Heuristic Containment Check of Partial Tree-Pattern Queries in the Presence of Index Graphs

Dimitri Theodoratos
Dept. of CS
NJIT
dth@cs.njit.edu

Stefanos Souldatos
Dept of EE & CE
NTUA
stef@dblab.ntua.gr

Theodore Dalamagas
Dept of EE & CE
NTUA
dalamag@dblab.ntua.gr

Pawel Placek
Dept. of CS
NJIT
pp@cs.njit.edu

Timos Sellis
Dept of EE & CE
NTUA
timos@dblab.ntua.gr

ABSTRACT

The wide adoption of XML has increased the interest of the database community on tree-structured data management techniques. Querying capabilities are provided through tree-pattern queries. The need for querying tree-structured data sources when their structure is not fully known, and the need to integrate multiple data sources with different tree structures have driven, recently, the suggestion of query languages that relax the complete specification of a tree pattern. In this paper, we use a query language which allows partial tree-pattern queries (PTPQs). The structure in a PTPQ can be flexibly specified fully, partially or not at all. To evaluate a PTPQ, we exploit index graphs which generate an equivalent set of "complete" tree-pattern queries.

In order to process PTPQs, we need to efficiently solve the PTPQ satisfiability and containment problems. These problems become more complex in the context of PTPQs because the partial specification of the structure allows new, non-trivial, structural expressions to be derived from those explicitly specified in a PTPQ. We address the problem of PTPQ satisfiability and containment in the absence and in the presence of index graphs, and we provide necessary and sufficient conditions for each case. To cope with the high complexity of PTPQ containment in the presence of index graphs, we study a family of heuristic approaches for PTPQ containment based on structural information extracted from the index graph in advance and on-the-fly. We implement our approaches and we report on their extensive experimental evaluation and comparison.

1. INTRODUCTION

The wide adoption of XML has boosted the interest of the database community on tree-structured data management techniques. Querying capabilities are provided through tree-pattern queries. For instance, XPath [1], the core query language of XQuery [2], specifies queries essentially in the form of tree patterns. Answers are computed by matching tree patterns against the data trees. Processing tree pattern queries (TPQs) is central to the optimization of XQuery. For this reason, a lot of attention has been paid recently to the satisfiability, containment, and minimization problems for TPQs in the presence and in the absence of schemas.

Consider the following XPath expression retrieving information about authors from a bibliographic database:
`//subject[/year]//article[/title]//author`.
Figure 1(a) shows the corresponding TPQ (query Q'). Two other TPQs (queries Q'' and Q''') are shown in Figure 1(b) and 1(c), respectively. The nodes are labeled by elements; double line arrows denote descendant relationships, and single line arrows denote child relationships; a star (*) marks the output node (the node returned in the answer). Even though TPQs provide some freedom in the specification of a tree structure (e.g. by allowing ancestor-descendant relationships), they all have a common restrictive requirement: *in every root-to-leaf path, there is a total order for the nodes*. For instance, in the rightmost path of TPQ Q'', node *subject* precedes node *year*, which precedes node *article*, which precedes node *title*. It is not possible in a TPQ to indicate that two nodes n_1 and n_2 occur in a path without specifying a precedence relationship between them: node n_1 has to precede node n_2 or vice-versa.

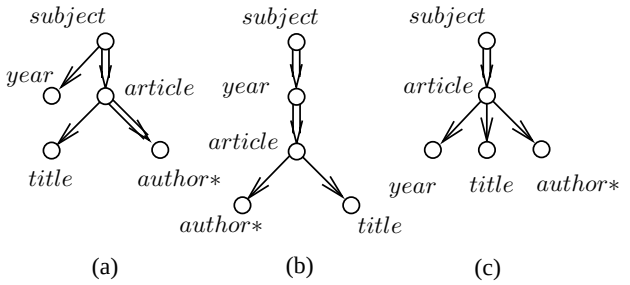

Figure 1: Three TPQs: (a) Q', (b) Q'', (c) Q'''

Such a requirement causes several problems. Consider for example a database that integrates tree-structured data exported from several data sources under different structures. These structures may order elements differently, and a single TPQ is not able to retrieve the desired information from all of them. In addition, users may not be fully aware of the database structure. Usually they are not not able to provide of total order for the elements specified in the query.

Evoking the techniques of previous approaches in this context does not solve these problems: using TPQs in conjunction with traditional data integration mapping rules between a global structure and local structures [7] requires extensive manual intervention, is complex and subject to errors. Approaches that relax the constraints in TPQs [4] result in approximate answers. Finally, structure-less keyword-based approaches [8, 13] do not allow the specification of structural conditions in the queries to filter out undesirable answers.

Our approach. To deal with the problems above, we use here a language of *partial* tree-pattern queries. This language is based on the language introduced in [23] to query tree-structured data using semantic information. Consider the query of Figure 2. This is

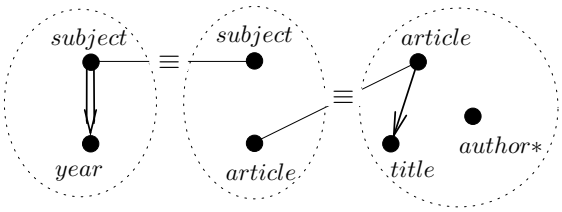

Figure 2: A Partial Tree Pattern Query Q

a tree pattern where the tree structure is partially specified. It has three paths. The first path involves elements *subject* and *year*, and *year* is a descendant of *subject*. The second path involves elements *subject* and *article* and no order is specified between them. The third path involves elements *article*, *title* and *author*. Element *title* is a child of *article*, but *author* can be an ancestor of *article* or a descendant of *title*. The first and the second paths have a common element *subject*, while the second and the third paths have a common element *article*. We call this type of queries Partial Tree-Pattern Queries (PTPQs). *The structure in a PTPQ can be flexibly specified fully, partially or not at all*.

The need for a language that allows a flexible specification of the structure has also been identified in [17]. However, the approach adopted there focuses on reducing the number of query matchings to those that are meaningful with respect to the context of the query. In contrast to our approach, the user has to specify TPQs; that is, the user has to specify an order for all the nodes in each path of the tree pattern.

Problem addressed. Efficient evaluation of queries requires techniques for efficiently processing the queries. In this paper, we address the satisfiability and containment problems for PTPQs and we study properties of PTPQs. These problems become more complex in the context of PTPQs because the partial specification of the structure in the queries allows new, non-trivial, structural expressions to be derived from those explicitly specified in a PTPQ. Therefore, we focus on finding heuristic approaches for checking PTPQ containment in the presence of index graphs.

Contribution. The main contributions of this paper are the following:

- We introduce a set of sound and complete inference rules, in order to deal with the derivation of structural expressions (constraints) in PTPQs. We further define a "normal form" for PTPQs, called full form, to enable the study and comparison of PTPQs. Intuitively, the full form of a PTPQ comprises all the structural expressions that can be derived from those explicitly specified in the PTPQ.
- We define index graphs that summarize the structure of data trees. Even though index graphs are not schemas, we exploit them to evaluate and process PTPQs in the same way schemas are exploited in relational databases.
- We define two types of PTPQ satisfiability: absolute and relative. Relative satisfiability is defined with respect to an index graph. We then provide necessary and sufficient conditions for both types of PTPQ satisfiability. Detecting relative unsatisfiability during the evaluation of a PTPQ prevents the access of the database which is typically much larger than its index graph.
- We define two types of PTPQ containment: absolute query containment and PTPQ containment with respect to an index graph (relative PTPQ containment). We provide necessary and sufficient conditions for checking both types of PTPQ containment.
- In order to deal with the high complexity of relative PTPQ containment, we design a family of sound but not complete heuristic techniques based on structural expressions extracted from the index graph. We devise two ways for applying our heuristic techniques, one that precomputes and stores the extracted structural expressions and one that computes the extracted structural expressions on-the-fly.
- We implemented the containment checking approaches mentioned above (absolute and relative) including all the heuristic ones. We performed an extensive experimental evaluation to compare the pros and cons of every approach. Our experiments show that the heuristic approaches are efficient compared to non-heuristic ones while maintaining high accuracy. These results show that our techniques can be directly exploited for PTPQ processing and optimization.

