consider an edge e with parent node τ_1 and child node τ_2. If e is an ℓ-edge, where $\ell \in \{a, c, d, p, s\}$, we say that τ_1 is a *ℓ-parent* of τ_2, and that τ_2 is a *ℓ-child* of τ_1.

To avoid cluttering G_C, if two nodes in G_C are connected by two solid edges (which must necessarily represent one forward and one backward constraint), then these two edges are combined and represented as a single, double-headed solid arrow. For example, if G_C contains both edges $\tau_1 \rightarrow \tau_2$ and $\tau_1 \leftarrow \tau_2$, then they can simply be represented by a single edge $\tau_1 \leftrightarrow \tau_2$. In this case, τ_1 is a p-parent as well as a c-parent of τ_2; τ_2 is a p-child as well as a c-child of τ_1.

Following [14], we assume that C is available as part of the data, and therefore both $closure(C)$ and G_C are computed only once offline. The closure of C, $closure(C)$, can be computed in $O(|\Sigma|^2)$. The size of $closure(C)$ is $O(|\Sigma|^2)$. G_C consists of $O(|\Sigma|)$ nodes and $O(|\Sigma|^2)$ edges.

Reachable subgraph. Given $\tau_i \in V_C$ and $L \subseteq \Sigma$, we define $G^L_{\tau_i} = (V^L_{\tau_i}, E^L_{\tau_i})$ to be the *reachable subgraph* of node τ_i in G_C, where $V^L_{\tau_i} \subseteq V_C$ and $E^L_{\tau_i} \subseteq E_C$ is the set of edges induced by $V^L_{\tau_i}$ in G_C. $V^L_{\tau_i}$ is defined as follows: $\tau_j \in V^L_{\tau_i}$ if one of the following conditions hold:



(a) G_C (b) Q (c) Q_C (d) Q_1 (e) Q_2 (f) Q_3

Figure 3: Minimization with FBST-Constraints

R1. $\tau_j = \tau_i$; or

R2. $\tau_k \in V^L_{\tau_i}$ and there is an a/c/d/p-edge from τ_k to τ_j in G_C; or

R3. $\tau_i \overset{t}{\dashrightarrow} \tau_j$ is in G_C and $t \in L$; or

R4. $\tau_k \overset{t}{\dashrightarrow} \tau_j$ is in G_C, $\{\tau_k, t\} \subseteq V^L_{\tau_i}$, and t is a c/p/s-child of τ_k in G_C.

Intuitively, a reachable subgraph $G^L_{\tau_i}$ represents all the nodes in G_C that are reachable from node τ_i by traversing the edges in G_C; $L \subseteq \Sigma$ can be used as edge labels in the traversal of s-edges (condition R3).

PCC-pair. For each s-edge $\tau_1 \overset{\tau_2}{\dashrightarrow} \tau_3$ in G_C, we refer to the type pair (τ_1, τ_2) as a *parent-conditional-child pair (PCC-pair)* with parent type τ_1 and conditional child type τ_2.

Type & PCC-pair Equivalence. For each type $\tau_i \in V_C$, we define $G^{\emptyset}_{\tau_i}$ to be the reachable subgraph for τ_i; and for each PCC-pair (τ_i, τ_j) in G_C, we define $G^{\{\tau_j\}}_{\tau_i}$ to be the reachable subgraph for (τ_i, τ_j).

Let X and Y be a type in V_C or a PCC-pair in G_C. We say that X and Y are *equivalent*, denoted by $X \equiv Y$, if the reachable subgraphs for X and Y are equal. We use $[X]$ to denote the equivalence class for X (based on type/PCC-pair equivalence); i.e., $[X] = \{Y \mid Y$ is a type in V_C or a PCC-pair in $G_C, Y \equiv X\}$.

Given an equivalence class $[X]$, a member $Y \in [X]$ is defined to be *minimal* if (1) Y is a type; or (2) Y is a PCC-pair (τ_i, τ_j) such that $\tau_i \notin [X]$ and $\tau_j \notin [X]$. An equivalence class $[X]$ is a *trivial equivalence class* if $[X]$ has only one minimal member; otherwise, $[X]$ is a *non-trivial equivalence class*.

The concept of an equivalence class is very fundamental in our TPQ minimization approach as it is used to characterize important properties of multiple minimal queries in Section 5.

Example 4.1 Consider $\Sigma = \{a, b, c, d, e, f, g\}$ with a set of constraints $C = \{b \overset{c}{\dashrightarrow} d,\ b \overset{d}{\dashrightarrow} c,\ d \rightarrow e,\ d \twoheadrightarrow e,\ d \overset{f}{\dashrightarrow} e,\ d \leftarrow e,\ b \leftarrow d,\ b \twoheadleftarrow e,\ f \rightarrow g,\ c \overset{e}{\dashrightarrow} f\}$. The set of non-trivial constraints $C' = \{b \overset{c}{\dashrightarrow} d,\ b \overset{d}{\dashrightarrow} c,\ d \rightarrow e,\ d \leftarrow e,\ b \leftarrow d,\ f \rightarrow$

$g,\ c \overset{e}{\dashrightarrow} f\}$. The constraint graph G_C built from C' is shown in Figure 3(a). G_C has a c-edge (p-edge resp.) from d to e (b resp.); thus $e \in V_d^{\emptyset}$ ($b \in V_d^{\emptyset}$ resp.). Now $b \in V_d^{\emptyset}$ has an s-child $d \in V_d^{\emptyset}$, and $b \overset{d}{\dashrightarrow} c \in G_C$; thus $c \in V_d^{\emptyset}$. Therefore, we have $V_d^{\emptyset} = \{b, c, d, e\}$. Similarly, $V_e^{\emptyset} = V_b^{\{c\}} = V_b^{\{d\}} = \{b, c, d, e\}$. We have an equivalence class $\{(b, c), (b, d), d, e\}$, of which (b, c), d and e are the only minimal members. □

5. PROPERTIES OF MINIMAL QUERIES

Previous work on TPQ minimization has shown that for F/FT-constraints, each query has a unique minimal query [5, 14]. However, beyond these results for F/FT-constraints, there has not been any systematic study and characterization of the properties of minimal queries.

In this section, we characterize important properties of minimal queries under various subclasses of FBST-constraints. First, we present a necessary condition for the existence of multiple minimal queries under FBST-constraints. Then, for each of the constraint classes FBT, FST, and FBST, we characterize the conditions for a query to have multiple minimal queries.

The following result states a necessary condition for the existence of multiple minimal queries under FBST-constraints.

PROPOSITION 5.1. *Consider the minimization of a query Q under a set of FBST-constraints C. A necessary condition for Q to have multiple minimal queries is the existence of a non-trivial equivalence class in G_C.*

The intuition behind Proposition 5.1 is as follows. Consider two distinct minimal queries Q_m and Q'_m for a query Q. Since Q_m and Q'_m are distinct, each minimal query must contain some components that are different from each other. Let C_m and C'_m denote these components of Q_m and Q'_m, respectively. However, since Q_m and Q'_m are equivalent, the reachable subgraphs of these components must be the same; i.e., they must both belong to some equivalence class $[X]$ in G_C. Furthermore, both C_m and C'_m are necessarily minimal members of $[X]$ given that Q_m and Q'_m are minimal queries. Thus, it follows that $[X]$ must be a non-trivial equivalence class in G_C.

The following result characterizes minimal queries under FBT-constraints.

PROPOSITION 5.2. *Let Q_m be a minimal query of Q under a set of FBT-constraints C. Then Q has another distinct minimal query Q'_m iff the following two conditions hold:*

1. *Q_m has an ad-edge (p, x), where x is a non-output leaf node; and*
2. *there exists $\tau_y \in [\tau(x)]$ such that*
   - *(a) $\tau(x) \leftarrow \tau_y \in G_C$ and $\tau(p) \not\leq \tau(x)$; or*
   - *(b) $\tau_y \leftarrow \tau(x) \in G_C$ and $\tau(p) \not\leq \tau_y$.*

Under FBT-constraints, the absence of sibling constraints means that each equivalence class contains only element types.

Based on Proposition 5.2, if Q_m is a minimal query of Q under a set of FBT-constraints C, then another minimal query of Q can be derived from Q_m by changing node x to a τ_y-node. Furthermore, all the minimal queries of Q must be of the same size and they differ only in their ad-leaf nodes.