Paper outline. The next section disusses related work. Section 3 presents our language for PTPQs, studies the inference of structural expressions, and introduces index graphs and additional concepts. Section 4 addresses PTPQ satisfiability check. In Section 5, we study the two types of PTPQ containment checking. Heuristic approaches for checking relative PTPQ containment are described in Section 6. Section 7 analyses the experimental results. We conclude and discuss future work in Section 8. Proofs are omitted because of lack of space. They will be included in the full version of the paper.

2. RELATED WORK.

There is considerable work on the satisfiability [12, 16], containment [9, 18, 25, 20, 10, 6] and minimization problems [3, 22] for tree-pattern queries in the presence and the absence of schemas. None of these papers addresses query containment for PTPQs. Most of these works focus on studying the complexity of these problems for different classes of TPQs. Our goal in this paper is different. We are focusing on providing sound (but not necessarily complete) techniques for checking query containment that can be used for efficiently processing and optimizing PTPQs. To the best of our knowledge, this issue has not been addressed previously for this type of queries.

Index graphs have been referred to with different names in the literature, including "path summaries", "path indexes" and "structural summaries". They differ in the equivalence relations they employ to partition the nodes of the database, which includes simulation and bisimulation [19, 14], or even semantic equivalence relations [23]. Index graphs have been extensively studied in recent years in both the "exact" [11, 19, 21, 5] and the "approximate" flavor [15, 14]. A common characteristic of those approaches is that the index graph is used as a back end for evaluating a class of path expressions without accessing the database. To this end, the equivalence classes of database nodes are attached to the corresponding index graph nodes. For the needs of PTPQs, we define index graphs where the equivalence classes are formed by all the nodes labeled by the same element in the database. In contrast to other approaches, here, the equivalence classes of the database nodes are not kept with the index graph. Therefore, PTPQs are ultimately evaluated on the database tree. The index graphs are used to support the evaluation of PTPQs and the satisfiability and containment checking.

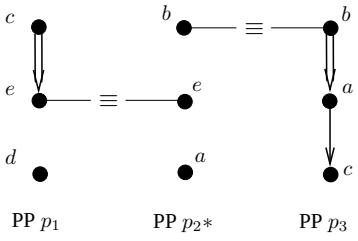

Figure 3: Query Q_1

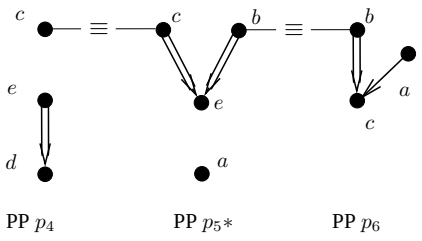

Figure 4: Query Q_2

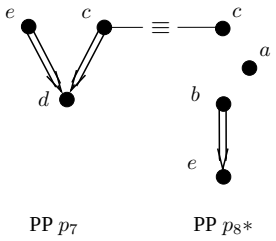

Figure 5: Query Q_3

Partially specified tree-pattern queries were initially introduced in [23]. That query language assumed a partitioning of the nodes of a data tree based on semantic information. Subsequently, this semantic information was taken into account for querying the data tree. On the contrary, in this paper we view data trees as XML documents, and we partition their nodes using merely their labeling elements (tags). This modifies also the way structural expressions can be derived. The containment problem for the class of queries considered in [23] was addressed in [24]. That paper does not address heuristic approaches for the containment of PTPQs which is the main focus of the present work.

3. THE PARTIAL TREE PATTERN QUERY LANGUAGE

We present in this section the data model and our query language that allows for partially specifying the structure of a tree pattern. Our goal in this paper is to focus on the structural aspects of the queries. Therefore, we retain only structural expressions and we abstract attribute and value restrictions, and variables that a full-fledged query language would normally comprise.

3.1 Data Model and Query Language

Let $\mathcal{E}$ be an infinite set of elements that includes a distinguished element r. A *database* is a finite tree of nodes labeled by elements in E, rooted at a node labeled by r (such a root node can always be added to a data tree if it is not initially there). For simplicity, we assume that the same element does not label two nodes on the same path. The attributes of an XML document are modeled here using the element nodes of such a tree.

DEFINITION 3.1. A *Partial Tree-Pattern Query* (PTPQ) is a triple $Q = (\mathcal{P}, \mathcal{N}, o)$, where:

(a) $\mathcal{P}$ is a nonempty set of pairs $(p, \mathcal{R})$ called *Partial Paths* (PPs). p is the name of the PP. $\mathcal{R}$ is a set of expressions of the form $e_i \rightarrow e_j$ (*child precedence relationship*) and/or $e_i \Rightarrow e_j$ (*descendant precedence relationship*), where e_i and e_j are elements. The names of the PPs in Q are distinct. Therefore, we identify PPs in Q with their names. The expression $e[p]$ denotes the element e in PP p.

(b) $\mathcal{N}$ is a set of expressions of the form $e[p_i] \equiv e[p_j]$, where p_i and p_j are PPs in $\mathcal{P}$, and e is an element. These expressions are called *node sharing expressions*. Roughly speaking, they state that PPs p_i and p_j have element e in common (they share it). Set $\mathcal{N}$ can be empty.

(c) o is a PP in $\mathcal{P}$. It is called *output* PP of Q. □

We graphically represent PTPQs using graph notation. Each PP of a PTPQ Q is represented as a (not necessarily connected) graph of elements. The name of each PP is shown by the corresponding PP graph. The name of the output PP of Q is followed by a '*'. Child and descendant precedence relationships in a PP are depicted using single ($\rightarrow$) and double ($\Rightarrow$) arrows between the respective elements in the PP graph. In particular, descendant precedence relationships of the form $r \Rightarrow e$ in a PP are shown only with the presence of element e in the PP graph. A node sharing expression $e[p_i] \equiv e[p_j]$ is represented by an edge between element e of the PP graph p_i and element e of the PP graph p_j labeled by $\equiv$. Figures 3, 4, and 5 show three PTPQs.

The answer of a PTPQ is based on the concept of PTPQ embedding.

DEFINITION 3.2. An *embedding* of a PTPQ Q to a database D is a mapping M of the elements of the PPs of Q to nodes in D such that: (a) an element e of Q is mapped by M to a node in D labeled by e; (b) the elements of a PP in Q are mapped by M to nodes in D that are on the same path; (c) $\forall\, e_i[p] \rightarrow e_j[p]$ (resp. $e_i[p] \Rightarrow e_j[p]$) in Q, $M(e_j[p])$ is a child (resp. descendant) of $M(e_i[p])$ in D; and (d) $\forall e[p_i] \equiv e[p_j]$ in Q, $M(e[p_i])$ and $M(e[p_j])$ coincide. □

We call *image* of a PP p in Q under M, denoted $M(p)$, the path from the root of D that comprises all the images of the elements of p under M and ends in one of them. Notice that more than one PP of Q may have their image on the same root-to-leaf path of D (M does not have to be a bijection). The *answer* of Q on D is the set of the images of the output PP of Q under all possible embeddings of Q to D.

Notice that the PTPQ language allows the formulation of queries with no structure at all by specifying a single node per PP and no node sharing expressions. This resembles a flat keyword-based query. However, unlike such a query, a PTPQ returns a structured answer. On the other side, the PTPQ language also allows the formulation of queries that are completely structured trees by specifying only child relationships and node sharing expressions. Between the two extremes, there are PTPQs that provide some description of the structure without completely specifying a tree.

3.2 Structural Expression Derivation

Precedence relationships and node sharing expressions are collectively called *structural expressions*. Because the structure of tree patterns is partially specified in PTPQs, new, non-trivial structural expressions can be derived from those explicitly specified in the queries. Many of these derivations are specific to PTPQs. Consider, for instance, query Q_1 of Figure 3. Since, $c[p_1] \Rightarrow e[p_1]$ and $e[p_1] \equiv e[p_2]$, we can derive $c[p_2] \Rightarrow e[p_2]$ (that is, c occurs in PP p_2). Since, $b[p_3] \Rightarrow a[p_3]$ and $a[p_3] \rightarrow c[p_3]$, we can derive $b[p_3] \Rightarrow c[p_3]$. Since, $b[p_3] \Rightarrow a[p_3]$, $b[p_3] \equiv b[p_2]$ and a occurs in PP p_2, we can derive $b[p_2] \Rightarrow a[p_2]$ (indeed, since $b[p_2] \equiv b[p_3]$, if $a[p_2] \Rightarrow b[p_2]$, we can derive $a[p_3] \Rightarrow b[p_3]$, a contradiction). Checking efficiently structural expression derivation is central in processing PTPQs. In this section, we address this problem.

Let S be a set of structural expressions of a PTPQ Q, and s be a structural expression. We say that s is *derived* from S iff for every embedding M of Q to a database, M satisfies s. The *closure* of S is the set that comprises exactly the structural expressions in S and all the structural expressions that can be derived from S.

To compute closures of sets of structural expressions, we introduce a set of inference rules. Let a, b, c and d be distinct elements and p, p_1, and p_2 be distinct PPs. We use the symbol $\vdash$ to denote that the expressions that precede it produce the expression that follows it. The absence of expressions that precede $\vdash$ denotes an axiom. Figure 6 shows a set of 14 inference rules.

(IR1) $\vdash r[p_1] \equiv r[p_2]$
(IR2) $a[p_1] \equiv a[p_2],\ a[p_2] \equiv a[p_3] \vdash a[p_1] \equiv a[p_3]$
(IR3) a *structural expression that involves* $a[p] \vdash r[p] \Rightarrow a[p]$
(IR4) $a[p] \rightarrow b[p] \vdash a[p] \Rightarrow b[p]$
(IR5) $a[p] \Rightarrow b[p],\ b[p] \Rightarrow c[p] \vdash a[p] \Rightarrow c[p]$
(IR6) $a[p] \rightarrow b[p],\ a[p] \Rightarrow c[p] \vdash b[p] \Rightarrow c[p]$
(IR7) $a[p] \rightarrow b[p],\ c[p] \Rightarrow b[p] \vdash c[p] \Rightarrow a[p]$
(IR8) $a[p_1] \rightarrow b[p_1],\ b[p_1] \equiv b[p_2] \vdash a[p_2] \rightarrow b[p_2]$
(IR9) $a[p_1] \Rightarrow b[p_1],\ b[p_1] \equiv b[p_2] \vdash a[p_2] \Rightarrow b[p_2]$
(IR10) $a[p_1] \Rightarrow b[p_1],\ a[p_1] \equiv a[p_2],\ r[p_2] \Rightarrow b[p_2] \vdash a[p_2] \Rightarrow b[p_2]$
(IR11) $a[p_1] \Rightarrow b[p_1],\ b[p_1] \equiv b[p_2] \vdash a[p_1] \equiv a[p_2]$
(IR12) $a[p_1] \rightarrow b[p_1],\ c[p_2] \rightarrow b[p_2],\ d[p_1] \equiv d[p_2] \vdash d[p_1] \Rightarrow a[p_1]$
(IR13) $a[p_1] \rightarrow b[p_1],\ a[p_2] \rightarrow c[p_2],\ d[p_1] \equiv d[p_2] \vdash d[p_1] \Rightarrow a[p_1]$
(IR14) $a[p_1] \Rightarrow b[p_1],\ b[p_2] \Rightarrow a[p_2],\ c[p_1] \equiv c[p_2] \vdash c[p_1] \Rightarrow a[p_1]$

Figure 6: A set of inference rules

The next theorem states that these rules correctly and completely characterize structural expression derivation.

THEOREM 3.1. Let S be a set of structural expressions of a query, and s be a structural expression not in S. The set of inference rules of Figure 6 is *sound* (if s can be produced from S using the inference rules, then s can also be derived from S), and *complete* (if s can be derived from S, then s appears in S, or can also be produced from S using the inference rules). □

To study properties of PTPQs, we introduce a "normal form" for PTPQs called *full form*. A PTPQ Q is in full form if the set S of structural expressions of Q equals the closure of S. Clearly, a PTPQ can be equivalently put in full form by replacing its set S of structural expressions in Q by the closure of S.

Figures 7 and 8 show the full form of the PTPQs Q_1 and Q_2 of Figures 3 and 4 respectively. Query Q_3 of Figure 5 is in full form. For clarity of presentation, when graphically representing queries in full form, we do not depict structural expressions that can be inferred trivially from the shown structural expressions using the inference rules IR1 - IR5.

Clearly, there can be only a polynomial number of structural expressions in the closure of a set of structural expressions. In practice, only a small percentage of the maximal possible number of expressions appear in the closure of the structural expressions of a PTPQ, and therefore, the cost of computing the full form of a PTPQ is insignificant.

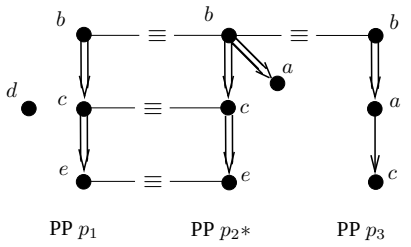

Figure 7: The full form of Query Q_1

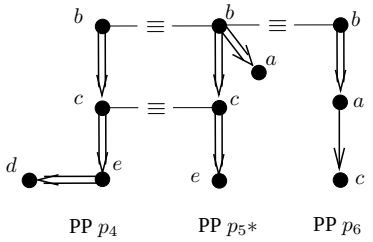

Figure 8: The full form of Query Q_2

3.3 Index Graphs and Sets of Complete TPQs

Given a partitioning of the nodes of a database D, an index graph for D is a graph G such that: (a) every node in G is associated with a distinct equivalence class of nodes in D, and (b) there is an edge in G from the node associated with the equivalence class a to the node associated with the equivalence class b, iff there is an edge in D from a node in a to a node in b. Figure 9 shows an index graph G. Even though the index graph for a database is not a schema in the form of a DTD or an XML Schema, we take advantage of it in the same way schema information is exploited in relational databases. We use index graphs to support the evaluation of a PTPQs through the generation of an equivalent set of complete TPQs. We also use index graphs to process PTPQs; this comprises identifying *valid clusters* (introduced later) in a PTPQ, checking a PTPQ for satisfiability, and checking two PTPQs for containment.