Example 5.1 Consider $C = \{c \leftarrow d,\ c \rightarrow d,\ a \rightarrow e,\ e \rightarrow f,\ e \leftarrow f\}$. We have the two non-trivial equivalence classes in G_C: $\{c, d\}$ and $\{e, f\}$. Consider TPQ Q in Figure 4(a) and a minimal query Q_1 of Q (w.r.t. C) in Figure 4(b). Since Q_1 has a non-output ad-leaf node c, and there is a type $d \in [c]$ such that $c \leftarrow d$ and $a \not\leq c$, by Proposition 5.2, Q has multiple minimal queries. Indeed, Q_2 in Figure 4(c) is another minimal query of Q that is obtained from Q_1 by changing c to d. Note that if C had an additional constraint $a \leq c$, then Q_2 would not be a minimal query of Q. To see this, consider a single data chain <a><d/><a/>. Clearly, this data (which satisfies C) is an answer to Q_2, but not to Q_1. □

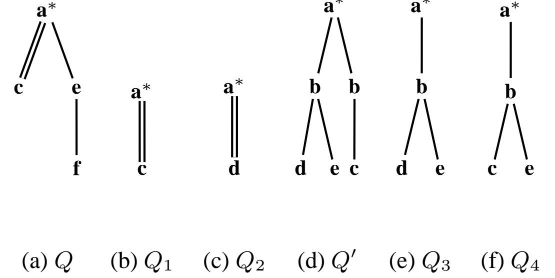


Figure 4: Illustration of Properties of Minimal Queries

The following result characterizes minimal queries under FST-constraints.

PROPOSITION 5.3. *Let Q_m be a minimal query of Q under a set of FST-constraints C. Then Q has another distinct minimal query Q'_m iff all the following conditions hold:*

1. *Q_m has an pc-edge (p, x), where x is a non-output leaf node; and*
2. *there exists $(\tau(p), \tau_y) \in [(\tau(p), \tau(x))]$, where $\tau_y \in \Sigma$ and $\tau_y \neq \tau(x)$.*

Under FST-constraints, the absence of backward constraints implies that each equivalence class contains only PCC-pairs.

Based on Proposition 5.3, if Q_m is a minimal query of Q under a set of FST-constraints C, then another minimal query of Q can be derived from Q_m by changing node x to a τ_y-node. Furthermore, all the minimal queries for Q must be of the same size and they differ only in their pc-leaf nodes.

Example 5.2 Consider $C = \{b \overset{c}{\dashrightarrow} d, b \overset{d}{\dashrightarrow} c\}$. We have $(b, c) \equiv (b, d)$. TPQ Q_3 (Figure 4(e)) is a minimal query of TPQ Q' (Figure 4(d)). Since Q_3 has a non-output pc-leaf d, and G_C has an equivalence class $[(b, d)]$ where $(b, c) \in [(b, d)]$, by Proposition 5.3, Q' has another minimal query Q_4 (Figure 4(f)), which is obtained from Q_3 by changing d to c. □

Finally, the following result characterizes minimal queries under FBST-constraints.

PROPOSITION 5.4. *Let Q_m be a minimal query of Q under a set of FBST-constraints C. Then Q has another distinct minimal query Q'_m iff one of the following conditions hold:*

1. *Q_m satisfies conditions 1 and 2 of Proposition 5.2;*
2. *Q_m satisfies conditions 1 and 2 of Proposition 5.3;*
3. *Q_m satisfies all of the following conditions:*
   - *(a) Q_m has an ad-edge (p, x), where x is a non-output leaf node; and*

Algorithm 1 SingleMinimizeFBST (Q)

1: compute the chase query Q_C
2: compute *Fsim*(u) and *FBsim*(u) for each query node $u \in Q_C$
3: Q_C = *ChaseMinimizeFBST*$(op(Q_C))$
4: delete the remaining chase nodes in Q_C to obtain a minimal query Q_m

---

(b) there exists a minimal member $(\tau_y, \tau_z) \in [\tau(x)]$, where $\tau_y \in \Sigma$ and $\tau_z \in \Sigma$.

4. Q_m satisfies all of the following conditions:

(a) Q_m has an pc-edge (p, x), where x is a non-output leaf node;

(b) p is an ad-child; and

(c) there exists some element type $\tau_i \in [(\tau(p), \tau(x))]$.

Under FBST-constraints, an equivalence class may contain both single types as well as PCC-pairs.

Based on Proposition 5.4, if Q_m is a minimal query of Q under a set of FBST-constraints C, then another minimal query Q'_m of Q can be derived from Q_m as follows. If Q_m satisfies condition 1 or 2, then Q'_m can be derived from Q_m by changing node x to a τ_y-node. If Q_m satisfies condition 3, then Q'_m can be derived from Q_m by changing node x to a pc-edge (τ_y, τ_z). Finally, if Q_m satisfies condition 4, then Q'_m can be derived from Q_m by changing the pc-edge (p, x) to a τ_i-node. Furthermore, if Q has multiple minimal queries, their sizes could differ.

Example 5.3 (continued from Example 4.1) TPQ Q_1 (Figure 3(d)) is a minimal query of TPQ Q (Figure 3(b)). From Example 4.1, we have $[d] = \{d, e, (b, d), (b, c)\}$. By the condition set 1 of Proposition 5.4 (Q_1 has a non-output ad-leaf d; $e \in [d]$ and $d \leftarrow e$), Q has distinct minimal queries. By changing ad-leaf $d \in Q_1$ to minimal members of $[d]$, e or (b, c), we have two other minimal queries Q_2 (Figure 3(e)) and Q_3 (Figure 3(f)). ☐

6. MINIMIZE WITH FBST-CONSTRAINTS

In this section, we present algorithms to minimize TPQs under the broadest class of FBST-constraints. We first present techniques for computing a single minimal query (Sections 6.1 to 6.4), and then extend the approach to enumerate all minimal queries (Section 6.5).

Our overall approach to compute a single minimal query for a TPQ Q (w.r.t. a set of constraints C) is shown in Algorithm 1 and consists of four main steps. The first step is to compute the chase query Q_C of the input query; the goal is to integrate the relevant constraints from C into Q to created an augmented query Q_C that contains additional chase nodes. The details of the chase query computation are discussed in Sections 6.1 and 6.3. The second step (Section 6.2) is to compute simulation relations for the nodes in Q_C; the purpose of this step is to enable the detection of redundant nodes in Q to generate a minimal query of Q, which is performed in the third step (Section 6.4). Finally, the fourth step simply removes any remaining chase nodes from Q_C to obtain a minimal query of Q.

While our overall approach follows the same principle as the previous work for query minimization with FT-constraints [14], the TPQ minimization problem with FBST-constraints is a more challenging task due to the intricacies of dealing with BS-constraints which requires the development of several new techniques:

1. The computation of $closure(C)$ requires new inference rules to handle BS-constraints (R9-R21 presented in Section 4.1).

2. When building Q_C, the presence of B-constraints requires a more intricate augmented chase computation (i.e., the ad-edge augmentation in Section 6.3) to address the inadequacy of the conventional chase computation (Section 6.1). Moreover, the detection of redundant nodes now requires computing forward & backward simulation which is more involved rather than computing forward simulation.

3. The presence of S-constraints, which represent conditional constraints, also demands some subtle extensions to the chase and minimization process (Sections 6.1 and 6.4).

4. The presence of BS-constraints leads to some fundamental new properties of minimal queries identified in Section 5; specifically, the possibility of multiple minimal queries with different sizes requires new techniques that exploit properties of minimal queries to efficiently enumerate all minimal queries (Section 6.5).

6.1 The Chase Procedure

Given a TPQ Q, the chase query of Q, denoted by Q_C, is computed using a two-step procedure:

S1. Initialize Q_C to be Q. For every query node u in Q_C, attach the reachable subgraph $G^L_{\tau(u)}$ to u, where $L = \{\tau(v) \mid$ v is a pc-child of u in $Q\}$.

S2. Augment Q_C with additional ad-edges.

Step S1 initializes Q_C to be Q, and then enhances each node u in Q with additional nodes from its reachable subgraph based on $\tau(u)$ and the types of u's child nodes in Q. The nodes and edges in Q_C can be classified into two types: the nodes (edges) that are originally in Q are called *original nodes (edges)* and the attached nodes (edges) added to Q_C are called *chase nodes (edges)*.

Step S2, which is referred to as the *augment chase step*, then inserts into Q_C a set (possibly empty) of additional ad-edges referred to as *augmented edges*. This important augmentation step is necessary due to the presence of backward constraints. The need for the augment chase step will be illustrated in Example 6.1; however, we will defer a detailed discussion of this step to Section 6.3 after we have explained the identification of redundant nodes in Q_C using FBsimulation in Section 6.2.