If G is the index graph of a database D, we say that D *underlies* G. Given a PTPQ Q and an index graph G, Q can be evaluated by computing a set of complete TPQs whose answers taken together are equal to the answer of Q on any database underlying G. By *complete* TPQ, we mean a TPQ that involves only child relationships (and therefore, completely specifies a tree pattern.) A *complete* TPQ *for* Q on G is a TPQ T rooted at a node labeled by r that satisfies the following conditions: (a) T has a distinguished node called *output node* that defines a from-the-root path called *output path* (b) there is a mapping M from the nodes of Q to the nodes of T that respects paths, output paths, labeling elements, precedence relationships, and node sharing expressions, and (c) there is a mapping M' from the nodes of T to the nodes of G that respects

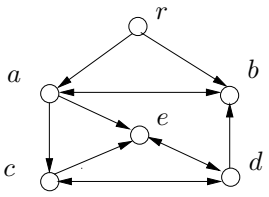

Figure 9: Index graph G

labeling elements and child precedence relationships. To minimize the set of TPQs that are able to compute the answer set of Q, we further require for T that: (a) every leaf node of T is the image of a node of Q under M, and (b) the images under M of two nodes $d[p_1]$ and $d[p_2]$ of Q do not coincide unless $d[p_1] \equiv d[p_2]$ can be derived from the set of structural expressions of Q. Figure 10 shows the complete TPQs T_1^1 and T_1^2 for the PTPQ Q_1 of Figure 3 on the index graph G of Figure 9. Notice that it is possible that all the nodes of two distinct PPs of Q are mapped by M to nodes on the same path in T (e.g. this is the case with PP p_1 and p_2 of Q_1 which are mapped on the same path in T_1^1 and T_1^2.)

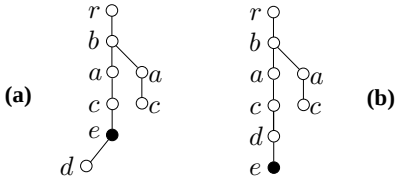

Figure 10: The complete TPQs of PTPQ Q_1 on index graph G: (a) T_1^1, (b) T_1^2

In terms of PTPQs, the output PTPQ of a complete TPQ T for a PTPQ on G is the path from the root of T to the output node of T. The answer of a complete TPQ T is defined similarly to that of a PTPQ. We can now show the following proposition.

PROPOSITION 3.1. Let Q be a PTPQ, G be an index graph, and $T^1, \ldots, T^k,\ k \geq 1$, be the complete TPQs of Q on G. Let also $A,\ A_1, \ldots, A_k$ be the answers of $Q,\ T^1, \ldots, T^k$, respectively, on a database underlying G. Then A = $\cup_{i \in [1,k]} A_i$. □

3.4 Additional Concepts and Assumptions

The presence of two node sharing expressions $a[p_1] \equiv a[p_2]$ and $b[p_2] \equiv b[p_3]$ in a PTPQ Q with no derived precedence relationship between $a[p_2]$ and $b[p_2]$ can create discrepancies: it may force every tree in which there is an embedding of Q to comprise a path that involves a number of dimensions which together do not appear in any PP of Q. A similar phenomenon appears when two node sharing expressions $a[p_1] \equiv a[p_2]$ and $b[p_1] \equiv b[p_2]$ appear in a PTPQ Q along with the child precedence relationships $a[p_1] \rightarrow a_1[p_1]$ and $b[p_2] \rightarrow b_1[p_2]$. Then, every tree in which there is an embedding of Q comprises a path that satisfies $a \rightarrow a_1$ and $b \rightarrow b_1$ even though these child precedence relationships do not appear together in any PP of Q. For simplicity, when checking absolute PSTP containment, we assume that: (a) if $a[p_1] \equiv a[p_2]$ and $b[p_2] \equiv b[p_3]$ appear in a PSTP Q, elements a and b appear in both PPs p_1 and p_3, and (b) if $a[p_1] \equiv a[p_2]$ and $b[p_1] \equiv b[p_2]$ appear in Q along with the child precedence relationships $a[p_1] \rightarrow a_1[p_1] \rightarrow \ldots \rightarrow a_k[p_1]$, and $b[p_2] \rightarrow b_1[p_2] \rightarrow \ldots \rightarrow b_m[p_2]$, the child precedence relationships $a[p_2] \rightarrow a_1[p_2] \rightarrow \ldots \rightarrow a_k[p_2]$, and $b[p_1] \rightarrow b_1[p_1] \rightarrow \ldots \rightarrow b_m[p_1]$ also appear in Q.

A *cluster* is a set C of PPs and node sharing expressions such that for every partition of C in two non-empty sets there is a node sharing expression on an element different than r that involves PPs from both sets (that is, the cluster does not comprise disconnected sets of PPs). Given an index graph G, a cluster is *valid* w.r.t. G if for every database D underlying G, there is an embedding of C into D. Figure 11 shows a valid cluster w.r.t. the index graph of Figure 9. The reader can verify that C is valid since for every path p from the root of G that comprises the edge $d \rightarrow e$ (there are two of them $r \rightarrow a \rightarrow c \rightarrow d \rightarrow e$ and $r \rightarrow b \rightarrow a \rightarrow c \rightarrow d \rightarrow e$) there is an element and precedence relationship preserving mapping of the nodes of C to the nodes of p.

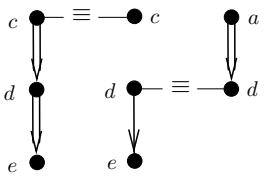

Figure 11: A valid cluster C

Valid clusters can be detected and removed from a PTPQ to yield an equivalent PTPQ. Therefore, in checking relative PTPQ containment, we assume that a PTPQ Q does not contain disconnected valid clusters that do not comprise the output PP of Q.

4. CHECKING PTPQ SATISFIABILITY

A PTPQ is *satisfiable* iff it has a non-empty answer on some database. In the presence of an index graph G, we say that a query is *satisfiable w.r.t.* G iff it has a non-empty answer on some database underlying G. We qualify the first type of PTPQ satisfiability as *absolute* and the second as *relative*. Clearly, a satisfiable query w.r.t. G is also satisfiable. The opposite is not necessarily true. As an example, adding the precedence relationship $e[p_2] \Rightarrow a[p_2]$ to PP p_2 of query Q_1 of Figure 3, results in a query Q_1' unsatisfiable w.r.t. index graph G of Figure 9. Nevertheless, Q_1' is absolutely satisfiable. Adding the precedence relationship $c[p_2] \Rightarrow b[p_2]$ to PP p_2 of query Q_1 of Figure 3, results in an unsatisfiable query.

Absolute satisfiability. PTPQ Q_1 becomes unsatisfiable after the addition of $c[p_2] \Rightarrow b[p_2]$ because $c[p_2] \Rightarrow b[p_2]$ contradicts the descendant precedence relationship $b[p_2] \Rightarrow c[p_2]$ that can be derived from the set of structural expressions of Q_1 (see the full form of Q_1 in Figure 7). This condition is necessary as the next proposition shows.

PROPOSITION 4.1. A PTPQ is unsatisfiable iff two contradicting descendant precedence relationships $a[p] \Rightarrow b[p]$ and $b[p] \Rightarrow a[p]$ (for the same PP p) appear in its full form.

Detecting an absolutely unsatisfiable PTPQ avoids evaluating the PTPQ to compute an empty answer. The overhead for this check amounts to computing the full form of the query which, in practice, is insignificant.

Relative satisfiability. Turning to relative satisfiability (satisfiability w.r.t. an index graph G), Proposition 3.1 suggests that if the set of complete TPQs of a PTPQ Q on G is empty, Q is unsatisfiable w.r.t. G. The following proposition shows that this condition is also necessary.

PROPOSITION 4.2. Let Q be a PTPQ and G be an index graph. PTPQ Q is unsatisfiable w.r.t. G iff there is no complete TPQ for Q on G. □

Query Q_1' resulting by adding the descendant precedence relationship $e[p_1] \Rightarrow a[p_1]$ to PP p_1 of query Q_1 of Figure 3 is unsatisfiable w.r.t. the index graph G of Figure 9: query Q_1 has two

complete TPQs on G and both of them violate the precedence relationship $e \Rightarrow a$. Therefore, Q'_1 has no complete TPQ w.r.t. G.