Graphically, we distinguish between original and chase nodes in Q_C by showing the former as boxed nodes and the latter as unboxed nodes. For ease of identification of nodes of the same type in Q_C, we also add subscripts to the node labels when convenient. For consistency of edge notations, the non-arrowed original edges in Q are represented using arrowed edges in Q_C as follows. Let $e = (x, y)$ be an original edge in Q with parent node x and child node y. Then e is represented in Q_C as $x \leftrightarrow y$ if e is a pc-edge; otherwise, e is an ad-edge and it is represented as x$\twoheadleftarrow\!\!\twoheadrightarrow$y. Finally, to distinguish normal ad-edges from augmented ad-edges (introduced by step S2), the latter are shown as bold edges.

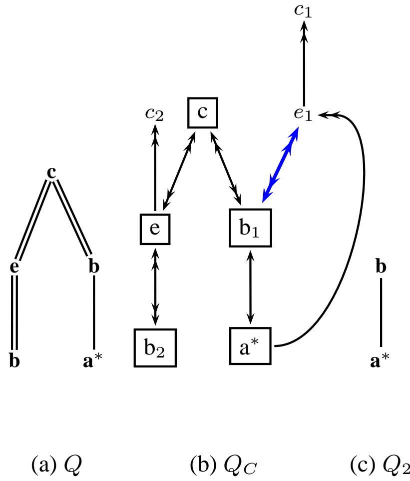
Note that the conventions and definitions introduced in Section 4 for G_C also apply to Q_C.

The following example illustrates step S1 of the chase computation and motivates the need for the augment chase step S2.

Example 6.1 Consider Q in Figure 5 with constraints $C = \{c \twoheadleftarrow e, e \twoheadleftarrow a\}$. We explain how step S1 of the chase is performed on Q to derive Q_C (shown in Figure 5(b)). The reachable subgraph $G^{\emptyset}_a$ is attached to nodes a^*, i.e., two chase nodes e_1 and c_1. The

reachable subgraph $G_e^{\emptyset}$ is attached to nodes e, i.e., chase node c_2. Specifically, no chase nodes are attached to the remaining nodes of Q since their reachable graphs contain only the node itself.

Note that there is an augmented edge between e_1 and b_1 in Q_C; this is added by the augment chase step S2. To appreciate why this addition is necessary, assume for the moment that the augmented edge is not present in Q_C. Observe that the output node a has two ancestor paths of nodes: one leads to c while the other leads to c_1. The purpose of adding the augmented edge (e_1, b_1) is to explicitly connect these two related paths to enable a correct detection of redundant nodes using FBsimulation (to be discussed in the next section). Specifically, the original nodes c, e, and b_2 are all actually redundant and need to be removed to generate a correct minimal query Q_2. However, without the augmented edge (e_1, b_1), the FB-simulation technique would wrongly identify these three nodes as non-redundant. □



(a) Q (b) Q_C (c) Q_2

Figure 5: Example of the Chase Procedure

Complexity of the chase. For a node u in Q, the reachable subgraph $G^L_{\tau(u)}$ is added to u in step S1. Since $G^L_{\tau(u)}$ can be as large as G_C, it consists of $O(|\Sigma|)$ nodes and $O(|\Sigma|^2)$ edges; thus the resultant graph of step S1 comprises of $O(n|\Sigma|)$ nodes and $O(n|\Sigma|^2)$ edges. The ad-edge augmentation in step S2 adds in $O(n^2|\Sigma|)$ edges. Thus the chase of Q, Q_C, comprises of $O(n|\Sigma|)$ nodes and $O(n^2|\Sigma| + n|\Sigma|^2)$ edges.

For each node u, $G^L_{\tau(u)}$ can be computed by at most one traversal on G_C, which takes $O(|\Sigma|^2)$ time. Hence step S1 can be done in $O(n|\Sigma|^2)$ time. Step S2 takes $O(n^3|\Sigma|)$ time. The overall time complexity of the chase procedure is $O(n^3|\Sigma| + n|\Sigma|^2)$. □

6.2 Forward & Backward Simulation

Once the chase query Q_C has been computed, any redundant nodes in Q_C can be detected and eliminated based on the concept of forward-backward simulation (FBsimulation) [4].

The FBsimulation on the nodes of Q_C is the largest binary relation $\preceq_{FB}$ on the nodes of Q_C such that $u \preceq_{FB} v$ iff all the following conditions hold:

(1) Preserve node types: $\tau(u) \leq \tau(v)$; moreover, if $u = op(Q)$, then $v = op(Q)$;

(2) Preserve parent-relationships: if u has a p-parent u', then v has a p-parent or c-parent or s-parent v' s.t. $u' \preceq_{FB} v'$;

(3) Preserve child-relationships: if u has a c-child u', then v has a p-child or c-child or s-child v' s.t. $u' \preceq_{FB} v'$;

(4) Preserve ancestor-relationships: if u has an a-parent u', then v has an ancestor v' s.t. $u' \preceq_{FB} v'$;

(5) Preserve descendant-relationships: if u has a d-child u', then v has a descendant v' s.t. $u' \preceq_{FB} v'$.

The forward simulation $\preceq_F$ on the nodes of Q_C is computed using only conditions (1), (3), and (5); while the backward simulation $\preceq_B$ on the nodes of Q_C is computed using only conditions (1), (2), and (4).

If $u \preceq_F v$ ($u \preceq_B v$, $u \preceq_{FB} v$ resp.), we say that v is a forward (backward, forward-backward resp.) simulator of u. Given a node u in Q_C, we use *Fsim*(u), *Bsim*(u), and *FBsim*(u) to refer to the set of forward, backward, and forward-backward simulators of u, respectively. Note that $FBsim(u)$ is always a subset of $Fsim(u)$ and $Bsim(u)$.

Algorithms for computing forward simulation Fsim(u) in TPQs were presented in [14]. These algorithms could be extended to compute $FBsim$ relation on Q_C for our context. Such modified algorithms take $O(n^3|\Sigma| + n^2|\Sigma|^2)$ time to compute Fsim(u) and FBsim(u) for every original node u in Q_C.

Example 6.2 (continued from Example 4.1) For Q_C in Figure 3, we have $Fsim(a) = FBsim(a) = \{a\}$, $Fsim(b) = FBsim(b) = \{b,\ b_1\}$, $Fsim(c) = FBsim(c) = \{c,\ c_1,\ c_2\}$, and $Fsim(d) = FBsim(d) = \{d,\ d_1\}$. □

We can now identify redundant nodes in Q_C based on the following result.

LEMMA 6.1. *Let $u \in Q_C$ be a non-redundant original node.*

(1) An original p-parent p of u is redundant iff u has another p-parent $p' \in FBsim(p)$;

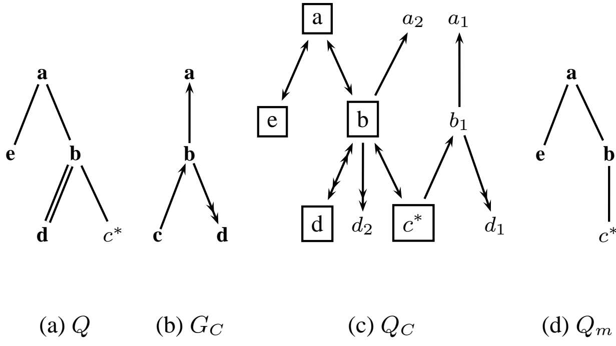
(2) An original a-parent p of u is redundant iff u has another ancestor $p' \in FBsim(p)$;

(3) An original c-child c of u is redundant iff u has another c-child or s-child $c' \in Fsim(c)$;

(4) An original d-child c of u is redundant iff u has another descendant $c' \in Fsim(c)$.

Lemma 6.1 is extension of a result in [14], where the focus there was identifying redundant c-child and d-child nodes using only forward simulation; i.e. conditions (3) and (4). The new two conditions (1) and (2) for identifying redundant p-parent and a-parent nodes, however, require a stronger test based on FBsimulation. We illustrate the necessity of FBsimulation using the following example.

Example 6.3 Consider Q in Figure 6(a) with $C = \{a \leftarrow b, b \twoheadrightarrow d, b \leftarrow c\}$. G_C and Q_C are shown in Figures 6(b)&(c). Note that the original node b is not redundant. In Q_C, we have $b_1 \notin FBsim(b)$, which is consistent with the fact that b is not redundant. Thus, node b remains in Q's minimal query Q_m shown in Figure 6(d). Observe that node c has another p-parent b_1 in Q_C such that $b_1 \in Fsim(b)$ and $b_1 \in Bsim(b)$. However, it is incorrect to conclude from $b_1 \in Fsim(b)$ and $b_1 \in Bsim(b)$ that b is redundant. □



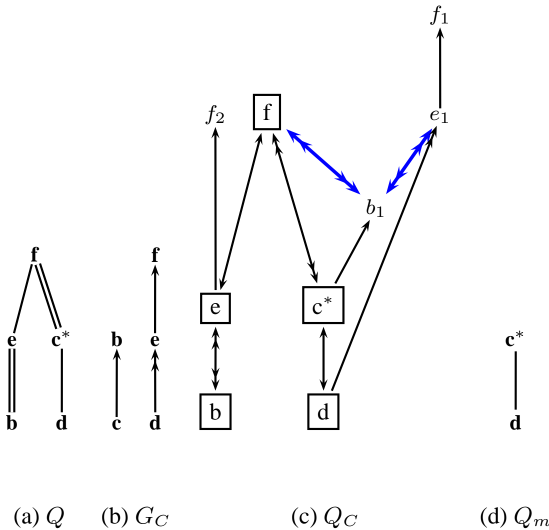
(a) Q (b) G_C (c) Q_C (d) Q_m

Figure 6: FBsimulation in Checking Redundant Parent

6.3 Augmented Chase

We are now ready to explain the details of the important augment chase step (step S2 in Section 6.1). Due to the presence of backward constraints, a node in Q_C could have multiple path of ancestor nodes which are necessarily related since each node in a query tree cannot have multiple unrelated paths of ancestor nodes. Thus, the augment chase step S2 is necessary to check for possible implied relationships among multiple ancestor paths. Without this important step, redundant nodes might not be correctly detected leading to incorrect minimal queries.

Recall that Example 6.1 illustrated the need to perform ad-edge augmentation between an original node and a chase node in Q_C. In general, ad-edge augmentation is necessary for any pair of related ancestor nodes. In the next example, we illustrate another scenario where ad-edge augmentation is required between a pair of related chase nodes.



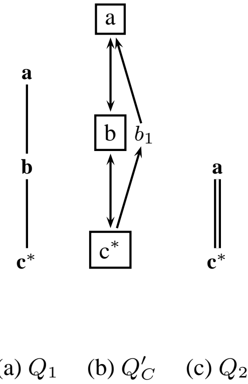
(a) Q (b) G_C (c) Q_C (d) Q_m

Figure 7: Example of ad-edge Augmentation

Example 6.4 Consider Q in Figure 7(a) with $C = \{b \leftarrow c, e \twoheadleftarrow d, f \leftarrow e\}$. G_C and Q_C are shown in Figures 7(b) and (c), respectively. Note that Q_C has an augmented ad-edge between the chase nodes b_1 and e_1 due to the tree structure property (b_1 is a p-parent of c, c is a p-parent of d, and e_1 is an a-parent of d). Similarly there is an augmented ad-edge between b_1 and f. From Q_C, we can identify that the original nodes f, e, and b are redundant (due to their forward-backward simulators f_1, e_1, and b_1, respectively), leading to the minimal query Q_m in Figure 7(d). Without the augmented ad-edges, Q_m cannot be derived based on FBsimulation. □

An important point to emphasize is that it is only necessary to perform edge augmentation for ad-edges: adding pc-edges between related ancestor nodes is not sound. We illustrate the intuition behind this with the following example.

Example 6.5 Consider Q_1 in Figure 8 with $C = \{b \leftarrow c\}$. If pc-edge augmentation is done, then the resultant chase query Q'_C is shown in Figure 8 with an augmented pc-edge between a and b_1. We then have $b_1 \in FBsim(b)$, leading to a wrong conclusion that b is redundant and an incorrect minimization of Q_1 to Q_2 that results from the removal of b from Q'. To see that Q_1 and Q_2 are not equivalent, consider data *<a><d><b><c/><b/><d/><a/>*. Q_2 matches the data, while Q_1 does not. □



(a) Q_1 (b) Q'_C (c) Q_2

Figure 8: Example of ad-edge Augmentation

6.3.1 Augmentation Algorithm

This section presents the algorithm to perform the augment chase step S2. The key challenge of this step is: given a node u in Q_C which has two ancestor paths of nodes, how to characterize the pair(s) of related nodes along these paths that need to be connected with augmented edges. We shall formalize this using the concepts of ancestor path, p-path and compatible nodes.

A path of nodes $(w_1, \cdots, w_n)$ in Q_C, $n \geq 1$, is said to be an *ancestor path of* w_1 if (1) w_i is a p/a-parent of w_{i-1} for each $i \in (1, n]$; and (2) w_n has no p/a-parent node in Q_C. A path of nodes $(v_1, \cdots, v_m)$ in Q_C, $m \geq 1$, is said to be a *p-path of* v_1 if (1) v_i is a p-parent of v_{i-1} for each $i \in (1, m]$; and (2) v_m has no p-parent node in Q_C.

A node v_i is said to be compatible with another node w_i iff $\tau(v_i) \leq \tau(w_i)$ or $\tau(w_i) \leq \tau(v_i)$.

Consider a node u in Q_C that has both a p-path $p_v = (u,\ v_1,\ \cdots,\ v_j,\ \cdots,\ v_k,\ \cdots,\ v_m)$ as well as another ancestor path $p_w = (u, w_1, \cdots, w_j, \cdots, w_n)$, where $1 \leq j \leq k \leq m$ and $j \leq n$. Figure 9 illustrates the situations where an augmented edge is required to connect a pair of related nodes in p_v and p_w, which are characterized as follows. An ad-edge needs to be augmented for the following two cases.

Case 1. An ad-edge is added between w_{k+1} (the ancestor) and v_k (the descendant), if the following two conditions hold:

1. v_i is compatible with w_i for each $i \in [1, k]$; and
2. w_{k+1} is an a-parent of w_k.

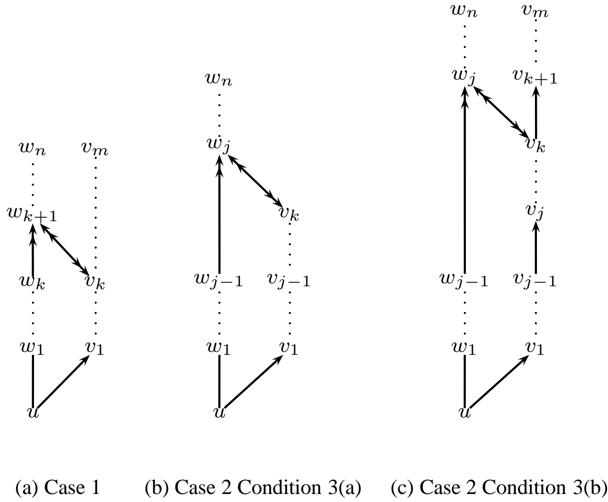
Case 2. An ad-edge is added between w_j (the ancestor) and v_k (the descendant), if all the three conditions hold:

1. v_i is compatible with w_i for each $i \in [1, j-1]$; and
2. w_j is an a-parent of w_{j-1}; and
3. (a) v_k is the last node in p_v (i.e., $k = m$), and v_s is not compatible with w_j for each $s \in [j, k]$; or

   (b) w_j is not compatible with v_t for each $t \in [j, k]$, and v_{k+1} is compatible with w_j where $k \in [j, m-1]$.

Figure 9(a) shows Case 1, where p_v and p_w have pairwise compatible nodes w_i and v_i for $i \in [1, k]$, and w_{k+1} is an a-parent of w_k. The tree-structure property requires w_{k+1} to be an ancestor of v_k. Figure 9(b) shows Case 2 where condition 3(a) holds. Here, p_v and p_w have pairwise compatible nodes w_i and v_i for $i \in [1, j)$. Node w_{j-1} has an a-parent w_j, and v_{s-1} has a p-parent v_s which is not compatible with w_j for $s \in [j, k]$. Clearly, node v_k must be a descendant of w_j based on the tree structure property. Note that if w_j and v_k were compatible, then adding the ad-edge (w_j, v_k) would be incorrect as w_j and v_k might be the same node instead of being an ancestor-descendant pair of nodes. The reasoning for Case 2 where condition 3(b) holds (depicted in Figure 9(c)) is similar.

Note that for all such augmented ad-edges, the implied descendant always belongs to the p-path. The nodes in Figure 9 can be chase or original nodes. The edge (w_1, u) can be either a pc-edge or an ad-edge.



(a) Case 1 (b) Case 2 Condition 3(a) (c) Case 2 Condition 3(b)

Figure 9: Illustration of Augmentation Algrithm

We propose an algorithm, `ad-augmentation` (Algorithm 2), to augment all the necessary ad-edges between each p-path and ancestor path of an original node u, which essentially follows the above fundamental principle.