Relative unsatisfiability of a PTPQ is detected during the generation of its complete TPQs on the index graph. This detection saves accessing the database which is typically orders of magnitude larger than its index graph.

Checking PTPQ containment subsumes checking PTPQ satisfiability: a PTPQ is unsatisfiable (w.r.t. an index graph G) iff it is contained in an unsatisfiable PTPQ (w.r.t. G). In studying PTPQ containment, we assume, in the following, that PTPQs are satisfiable (w.r.t. G).

5. CHECKING PTPQ CONTAINMENT

Let Q_1 and Q_2 be two PTPQs. Q_1 *contains* Q_2 (denoted $Q_2 \subseteq Q_1$) iff for every database D, the answer of Q_2 on D is a subset of the answer of Q_1 on D. PTPQs Q_1 and Q_2 on $\mathcal{D}$ are *equivalent* (denoted $Q_2 \equiv Q_1$) iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. This type of PTPQ containment is called *absolute*. To exploit the presence of an index graph we also define PTPQ containment w.r.t. an index graph G: Q_1 *contains* Q_2 *w.r.t.* G (denoted $Q_2 \subseteq_G Q_1$) iff for every database D underlying G, the answer of Q_2 on D is a subset of the answer of Q_1 on D. PTPQs Q_1 and Q_2 on $\mathcal{D}$ are *equivalent w.r.t.* G (denoted $Q_2 \equiv_G Q_1$) iff $Q_1 \subseteq_G Q_2$ and $Q_2 \subseteq_G Q_1$. The latter type of PTPQ containment is characterized as *relative*. An unsatisfiable PTPQ (w.r.t. an index graph G) is contained in any PTPQ (w.r.t. G). Similar to PTPQ satisfiability, absolute containment implies relative containment, while the opposite is not necessarily true.

Considering the PTPQs Q_1, Q_2 and Q_3 of Figures 3, 4, and 5 and the index graph G of Figure 9, one can see that $Q_2 \subseteq Q_3$. Therefore, $Q_2 \subseteq_G Q_3$. However, $Q_3 \not\subseteq Q_2$. In contrast, $Q_3 \subseteq_G Q_2$. Therefore, $Q_2 \equiv_G Q_3$. PTPQs Q_1 and Q_2 (and Q_1 and Q_3) are not related in terms of absolute or relative containment.

Checking absolute query containment. In order to deal with absolute query containment we need the concept of homomorphism between PTPQs:

DEFINITION 5.1. Let Q_1 and Q_2 be two queries on $\mathcal{D}$. An *homomorphism* from Q_2 to Q_1 is a mapping H from the nodes of Q_2 to the nodes of Q_1 such that: (a) nodes of Q_2 are mapped by H to nodes of Q_1 labeled by the same element, (b) nodes of Q_2 on the same PP are mapped by H to nodes of Q_1 on the same PP, (c) the nodes in the output PP o_2 of Q_2 are mapped under H to nodes in the output PP o_1 of Q_1, and every node in o_1 is the image under H of a node in o_2, (d) $\forall\ e_i[p] \rightarrow e_j[p]$ (resp. $e_i[p] \Rightarrow e_j[p]$) in Q_2, $H(e_i[p]) \rightarrow H(e_j[p])$ (resp. H$(e_i[p]) \Rightarrow H(e_j[p])$) is in Q_1, and (e) $\forall\ e[p_i] \equiv e[p_j]$ in Q_2, $H(e[p_i])$ and $H(e[p_j])$ coincide or $H(e[p_i]) \equiv H(e[p_j])$ is in Q_1. □

As we mentioned above, the PTPQ Q_2 of Figure 4 is absolutely contained in the PTPQ Q_3 of Figure 5. One can see that there is a homomorphism from Q_3 to Q_2 that maps the nodes of PP p_7 of Q_3 to those of PP p_4 of Q_2 and the nodes of PP p_8 (the output PP of Q_3) to those of PP p_5 (the output PP of Q_2). The next theorem shows that the existence of an homomorphism between PTPQs is a necessary and sufficient condition for absolute query containment.

THEOREM 5.1. Let Q_1 and Q_2 be two PTPQs in full form. $Q_1 \subseteq Q_2$ if and only if there is a homomorphism from Q_2 to Q_1. □

Notice the absence of an homomorphism from Q_2 to Q_1 only because $e[p_4] \Rightarrow d[p_4]$ in Q_2 cannot be mapped to a precedence relationship in Q_1. This explains our earlier claim in this section that $Q_1 \not\subseteq Q_2$.

Checking relative query containment The next theorem provides necessary and sufficient conditions for relative PTPQ containment, in terms of absolute containment of complete TPQs.

THEOREM 5.2. Let Q_1 and Q_2 be two PTPQs and G be an index graph. Let also $\mathcal{T}_1$ be the set of complete TPQs of Q_1 on G, and $\mathcal{T}_2$ be the set of complete TPQs of Q_2 on G. $Q_1 \subseteq_G Q_2$ if and only if there is a mapping f from $\mathcal{T}_1$ to $\mathcal{T}_2$ such that, for every complete TPQ T in $\mathcal{T}_1$, $T \subseteq f(T)$. □

Consider again the PTPQs Q_1, Q_2 and Q_3 of Figures 3, 4, and 5. Their complete TPQs on the index graph G of Figure 9 are shown in the figures 10 and 12.

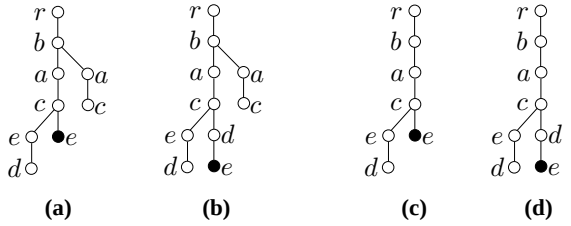

Figure 12: The complete TPQs of PTPQ Q_2 on index graph G: (a) T_2^1, (b) T_2^2. The complete TPQs of PTPQ Q_3 on index graph G: (c) T_3^1, (d) T_3^2

Theorem 5.2 proves our earlier claim in this section that Q_2 and Q_3 are equivalent w.r.t. G: considering the bijection $f(T_2^1) = T_3^1$ and $f(T_2^2) = T_3^2$, one can easily verify, based on Theorem 5.1, that T_3^1 and T_2^1 are absolutely equivalent and the same holds for T_3^2 and T_2^2. In contrast, one can see that $Q_2 \not\subseteq_G Q_1$ only because the node sharing expressing on e in T_1^1 cannot be mapped through an homomorphism to a node sharing expression on e in T_2^1.

6. HEURISTIC APPROACHES FOR RELATIVE QUERY CONTAINMENT

Checking relative PTPQ containment can be time consuming since, as we saw in the previous section, it involves checking several pairs of complete TPQs for absolute containment. In contrast, checking absolute PTPQ containment requires only the detection of an homomorphism between the two PTPQs. In this section, we suggest heuristic approaches for checking query containment w.r.t. an index graph G. These heuristic approaches reduce relative PTPQ containment to absolute PTPQ containment.

Suppose that Q_1 and Q_2 are two PTPQs and we want to check whether $Q_1 \subseteq_G Q_2$. The basic idea is to extract all the precedence relationships that hold on the all paths from the root of G under the condition that some other precedence relationships hold these paths. This information is represented in the form of *rule instances* where the extracted precedence relationships form the conclusion, and the conditional precedence relationships form the premise. For example, the reader can verify that the rule instance $\{e \Rightarrow d\} \Longrightarrow \{e \rightarrow d\}$ holds on the index graph of Figure 9.