Let Q_C^u be the subgraph of Q_C consisting of u, and the ancestor nodes of u that are also chase nodes of u. Let Q_C^{u+} be the subgraph of Q_C consisting of u and all ancestor nodes of u, which can be original or chase nodes. Thus, Q_C^u is a subgraph of Q_C^{u+}. Each original query node has exactly one p-path l_c in Q_C^u, and one or more p-paths l_i in Q_C^u. We have noted that all the ancestor paths of u are contained in Q_C^{u+}, and that every p-path l_i of u ($l_i \neq l_c$) is necessarily a p-path of u's original p-parent. Instead of directly applying the above principle to every combination of a p-path and

Algorithm 2 ***ad-augmentation*** (Q_C)

1: **for** each original node u in bottom-up order **do**
2: $\quad l_c$:= the p-path in Q_C^u
3: $\quad$ *ad-PathToGraph*(l_c, Q_C^{u+})
4: $\quad$ **for** each p-path $l_i \neq l_c$ in Q_C^{u+} **do**
5: $\quad\quad$ *ad-PathToGraph*(l_i, Q_C^u)

an ancestor path of u, we apply the principle to (1) l_c with Q_C^{u+} (line 3 of Algorithm 2), and (2) l_i with Q_C^u for each $l_i \neq l_c$ (lines 4-5 of Algorithm 2).

The function *ad-PathToGraph* used in Algorithm 2 augments the necessary ad-edges from the nodes in a p-path of u, l_p, to the nodes in the subgraph Q_s (either Q_C^u or Q_C^{u+}) rooted at u, which closely follows the above fundamental principle. Function *ad-PathToGraph* takes $O(n|V_s|)$ time and adds at most $O(|V_s|)$ ad-edges, where V_s is the set of nodes in Q_s.

Example 6.6 Consider Q in Figure 3 with the same constraints as in Example 4.1. Q_C is shown in Figure 3(c). The chase nodes in Q_C is added during step S1 of the chase by attaching the reachable subgraphs. Note that the chase edge between a and b_1 is added during step S2 of the chase. □

Complexity of ad-augmentation Q_C^{u+} (may be as big as Q_C) has $O(n|\Sigma|)$ nodes, and Q_C^u (may be as big as G_C) has $O(|\Sigma|)$ nodes. The number of p-paths of u in Q_C is bounded by $O(n)$. Algorithm `ad-augmentation` takes = $O(n^3|\Sigma|)$ time, and adds $O(n^2|\Sigma|)$ ad-edges. □

6.4 Generating a Minimal Query

In this section, we present the algorithm `ChaseMinimizeFBST` (shown in Algorithm 3) to compute a single minimal query of an input query Q w.r.t. a set of FBST-constraints.

The input to `ChaseMinimizeFBST` is a non-redundant original node u in Q_C, and all redundant original nodes in Q_C are detected (by applying Lemma 6.1) and removed via recursive calls to `ChaseMinimizeFBST`. Each query Q has at least one non-redundant node given by its output node $op(Q)$, $u = op(Q)$ in the first call to `ChaseMinimizeFBST`. Steps 2 to 6 check whether the p-parent and a-parent v of u in Q_C is redundant. If v is redundant, v as well as its chase nodes (if any) are removed. If v is not redundant, a recursive call to `ChaseMinimizeFBST` is made with v as the input parameter. Similarly, steps 7 to 13 check and remove any redundant c-child and d-child nodes of u. Nodes that have been checked are marked to avoid repeated redundant checkings.

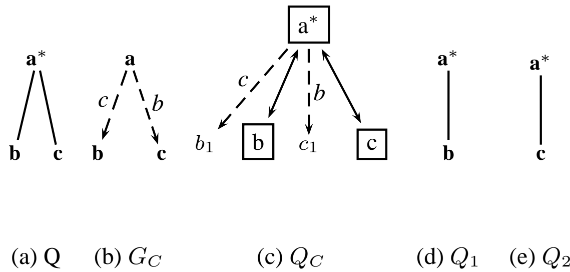
It is important to point out that due to possible sibling constraints in C, the removal of redundant c-child nodes in Q_C entails some additional checking (in steps 10 and 11) for correctness. Specifically, if an original c-child v of u is detected to be redundant, it is necessary to also check whether u has another c-child of type $\tau(v)$. If there is no such c-child, then s-child nodes of u with edge label $\tau(v)$ must also be deleted. This is due to the fact that the condition (u has a c-child of type $\tau(v)$) no longer exists. We illustrate this subtle point using the following example.

Example 6.7 Consider Q in Figure 10(a) with G_C shown in Figure 10(b). Q_C is shown in Figure 10(c). `ChaseMinimizeFBST` identifies the original c-child b of node a to be redundant (because a has another c-child $b_1 \in FBsim(b)$) and b is deleted. In addition, due to steps 10 and 11, `ChaseMinimizeFBST` also finds that a has no other c-child of type b. Consequently, the c-child c_1 with edge label b is deleted together with b, leading to a mini-

Algorithm 3 ChaseMinimizeFBST (u)

1: mark u as visited
2: **while** (u has an original parent v that is unvisited) **do**
3: **if** (v is a p-parent of u and u has another p-parent $q \in FBsim(p)$) or (v is an a-parent of u and u has another ancestor $q \in FBsim(p)$) **then**
4: delete v and its chase nodes (ad-edges are added between v's original parent and children if any)
5: **else**
6: ChaseMinimizeFBST(v)
7: **while** (u has an original child v that is unvisited) **do**
8: **if** (v is a c-child of u and u has another c-child or s-child $w \in Fsim(v)$) or (v is a d-child of u and u has another descendant $w \in Fsim(v)$) **then**
9: delete v and its chase nodes (ad-edges are added between u and v's original children if any)
10: **if** u has no more c-child of $\tau(v)$ **then**
11: delete u's s-edge (u, w) with edge label $\tau(v)$
12: **else**
13: ChaseMinimizeFBST(v)

mal query Q_2 (Figure 10(e)). Note that had the chase node c_1 not been deleted together with b, a's original c-child c would have been wrongly detected to be redundant (due to the fact that a has another c-child $c_1 \in FBsim(c)$), leading to an incorrect result. □



(a) Q (b) G_C (c) Q_C (d) Q_1 (e) Q_2

Figure 10: Lines 10 and 11 of `ChaseMinimizeFBST`

Example 6.8 (continued from example 6.2) For Q_C in Figure 3(c), starting from output node a, `SingleMinimizeFBST` finds the d-child b of a is redundant as a^* has another descendant node $b_1 \in FBsim(b)$. Then b and b's chase nodes (c_2, d_1 and e_1) are deleted. Child nodes c and d of b are connected to a using ad-edges. Now c is a d-child of a, and Lemma 6.1 finds it redundant as a has another descendant $b_1 \in FBsim(b)$. Node c is then deleted. `SingleMinimizeFBST` outputs a minimal query Q_1 (Figure 3(d)) after deleting the remaining chase nodes. □

Complexity of ChaseMinimizeFBST The worst case of algorithm `ChaseMinimizeFBST` occurs when each redundancy check of Lemma 6.1 enables only one node removal (i.e., one node is deleted at a time). For a non-redundant node p, for each child (parent resp.) u of p, it takes time proportional to the number of descendant (ancestor resp.) nodes of p to check whether u is redundant. Hence the time of checking whether u is redundant using Lemma 6.1 is $O(|V_C|) = O(n|\Sigma|)$ where V_C is the set of nodes in Q_C. Deleting u and its chase nodes takes $O(n|\Sigma|)$ time. Thus the worst case of algorithm `ChaseMinimizeFBST` takes $O(n*(n|\Sigma|)) = O(n^2|\Sigma|)$ time. □

For algorithm `SingleMinimizeFBST`, we have Lemma 6.2 .

LEMMA 6.2. *Algorithm SingleMinimizeFBST correctly computes a minimal query of a given TPQ Q in the presence of a set of FBST-constraints. The time-complexity of SingleMinimizeFBST is* $O(n^3|\Sigma| + n^2|\Sigma|^2)$.

6.5 Enumerating All Minimal Queries

In this section, we present an algorithm (referred to as `MultipleMinimizeFBST`) to enumerate all the minimal queries of an input query Q w.r.t. a set of FBST-constraints. The algorithm consists of two key steps. First, `SingleMinimizeFBST` is used to generate a single minimal query Q_m of Q. Next, by applying Proposition 5.4, the remaining minimal queries of Q (if any) can be generated by modifying Q_m. Specifically, additional minimal queries of Q are enumerated from Q_m by replacing each X, where X is an ad-leaf node or a parent-child edge in Q_m that satisfies the conditions of Proposition 5.4, with the minimal members in the equivalence class $[X]$.

The time complexity of `MultipleMinimizeFBST` is $O(max\{n^3|\Sigma| + n^2|\Sigma|^2, |Q_M|\})$, where $O(n^3|\Sigma| + n^2|\Sigma|^2)$ is the time complexity of algorithm `SingleMinimizeFBST`; and Q_M is the set of distinct minimal queries of Q, which could be exponential in the input query size.

Example 6.9 (continued from Example 6.8) From Example 6.8, we know that Q_1 (Figure 3(d)) is a minimal query of Q (Figure 3(b)). Q_1 has one ad-leaf node, d and no parent-child edges (u, v) where v is a leaf. Following Proposition 5.4, Q_1 has two other minimal queries, as shown in Figures 3(e) and (f). □

7. MINIMIZE WITH FST-CONSTRAINTS

In this section, we consider the TPQ minimization problem under FST-constraints. Our key result here is that in the absence of backward constraints, a query can be minimized without explicitly computing the chase of the query, resulting in a more efficient approach. Section 7.1 first presents a new algorithm for FT-constraints. Our algorithm has a time complexity of $O(n|\Sigma|+n^2)$, which improves over the $O(n^4)$ time-complexity of the state-of-the-art algorithm [14]. In Section 7.2, we extend our approach to handle FST-constraints.

For convenience, we use *simulate* to mean forward simulate and $sim(u)$ to mean $Fsim(u)$ in this section.

Let V denote the set of nodes in a query Q, and let $S \subseteq V$ be a subset of query nodes, and let $T \subseteq \Sigma$ be a subset of element types. We define $type(S) = \{\tau(v) \mid v \in S\}$, $node(T) = \{v \in V \mid \tau(v) \in T\}$, $par(S) = \{v \in V \mid v$ has a pc-child in $S\}$, and $anc(S) = \{v \in V \mid v$ has a descendant in $S\}$.

7.1 Minimize with FT-constraints

Our new improved algorithm is based on the idea of *type simulators*. The complexity reduction of our algorithm is essentially achieved by avoiding the construction of the chase query Q_C and the subsequent computation of simulation relation on Q_C, both of which are costly.

Type simulators. Consider a query node u and a type $t \in \Sigma$. Formally, we say that t *simulates* u (denoted by $u \preceq t$) if all the following conditions hold:

1. u is neither the query output node nor its ancestor;
2. $t \leq \tau(u) \in$ closure(C);
3. for each pc-child v of u, there exists a type $t' \in \Sigma$ such that $t \rightarrow t' \in$ closure(C) and $v \preceq t'$;

4. for each ad-child v of u, there exists a type $t' \in \Sigma$ such that $t \twoheadrightarrow t' \in$ closure(C) and $v \preceq t'$.

The above conditions (2) to (4) are straightforward as they follow the definition of simulation. Condition (1) follows from a property of FT-constraints: if a query node v is redundant, then any descendant query node of v is necessarily also redundant [14]. Since the query output node is non-redundant, all its ancestor nodes must also be non-redundant.

We use $simtype(u)$ to denote the set of types in Σ that simulate u; i.e, $simtype(u) = \{t \in \Sigma \mid u \preceq t\}$. The function $simtype()$ can be computed efficiently using two auxiliary functions $partype()$ and $anctype()$ which are defined as follows.

Given a set $T \subseteq \Sigma$, $partype(T)$ is defined to be the set of types in Σ having a required-child type in T; and $anctype(T)$ is defined to be the set of types in Σ having a required-descendant type in T as follows:

$$partype(T) = \{t \in \Sigma \mid t \rightarrow t' \in closure(C), t' \in T\};$$

$$anctype(T) = \{t \in \Sigma \mid t \twoheadrightarrow t' \in closure(C), t' \in T\}.$$

Based on $partype()$ and $anctype()$, $simtype(u)$ can be defined as follows: if u is the output query node or its ancestor, then $simtype(u) = \emptyset$; otherwise,

$$simtype(u) = \{t \in \Sigma \mid t \leq \tau(u) \in closure(C), \\ t \in partype(simtype(v)) \text{ for each } pc\text{-child } v \text{ of } u, \\ t \in anctype(simtype(v)) \text{ for each } ad\text{-child } v \text{ of } u\}.$$

Node simulators. Recall that $sim(u)$ denotes the set of forward simulators of u. The function $sim()$ can be defined in terms of two additional auxiliary functions, namely, $augpar()$, and $auganc()$, which are defined as follows:

- $augpar(sim(u)) = par(sim(u)) \cup \{v \in V \mid \tau(v) \in partype(simtype(u))\}$; and
- $auganc(sim(u)) = anc(sim(u)) \cup S_{RA} \cup anc(S_{RA})$, where $S_{RA} = \{v \in V \mid \tau(v) \in anctype(simtype(u))\}$.

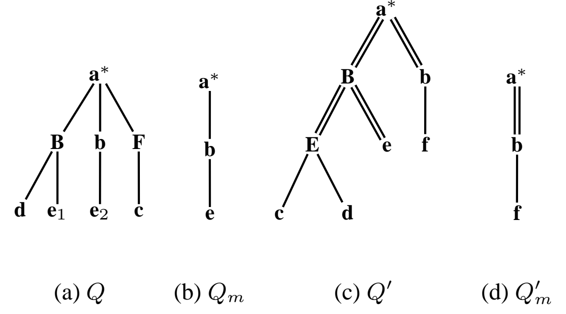
Based on $augpar()$ and $auganc()$, $sim(u)$ can now be defined in terms of three cases as follows:

- if u is the output node, then $sim(u) = \{u\}$;
- if u is a non-output leaf node, then $sim(u) = \{v \in V \mid \tau(v) \leq \tau(u) \in closure(C)\}$;
- if u is neither the output node nor a leaf node, then

  $sim(u) = \{v \in V \mid \tau(v) \leq \tau(u) \in closure(C)$, $v \in augpar(sim(u'))$ for each pc-child u' of u, and $v \in auganc(sim(u''))$ for each ad-child u'' of $u\}$.

Example 7.1 Consider query Q shown in Figure 11(a) with constraints $C = \{b \leq B,\ b \rightarrow d,\ f \leq F,\ a \rightarrow f,\ f \rightarrow c\}$. We have $sim(d) = \{d\}$, $sim(e_1) = \{e_1, e_2\}$, $augpar(sim(d)) = \{B, b\}$, $par(sim(d)) = \{B\}$, and $augpar(sim(e_1)) = \{B, b\}$. Since $b \in augpar(sim(d))$ and $b \in augpar(sim(e_1))$, and $b \leq B$, we have $b \in sim(B)$. We also have $sim(c) = \{c\}$, $augpar(sim(c))$ = $\{F\}$, $simtype(F) = \{f\}$ and $partype(simtype(F)) = \{a\}$. □

Algorithm `MinimizeQuery-FT` minimizes a given TPQ Q in the presence of a set C of FT-constraints. It first computes a reduced query $R(Q)$, which is the resultant query of Q after repeatedly removing a leaf node that is redundant due to $closure(C)$ until no leaf is redundant [14]. As defined in [14], a pc-leaf node v (with parent u) of Q is redundant if $\tau(u) \rightarrow \tau(v) \in closure(C)$; an ad-leaf node v (with parent u) of Q is redundant if $\tau(u) \twoheadrightarrow \tau(v) \in closure(C)$. $R(Q)$ can be computed in $O(n^2)$ time and has at most n nodes.



(a) Q (b) Q_m (c) Q' (d) Q'_m

Figure 11: Minimization with FST-constraints

Algorithm 4 MinimizeQuery-FT(TPQ Q)
1: Compute the reduced query of Q, $R(Q)$
2: Compute the simulation on $R(Q)$ as follows: for each node u in $R(Q)$ in bottom-up order do compute $simtype(u)$, $partype(simtype(u))$, $anctype(simtype(u))$, $sim(u)$, $augpar(sim(u))$, and $auganc(sim(u))$.
3: Minimize-FT(v_{root}), where v_{root} is the root node of $R(Q)$

Algorithm `MinimizeQuery-FT` then computes the type simulators and node simulators. Then it calls the recursive algorithm `Minimize-FT` to remove the redundant nodes as follows.

For a non-redundant node u starting from the root of $R(Q)$,

- a pc-child v of u is redundant if u has another pc-child $w \in sim(v)$ or $\tau(u) \in partype(simtype(v))$;
- an ad-child v of u is redundant if u has another child $w \in sim(v) \cup auganc(sim(v))$ or $\tau(u) \in anctype(simtype(v))$.

If v is redundant, the subtree rooted at v is deleted. Otherwise, `Minimize-FT` continues minimization on v.