Given a set of rule instances that hold on G, if the precedence relationships in the premise of a rule instance R can be derived from the precedence relationships in a PP p of Q_1, the precedence relationships in the conclusion of R are added to PP p. This process is repeated until a fixed-point Q'_1 is reached. PTPQ Q'_1 is called *augmented* Q_1. Clearly, $Q'_1 \equiv_G Q_1$. Then, Q'_1 is checked for *absolute* containment into Q_2.

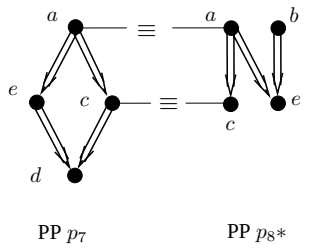

Figure 13: Aug. Q_3 w.r.t. $R1$ and G

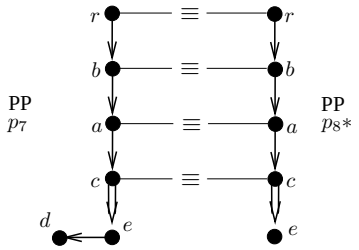

Figure 14: Aug. Q_3 w.r.t. $R3$ and G

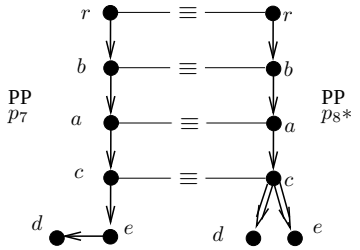

Figure 15: Aug. Q_3 w.r.t. G

Rule instances are grouped into *rules* which are patterns of rule instances. The following are rules:
$R1 : \{r \Rightarrow x\} \Longrightarrow \{y \rightarrow x, y \Rightarrow x\}$,
$R2 : \{r \Rightarrow x, r \Rightarrow y\} \Longrightarrow \{u \rightarrow v, u \Rightarrow v\}$, and
$R3 : \{x \Rightarrow y\} \Longrightarrow \{u \rightarrow v, u \Rightarrow v\}$

For example, the rule $R1$ groups together all the rule instances that extract child and descendant precedence relationships from any node *to* a node x. The instances of $R1$ that hold on the index graph G of Figure 9 and extract non-empty sets of precedence relationships from G are:
$\{r \Rightarrow c\} \Longrightarrow \{r \Rightarrow c, a \Rightarrow c\}$,
$\{r \Rightarrow e\} \Longrightarrow \{r \Rightarrow e, a \Rightarrow e\}$, and
$\{r \Rightarrow d\} \Longrightarrow \{r \Rightarrow d, a \Rightarrow d\}$.
An example of an instance of $R3$ that holds on G is:
$\{b \Rightarrow e\} \Longrightarrow \{r \Rightarrow b, r \Rightarrow e, r \Rightarrow a, r \rightarrow b, b \rightarrow a\}$.
The reader is encouraged to find the rest of the instances for rules $R1$, $R2$ and $R3$ that hold on G and extract non-empty sets of precedence relationships.

Consider a PTPQ Q, an index graph G, and a rule R. The *augmented* query Q (say Q') w.r.t. R and G is constructed from Q as follows:

```
Let initially Q' = Q.
Repeat the following steps until no more changes
can be applied to Q':
   - Let Q' be the full form of Q'.
   - For every PP p in Q' and for every instance
     P ⟹ C of R that holds on G, if the prece-
     dence relationships in P appear in p, add to
     p the precedence relationships in C.
```

Figures 13 and 14 show the augmented Q_3 w.r.t. R_1 and G and w.r.t. R_3 and G respectively. The augmented Q_3 w.r.t. G and R_2 is identical to the augmented Q_3 w.r.t. G and R_1.

As mentioned in Section 5, absolute containment implies relative containment with respect to any dimension graph, and consequently w.r.t. G. Therefore, if $Q'_1 \subseteq Q_2$, it is guaranteed that $Q_1 \subseteq_G Q_2$. As stated in the following proposition, this approach is sound but not complete: if $Q'_1 \not\subseteq Q_2$, it is not guaranteed that $Q_1 \not\subseteq_G Q_2$.

PROPOSITION 6.1. Let Q_1 and Q_2 be two PTPQs, G be an index graph, and R be a rule. Let also Q'_1 be the augmented Q_1 w.r.t. R and G. If $Q'_1 \subseteq Q_2$ then $Q_1 \subseteq_G Q_2$. If $Q'_1 \not\subseteq Q_2$, then it is possible that $Q_1 \subseteq_G Q_2$. □

Let Q'_3 be the augmented PTPQ Q_3 shown in Figure 14. One can easily see that there is an homomorphism from Q_2 to Q'_3. Based on Theorem 5.1, $Q'_3 \subseteq Q_2$. Then, Proposition 6.1 guarantees that $Q_3 \subseteq_G Q_2$. This is precisely what we proved in Section 5.

We have employed two heuristic approaches, a precomputation and an on-the-fly approach, presented below:

Precomputation heuristic approach. In the precomputation heuristic approach, precedence relationships are extracted from the index graph in advance. When a query emerges, these precedence relationships are ready for immediate application.

Using additional rules in the heuristic approach improves its accuracy (percentage of pairs of queries detected with relative containment out of a set of pairs of queries that are relatively but not absolutely contained). Consider, for example, the rule $R1$ and the augmented PTPQ Q_3 w.r.t. $R1$ and G shown in Figure 13. This PTPQ is not contained in Q_2. Therefore, a heuristic approach that uses only $R1$ fails to detect the relative containment in this case. However, if the heuristic approach employs in addition rule $R3$, the augmented PTPQ Q_3 w.r.t. $\{R_1, R3\}$ and G (which is the same as the one shown in Figure 14) succeeds in detecting containment. These gains in accuracy are obtained at the expense of (a) extra space for storing the additional rule instances, and (b) additional time for computing the augmented query (possibly more rule instances to be checked for application, more precedence relationships to be added to the PTPQ, and more iterations in the computation of the augmented PTPQ). Therefore, a trade-off has to be determined between desired accuracy on the one side and space and time resources on the other side.

Using several rules, a family of heuristic approaches can be defined, each approach involving several rules, which have to be tested for appropriateness. Skipping the details because of space consideration, we only mention that in constructing such a family of heuristics, the designer has to (a) exclude rules that are redundant in the presence of other rules, and (b) minimize the effect of the extensive horizontal (among rule instances of the same rule) and vertical (among instances of different rules) overlapping of the rule instances by employing incrementality in storing the rule instances and in applying them to the PTPQs. We have used such incremental techniques in our implementation and experiments presented in the next section.

On-the-fly heuristic approach. Using all the precedence relationships in every PP of the full form of a PTPQ Q, we can extract more precedence relationships from G. Adding iteratively those precedence relationships to Q until a fixed-point is reached results in a query called augmented Q w.r.t. G. Clearly, the augmented Q w.r.t. G is not less restrictive than the augmented Q w.r.t. any set of rules and G. Consider, for example, PTPQ Q_3 of Figure 5. Figure 15 shows the augmented Q_3 w.r.t index graph G of Figure 9. This PTPQ is more restrictive than the augmented Q_3 w.r.t. $R3$ and G shown in Figure 14. Nevertheless, this heuristic approach can only be applied after the query is issued and it is subject to the additional cost of extracting the precedence relationships on-the-fly and computing the augmented query w.r.t. G.

7. EXPERIMENTAL EVALUATION

To study the effectiveness of our PTPQ containment checking techniques, we ran a comprehensive set of experiments. Checking PTPQ containment in the presence of index graphs (relative containment), is expected to be time consuming compared to checking PTPQ containment in the absence of index graphs (absolute containment). However, our experimental evaluation shows that the heuristic approaches for checking relative containment can save a considerable amount of time, while maintaining high accuracy.

Setup. We ran our experiments on a dedicated Linux PC (AMD Sempron $2600+$) with $2GB$ of RAM. The reported values are the average of repeated measurements. Specifically, for every measure point, 100 pairs of queries were generated (10 pairs of queries for each one of the 10 index graphs used). Only satisfiable queries w.r.t. the index graphs were generated. For all pairs (Q_1, Q_2) of PTPQs used for containment check, $Q_1 \subseteq_{\mathcal{G}} Q_2$, but $Q_1 \nsubseteq Q_2$.

We used index graphs whose number of root-to-leaf paths does not exceed five times the number of their nodes. This is in conformance with the index graphs of several popular XML benchmarks, like XMark[1] and XMach[2], where the number of root-to-leaf paths does not exceed twice the number of their nodes.

Experiments. In our experiments, we compared the execution time and the accuracy for containment check among the following cases: (a) PTPQ containment in the absence of an index graph (Absolute Containment - AC), (b) query containment in the presence of an index graph (Relative Containment - RC), (c) heuristic query containment in the presence of an index graph, where precedence relationships are extracted from the index graph on-the-fly ($RCFH$), (d) three approaches of heuristic query containment in the presence of an index graph, where the precedence relationships extracted from the index graph are precomputed (RCH_1, RCH_2, RCH_3). Approach RCH_1 uses only the rule $R1$. Approach RCH_2 uses the rules $R1$ and R_2. Approach RCH_3 uses the rules $R1$, R_2 and $R3$ (see Section 6).

We also tested the impact of the index graph's and queries' density on the execution time and on the accuracy for contaiment check among the above casesfor different structures of the index graph and of the queries. Next, we present the detailed results.

Execution time and accuracy varying the density of the index graph. We measured the execution time and the accuracy for checking PTPQ containment varying the number of root-to-leaf paths for different numbers of nodes in the index graph. In Figures 16 and 17, we present the results obtained for index graphs having 20, 30 and 40 nodes. The number of PPs in the queries and the number of nodes per PP are fixed to 2 and 4, respectively.

As expected, RC check is time consuming compared to AC check. The larger the number of paths in the index graph, the more is the time taken by RC check. This is due to the increase in the number of matchings of the PPs to the paths of the index graph. Such an increase causes more complete TPQs to be produced. Obviously, the execution time for AC check is not affected by the index graph parameters.

Overall, our results show that all of our heuristic techniques clearly improve checking of RC. Note that RCH_1 is the fastest among all the heuristic containment checks we suggest, giving in some cases an improvement of more than two orders of magnitude compared to RC check.

The execution time for RCH_1, RCH_2 and RCH_3 checks slightly drops as the number of paths in the index graph increases. The reason is that the density of the index graph increases, too, which in turn decreases the number of precedence relationships extracted from the graph. Note that the precedence relationships from the index graph are precomputed. Thus, the execution time does not include the time required to extract the precedence relationships from the index graph.

For a growing number of paths in the index graph, the execution time for the on-the-fly heuristic containment check $RCFH$ increases. This is caused by the increase in the number of paths examined during the (on-the-fly) precedence relationship extraction.

Regarding the accuracy, the on-the-fly heuristic containment approach $RCFH$ is clearly more accurate than all the other heuristic containment checking approaches, approximating 100% of the accuracy of the non-heuristic relative containment check RC. Heuristics RCH_1, RCH_2 and RCH_3 have an accuracy higher than 45%, 65% and 85%, respectively, for index graphs whose number of root-to-leaf paths does not exceed twice the number of their nodes.

Execution time and accuracy varying the density of queries. We measured the execution time and the accuracy for checking PTPQ containment varying the number of nodes per PP for different numbers of PPs in the query. In Figures 18 and 19, we present the results obtained for queries having 2, 3 and 4 PPs. The number of nodes and paths in the index graph are fixed to 30 and 15 respectively.

For a growing number of nodes per PP in the query, the execution time of AC check is almost unaffected. However, it slightly increases as the number of PPs goes up, because of the raise in the number of matchings between the PPs of the involved queries that need to be examined.

The execution time of RC check also goes up as the number of PPs in the queries increases. The reason is that a larger number of complete TPQs are generated and, then, examined in the containment check. On the other hand, the execution time of RC check decreases as the number of nodes per PP goes up, since more restricted queries result in a smaller number of complete TPQs to be examined in the containment check.

Again, our heuristic techniques clearly improve RC check, with RCH_1 being the fastest among all. For a growing number of nodes per PP in the query, the execution time of the precomputed heuristic containment checks RCH_1, RCH_2 and RCH_3 is only slightly affected. On the contrary, the larger is the number of PP nodes, the less is the time spent for $RCFH$ check. This is due to the decrease in the number of index graph paths examined during the (on-the-fly) precedence relationship extraction.

At the same time, the accuracy of $RCFH$ is close to 100%. The accuracy for the other heuristic containment checking approaches decreases as the number of nodes per PP in the queries increases. However, the accuracy of RCH_3 is almost in all cases above 80% for an execution time which is close to that of the approaches RCH_1 and RCH_2.

Remarks. All of our heuristic techniques clearly improve the time of PTPQ relative query containment check RC. However, these techniques are not complete. Therefore, a trade-off has to be determined between desired accuracy on the one side and time resources on the other side. Our experiments show clearly the benefit of using the $RCFH$ heuristic containment check (on-the-fly) if accuracy is the goal ($RCFH$ check is more than one order of magnitude faster than RC containment check, while scoring an accuracy close to 100%). When efficiency is important, a full spectrum of heuristic

[1] http://monetdb.cwi.nl/xml/

[2] http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html

approaches (including those that involve more rules than RCH_1 - RCH_3) gradually trade accuracy for efficiency.

8. CONCLUSION

In this paper, we considered partial tree-pattern queries (PTPQs). A key feature of this type of queries is that the structure in a tree pattern can be flexibly specified fully, partially, or not at all. To efficiently process PTPQs, we defined index graphs on databases that summarize the structure of data trees. We studied the inference of structural expressions of PTPQs and we presented a set of inference rules. We addressed the satisfiability and containment problems for PTPQs in the absence (absolute PTPQ containment) and in the presence (relative PTPQ containment) of index graphs, and we provided necessary and sufficient conditions for each type of query containment.

To cope with the high complexity of relative containment, we designed a family of (sound) heuristic techniques based on structural expressions extracted from the index graph in advance and on-the-fly. We implemented all our approaches and we performed extensive experimental evaluation to report on their efficiency and accuracy. Our results showed that the heuristic relative containment check provides reasonable execution time, while maintaining high accuracy. Those results make our approache appropriate for integration into a query processor for PTPQs.

Currently, our evaluation method of PTPQs is based on the generation of complete TPQs using an index graph. Our future work comprises studying ad-hoc techniques for the optimization of this type of queries.

9. REFERENCES

[1] XML Path Language (XPath). World Wide Web Consortium site, W3C XPath: http://www.w3.org/TR/xpath20.

[2] XML Query (XQuery). World Wide Web Consortium site, W3C XQuery: http://www.w3.org/XML/Query.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 497–508, 2001, Santa Barbara, Cal, USA.

[4] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the 8th Intl. Conf. on Extending Database Technology, Prague, Czech Republic*, 2002.

[5] A. Barta, M. P. Consens, and A. O. Mendelzon. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 133–144, 2005.

[6] L. Chen and E. A. Rundensteiner. Xquery Containment in Presence of Variable Binding Dependencies. In *Proc. of the 14th Intl. Conf. on World Wide Web*, pages 288–297, 2005.

[7] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale xml repository. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, 2001.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A Semantic Search Engine for XML. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, 2003.