Example 7.2 (continued from Example 7.1) The pc-child of a, B, is redundant as a has another pc-child $b \in sim(B)$. Thus the subtree rooted at B is deleted. At the same time, we have F is redundant as $a \in partype(simtype(F))$. Thus the subtree rooted at F is also deleted, leading to the minimal query Q_m as shown in Figure 11(b). □

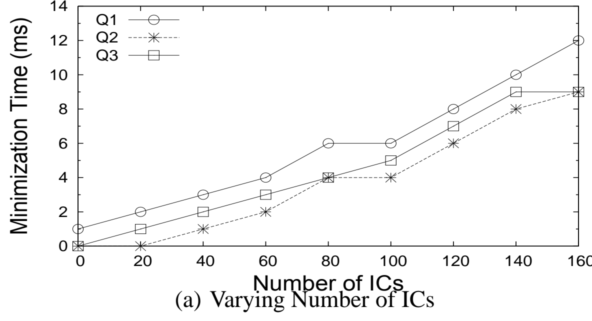
Complexity of MinimizeQuery-FT Computing the reduced query $R(Q)$ takes $O(n^2)$ time. The simulation relations are computed in $O(n|\Sigma|)$ time. Minimize-FT(v_{root}) takes $O(n^2)$ time. Hence the time-complexity of MinimizeQuery-FT is $O(n|\Sigma|+n^2)$. □

For `MinimizeQuery-FT`, we have the following result.

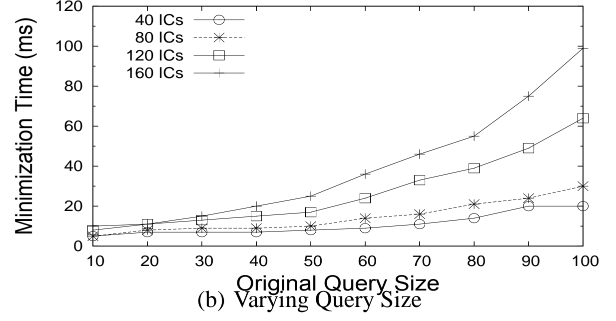
LEMMA 7.1. *Algorithm MinimizeQuery-FT correctly generates the minimal query for a given TPQ Q in the presence of FT-constraints in $O(n|\Sigma|+n^2)$ time.*

7.2 Extensions for FST-constraints

The minimization algorithms for FST-constraints are equivalent to those for FT-constraints except that we need to use extended definitions of both $augpar()$ and $auganc()$ to take into account of sibling constraints.



(a) Varying Number of ICs



(b) Varying Query Size

Figure 12: Experimental Results: Minimization Time

Algorithm 5 Minimize-FT(node u)

1: **for** each pc-child v of u **do**
2: **if** u has another pc-child $w \in sim(v)$ that has not been deleted **then**
3: delete the subtree rooted at v
4: **else if** $\tau(u) \in partype(simtype(v))$ **then**
5: delete the subtree rooted at v
6: **else**
7: Minimize-FT(v)
8: **for** each ad-child v of u **do**
9: **if** u has another child $w \in sim(v) \cup auganc(sim(v))$ that has not been deleted **then**
10: delete the subtree rooted at v
11: **else if** $\tau(u) \in anctype(simtype(v))$ **then**
12: delete the subtree rooted at v
13: **else**
14: Minimize-FT(v)

If v has a pc-child w and $\tau(v) \overset{\tau(w)}{\dashrightarrow} t \in closure(C)$, then v must necessarily have another pc-child of type t. When sibling constraints are included in C, the following extensions are needed:

- $augpar(sim(u)) = par(sim(u)) \cup \{v \in V \mid \tau(v) \in partype(simtype(u)) \cup P\}$;
- $auganc(sim(u)) = anc(sim(u)) \cup S_{RA} \cup anc(S_{RA}) \cup P \cup anc(P)\}$

where $S_{RA} = \{v \in V \mid \tau(v) \in anctype(simtype(u))\}$; and $P = \{v \in V \mid v$ has a pc-child w, $\tau(v) \overset{\tau(w)}{\dashrightarrow} t \in closure(C)$, $t \in simtype(u)\}$.

Similarly, we have the following result.

LEMMA 7.2. *In the presence of FST-constraints, a TPQ Q can be minimized in $O(n|\Sigma|+n^2)$ time.*

Example 7.3 Consider minimization of Q' in Figure 11(c) with constraints $C = \{b \leq B,\ e \leq E,\ e \rightarrow c,\ e \rightarrow d,\ b \overset{f}{\dashrightarrow} e\}$. Q itself is a reduced query. We have $sim(c) = \{c\}$, $sim(d) = \{d\}$, and $simtype(c) = \{c\}$, $simtype(d) = \{d\}$; and $augpar(sim(c)) = \{E, e\}$, $augpar(sim(d)) = \{E, e\}$; we have $sim(E) = \{E, e\}$, $sim(e) = \{e\}$, and $simtype(E) = simtype(e) = \{e\}$. We have $auganc(sim(E)) = \{B, a, b\}$ and $auganc(sim(e)) = \{B, a, b\}$ due to the extensions of definitions of $augpar()$ and $auganc()$: b has a pc-child f such that $b \overset{f}{\dashrightarrow} e$ holds where $e \in simtype(E)$ ($e \in simtype(e)$ too). Then we have $b \in sim(B)$. The subtree rooted at B is then deleted, which leads to minimal query Q'_m as shown in Figure 11(d). □

8. OTHER MINIMIZATION ALGORITHMS

In this section, we summarize the time complexity results for computing a single minimal query for the remaining classes of constraints shown in Figure 2(b). For FBT-constraints, the time complexity is the same as that for FBST-constraints.

However, for FB/FBS-constraints, the time computing reduces to $O(n^4)$. The improvement in complexity is due to an optimized chase procedure: during step S1 of the chase procedure, only the types that exist in the query prior to chase are attached. This leads to a smaller chase query Q_C with $O(n^2)$ nodes and $O(n^3)$ edges. Consequently, the simulation relations can be computed in $O(n^4)$ time.

9. EXPERIMENTAL RESULTS

In this section, we present an experimental study on the efficiency and effectiveness of our minimization algorithms. The algorithms being compared include both of our proposed algorithms `SingleMinimizeFBST` (`FBST` for short) and `MultipleMinimizeFBST` (`MFBST` for short), as well as `CTPQMinimize` (`CTPQ` for short) proposed in [14] for forward constraints. We used both real (DBLP records [11]) as well as synthetic (XMark [15]) datasets in our experiments. Our experimental results on both the XMark and DBLP datasets show the efficiency and scalability of `FBST`. All the algorithms were implemented in Java, and the experiments were run on a Pentium 4 PC with a 3.0Ghz processor, 1 GB of main memory, and a 30 GB hard disk.

9.1 Efficiency and Scalability of Minimization

In our first set of experiments, we study the effect of the number of constraints and the effect of query size on the *query minimization time* of `FBST`. This metric measures the time to generate a minimal query for an input query. Our experimental results on both the XMark and DBLP datasets show the efficiency and scalability of `FBST`; however, due to space limitation, we only present the experimental results on the DBLP dataset.

For the DBLP dataset of 127MB, we extracted 160 non-trivial constraints (29 forward, 10 backward and 121 sibling constraints). The purpose of using such a large number of constraints is to evaluate the scalability of our algorithms; the minimal queries, however, can actually be obtained using a set of no more than 20 constraints.

Input query	Minimal query produced by FBST
X1: //site/people/person[name]//profile//education	//education
X2: //site/open_auctions/open_auction/bidder/increase	//increase
X3: //site/open_auctions/open_auction/reserve	//reserve

Table 2: Queries in comparison with CTPQMinimize

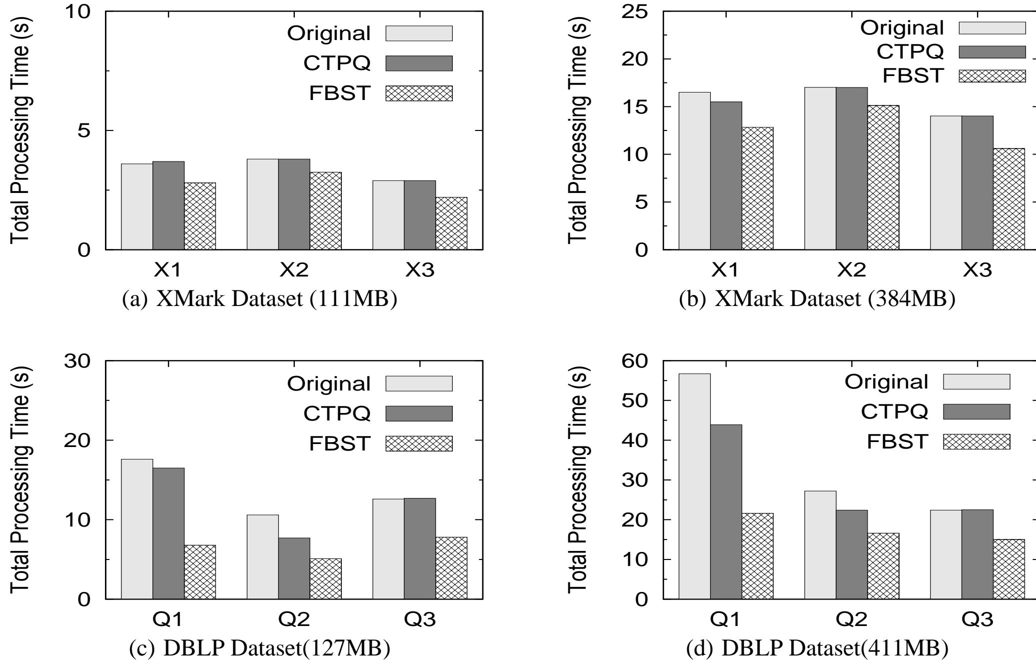


Figure 13: Experimental Results: Total Processing Time (Minimization + Evaluation)