[9] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath. In *Proc. of the 8th Intl. Workshop on Knowledge Representation meets Databases*, 2001.

[10] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of Nested XML Queries. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 132–143, 2004.

[11] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the 23rd Intl. Conf. on Very large Databases*, pages 436–445, 1997.

[12] J. Hidders. Satisfiability of XPath Expressions. In *Proc. of the 9th Intl. Workshop on Database Programming Languages*, pages 21–36, 2003.

[13] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *Proc. of the 19th Intl. Conf. on Data Engineering*, pages 367–378, 2003.

[14] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Madison, USA*, pages 133–144, 2002.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured data. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 129–140, 2002.

[16] L. V. S. Lakshmanan, G. Ramesh, H. W. Wang, and Z. J. Zhao. On Testing Satisfiability of Tree Pattern Queries. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 120–130, 2004.

[17] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free Xquery. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 72–83, 2004.

[18] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Proc. of the 21st ACM Symp. on Principles of Database Systems*, pages 65–76, 2002.

[19] T. Milo and D. Suciu. Index structures for Path Expressions. In *Proc. of the 9th Intl. Conf. on Database Theory*, pages 277–295, 1999.

[20] F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proc. of the 13th Intl. Conf. on Database Theory*, pages 315–329, 2003.

[21] N. Polyzotis and M. Garofalakis. Statistical Synopsis for Graph-structured XML Databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[22] P. Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Madison, USA*, pages 299–309, 2002.

[23] D. Theodoratos, T. Dalamagas, A. Koufopoulos, and N. Gehani. Semantic Querying of Tree-Structured Data Sources Using Partially Specified Tree-Patterns. In *Proc. of the 14th ACM Intl. Conf. on Information and Knowledge Management*, pages 712–719, 2005.

[24] D. Theodoratos, T. Dalamagas, P. Placek, and S. Souldatos. Containment of Partially Specified Tree-Pattern Queries. In *Proc. of the Intl. Conference on Scientific and Statistical Databases*, 2006.

[25] P. T. Wood. Containment for XPath Fragments under DTD Constraints. In *Proc. of the 13th Intl. Conf. on Database Theory*, pages 300–314, 2003.

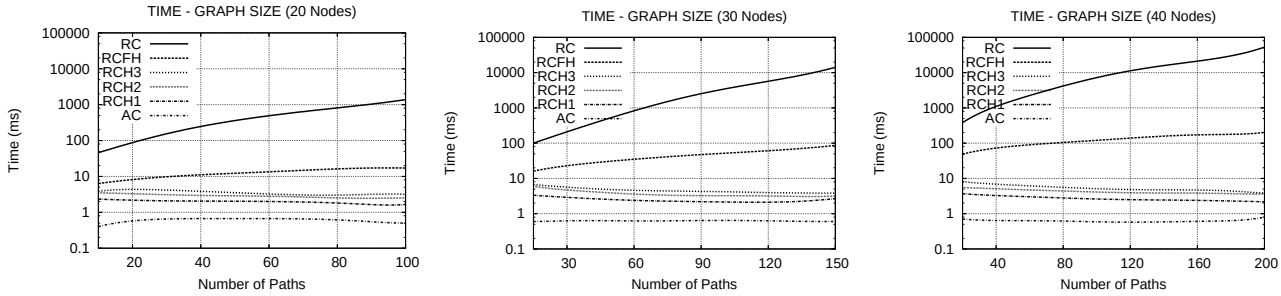

Figure 16: Execution time for checking query containment varying the number of root-to-leaf paths for different numbers of nodes in the index graph.

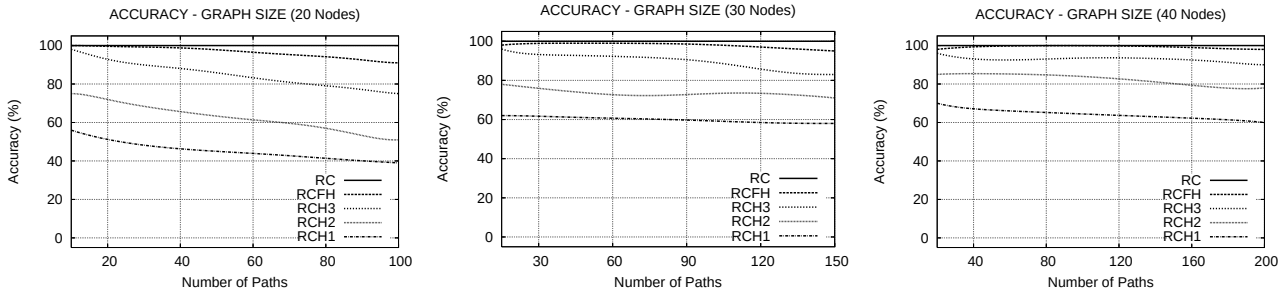

Figure 17: Percentage of correct answers in checking relative query containment varying the number of root-to-leaf paths for different numbers of nodes in the index graph.

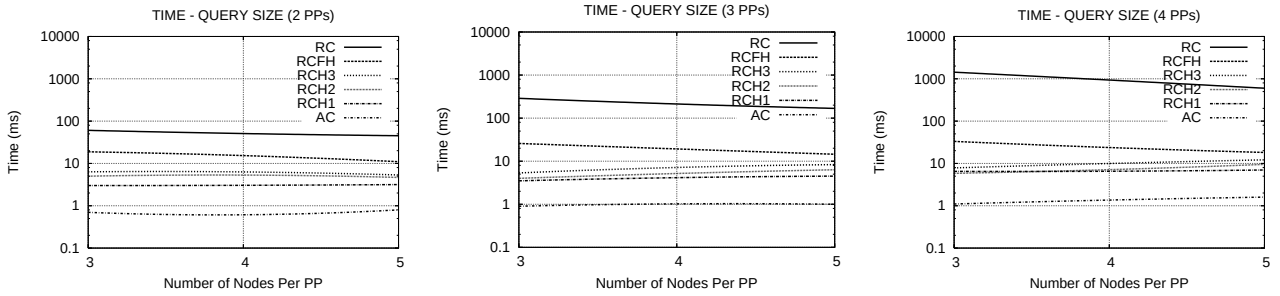

Figure 18: Execution time for checking query containment varying the number of nodes per PP for different numbers of PPs in the query.

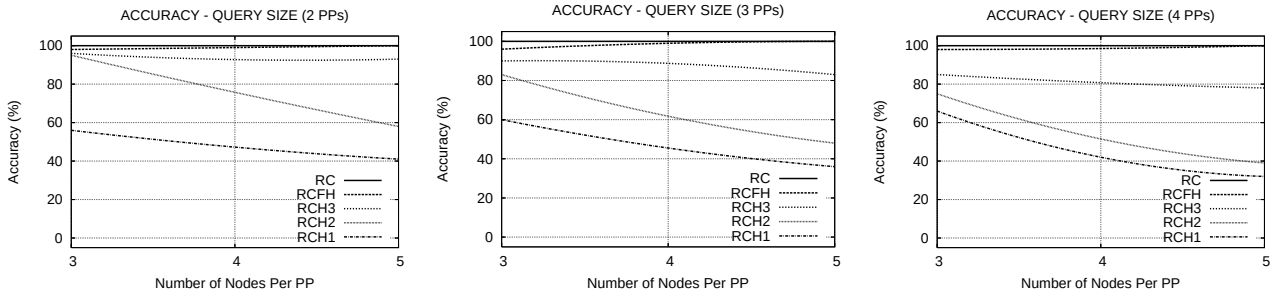

Figure 19: Percentage of correct answers in checking relative query containment varying the number of nodes per PP for different numbers of PPs in the query.