Effect of #ICs on minimization time. This experiment studies the effect of the number of ICs on the minimization time of FBST for the following three queries by varying the input set of non-trivial constraints used:

$Q1$: //dblp/inproceedings[booktitle][year][author][url][crossref][number][pages]/title
$Q2$: //dblp/article[volume][author][url][title][year][journal]/cite
$Q3$: //dblp/book[cite][author][year][isbn][publisher][title]

The sizes of $Q1$, $Q2$, and $Q3$ are 10, 9, and 8 nodes, respectively, and their minimal queries (w.r.t. 160 ICs) have 3, 2, and 2 nodes respectively (Q_{min1} = *//inproceedings//[number]/title*, Q_{min2} = *//article/cite*, Q_{min3} = //book[cite]).

The results shown in Figure 12(a) indicate that the minimization time increases with the number of ICs (as expected). Note that the minimization times reported are rather small: the minimization time for $Q1$, $Q2$, and $Q3$ (w.r.t. 160 ICs) is only 12ms, 9ms, and 9ms respectively; which is a negligible overhead in a typical XPath query evaluation process.

Effect of query size on minimization time. This experiment studies the effect of the query size on the minimization time of FBST.

We generated a set of 10 test queries of increasing sizes from 10 to 100 nodes. The 10 test queries are composed using a collection of five query fragments (Q_a through Q_e) together with a query root fragment Q_o = *//dblp*. Note that Q_a has 9 nodes, and each of Q_b through Q_e has 10 nodes.

Q_a: phdthesis[ee][series][title][year][school][publisher][author][number]
Q_b: proceedings[cite][journal][volume][number][publisher][editor][isbn][series]/title
Q_c: inproceedings[title][booktitle][year][number][cdrom][author][pages][url]/crossref
Q_d: incollection[chapter][isbn][publisher][ee][url][pages][author][cite]/booktitle
Q_e: book[url][isbn][title][month][year][cdrom][cite][publisher]/author

The first test query Q_{10}, which consists of 10 nodes, is composed using Q_o and Q_a; and the remaining 9 test queries are composed by appending query fragments from $\{Q_b, \cdots Q_e\}$ to Q_{10} under Q_o as branches repeatedly. For each test query, we measure its minimization time w.r.t. 4 different collections of ICs consisting of 40, 80, 120, and 160 constraints, respectively.

Figure 12(b) shows the minimization time for the 10 test queries w.r.t. the 4 sets of ICs. As expected, the minimization time (w.r.t. a given set of ICs) increases with a larger query size. The results also demonstrate the scalability of FBST algorithm: the minimization time for the largest test query (with 100 nodes) w.r.t. the largest set of 160 ICs is only 99ms. We note that for each of the test queries, the size of its minimal query (w.r.t. to the complete set of ICs) is no more than 30% of the size of the original query.

We also conducted experiments comparing the performance of FBST and MFBST on both DBLP and XMark datasets. Our results indicate that the minimization time of MFBST (to enumerate all minimal queries) is no more than 10% longer than the minimization time of FBST (to compute a single minimal query).

9.2 Comparison of Total Processing Time

In our second set of experiments, we compare the *total query processing time* of algorithm FBST and CTPQ. This metric measures both the time taken to generate a minimal query for an input query as well as the time taken to evaluate the minimized query against a dataset. We also compare the total processing time of FBST and CTPQ with the evaluation time of the original query without minimization.

We first ran both FBST and CTPQ to minimize the input queries on both XMark and DBLP datasets, and then evaluated the respective minimized queries on the XML datasets using the efficient query evaluation engine, GCX [16].

The test queries on XMark are shown in the first column of Table 2. Queries $X2$ corresponds to XMark's benchmark queries Q2; while $X1$ and $X3$ are slightly modified versions of XMark's benchmark queries $Q1$ and $Q4$. As there are no benchmark queries for DBLP, we used the same synthetic queries $Q1$, $Q2$ and $Q3$ as in Section 9.1.

The constraints used for query minimization are derived from the XMark DTD and DBLP datasets. The minimized queries produced by FBST for $X1$, $X2$ and $X3$ are shown in the second column of Table 2. The minimal queries of $X1$, $X2$ and $X3$ produced by CTPQ are the same as their input queries except for query $X1$, of which the minimized query is //site/people/person//profile//education. Comparing the minimized queries produced by FBST and CTPQ, it is clear that using additional backward and sibling constraints are effective for query minimization.

In terms of query minimization time, the minimization time required by both CTPQ and FBST is less than 10 ms for each of the test queries. Thus, the overhead incurred by FBST using more constraints is negligible.

The minimized queries of $X1$ through $X3$, and $Q1$ through $Q3$ were then evaluated to measure their evaluation times on the XML datasets. We used two datasets of different sizes for both XMark and DBLP. For XMark, datasets of 111MB (the standard XMark dataset [15]) and 384MB (generated using XMark data generator [15]) were used; for DBLP, the 127MB dataset [11] and the 411MB dataset [2] were used.

Figure 13 compares compare the total processing time for the input queries: *Original* refers to the non-minimized original query, *CTPQ* refers to the query minimized using CTPQ, and *FBST* refers to the query minimized using FBST. Essentially, the minimization overhead is negligible compared to the gain in total query processing time. FBST gives the best performance compared to both CTPQ and *Original*. Our experimental results demonstrate the effectiveness and efficiency of TPQ minimization using a richer class of constraints.

10. CONCLUSION

In this paper, we examined the problem of tree pattern query minimization using a richer class of integrity constraints (FBST constraints) that includes not only forward and subtype constraints but also backward and sibling constraints. Our study revealed several interesting properties about minimal queries under FBST constraints that makes the query minimization problem more challenging. First, there can be multiple minimal queries of different sizes for an input query; and second, a minimal query can include element types that are not present in the input query. We have characterized the properties of minimal queries for various subclasses of FBST constraints, and have developed efficient algorithms, based on these properties, to both compute a single minimal query as well as enumerate all minimal queries. In addition, we have also developed more efficient minimization algorithms for the previously studied class of FT constraints. Our experimental study demonstrated the effectiveness and efficiency of query minimization using FBST constraints.

11. REPEATABILITY ASSESSMENT RESULT

All the results in the submitted paper were verified by the SIGMOD repeatability committee. However, Figures 12(a)&(b) in the proceedings version of this paper include additional results that have not been verified.

Acknowledgements This research is supported in part by NUS Grant R-252-000-271-112.

12. REFERENCES

[1] The protein sequence database mark-up language. http://pir.georgetown.edu/pirwww/xml/psdml.dtd.

[2] The DBLP XML records. http://dblp.uni-trier.de/xml/.

[3] The Mondial database. http://www.dbis.informatik.uni-goettingen.de/Mondial/.

[4] S. Abiteboul, P. Buneman, and D. Suciu. Data on the web: from relations to semistructured data and XML. *Morgan Kaufman*, 2000.

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

[6] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, 2006.

[7] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. GraphML progress report: Structural layer proposal. *LNCS*, 2002.

[8] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *VLDB*, 2003.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *TODS*, 30(2), 2005.

[10] S. Hinkelman. Business integration - information conformance statements (bi-ics). *IBM*, 2005.

[11] G. Miklau. The XML data repository. 2002. http://www.cs.washington.edu/research/xmldatasets/.

[12] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *JACM*, 51(1), 2004.

[13] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTD, and variables. *LMCS*, 2(3), 2006.

[14] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[15] A. Schmidt, F. Waas, M. Kersten, D. Florescu, and I. Manolescu. The XML benchmark project. Technical Report INS-R0103, CWI, Netherlands, 2001.

[16] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *ICDE*, 2007.

[17] W3C. XML path language (XPath). 1999. http://www.w3.org/TR/xpath.

[18] W3C. XQuery 1.0: An XML query language. 2001. http://www.w3.org/TR/xquery.

[19] P. T. Wood. Minimising simple XPath expressions. In *WebDB*, 2001.

[20] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.