Adding Logical Operators to Tree Pattern Queries on Graph-Structured Data

Qiang Zeng Xiaorui Jiang Hai Zhuge
Institute of Computing Technology, Chinese Academy of Sciences
Beijing 100190, China
{zengqiang, xiaoruijiang}@kg.ict.ac.cn zhuge@ict.ac.cn

ABSTRACT

As data are increasingly modeled as graphs for expressing complex relationships, the tree pattern query on graph-structured data becomes an important type of queries in real-world applications. Most practical query languages, such as XQuery and SPARQL, support logical expressions using logical-AND/OR/NOT operators to define structural constraints of tree patterns. In this paper, (1) we propose generalized tree pattern queries (GTPQs) over graph-structured data, which fully support propositional logic of structural constraints. (2) We make a thorough study of fundamental problems including satisfiability, containment and minimization, and analyze the computational complexity and the decision procedures of these problems. (3) We propose a compact graph representation of intermediate results and a pruning approach to reduce the size of intermediate results and the number of join operations – two factors that often impair the efficiency of traditional algorithms for evaluating tree pattern queries. (4) We present an efficient algorithm for evaluating GTPQs using 3-hop as the underlying reachability index. (5) Experiments on both real-life and synthetic data sets demonstrate the effectiveness and efficiency of our algorithm, from several times to orders of magnitude faster than state-of-the-art algorithms in terms of evaluation time, even for traditional tree pattern queries with only conjunctive operations.

1. INTRODUCTION

Graphs are among the most ubiquitous data models for many areas, such as social networks, semantic web and biological networks. As the most common tool for data transmissions, XML documents are desirably modeled as graphs rather than trees to represent flexible data structures by incorporating the concept of ID/IDREFs. Semantic Web data are also modeled as graphs, e.g. in RDF/RDFS. On graph data, tree pattern queries (TPQs) are one of important queries of practical interest. In query languages such as XQuery and SPARQL, many queries can be regarded as TPQs over graphs. As most of them support logical operations including conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$) in the query conditions, it is necessary to study TPQs over graphs with multiple logical predicates, as illustrated in the following example.

Example 1. A DBLP XML document separately stores inproceeding records for papers and proceeding records for volumes, linked by crossref elements indicating where a paper is published [24]. The underlying data structure is clearly a graph. Consider the following three queries which ask for information of publications for which a certain tree pattern of data holds.

Q_1: Retrieve the information about Alice's conference papers that are published from 2000 to 2010 and co-authored with Bob.

Q_2: Retrieve the information about the conference papers of either Alice or Bob published from 2000 to 2010.

Q_3: Retrieve the information about Alice's conference papers that are not co-authored with Bob and published from 2000 to 2010.

They can be expressed in XQuery and are essentially TPQs on graph-structured data (see [1] for the corresponding XQuery expressions), but Q_2 and Q_3 cannot be expressed in traditional TPQs, which only contain conjunctive predicates. Indeed, they share the same tree representation as depicted in Fig. 1, but different structural predicates should be imposed on the inproceedings element u_1. For example, in Q_1, each embedding of the pattern should satisfy all paths specified in the query; but for Q_2, the two path conditions "u_1–u_2" and "u_1–u_3" are not required to be satisfied simultaneously. A predicate that specifies those edge constraints and incorporates disjunction and negation needs to be attached to each query node in order to express Q_2 and Q_3. In general, (1) it is common in practice that logical expressions on query nodes needs to be imposed to specify complex relationships for not only attribute predicates (e.g. $2000 \leq \text{year} \leq 2010$) but also structural constraints $\big($e.g. (u_1–u_2 or u_1–u_3) in Q_2 and $not(u_1$–$u_3)$ in $Q_3\big)$; (2) some of the nodes $\big($e.g. $u_i(i \in \{1,2,3,6,8\})\big)$ in the query pattern only serve as filters for pruning unexpected results, which means that the results of a TPQ should consist of matches for a portion of the query nodes only. □

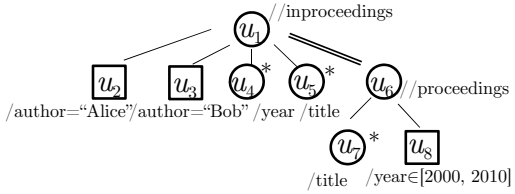


Figure 1: The tree representation of Q_1, Q_2, and Q_3 in Example 1. Document elements matching the starred query nodes are required to be returned and the single-/double-lined edges denote the parent-child/ancestor-descendant relationships between elements.

Although TPQs have been widely studied for many years, few of the proposed processing algorithms can be used to efficiently evaluate such queries over general graphs. They can neither support disjunction and negation on structural constraints nor be optimized for the situation where output nodes take only a portion of query nodes (see Related work for details).

Proceedings of the VLDB Endowment, Vol. 5, No. 8

Contributions & Roadmap. This work makes the first effort to deal with TPQ over general graph-structured data with Boolean logic support. The contributions are summarized as follows.
(1) We introduce a new class of tree pattern queries over graph-structured data, called generalized tree pattern queries (GTPQs) (Section 2). In a GTPQ, a node is not only associated with an attribute predicate, which specifies the property conditions, but also a structural predicate in terms of propositional logic with logic connectives including conjunction, negation and disjunction to specify structural conditions with respect to its descendants. The query allows a portion of the query nodes to be output nodes. We also show that our formalization of query is advantageous over those in the literature on queries against tree-structured data.
(2) We investigate fundamental problems for GTPQs, including satisfiability, containment, equivalence and minimization (Section 3). We show that the satisfiability of a special GTPQ with only conjunction and disjunction is solvable in linear time, but the satisfiability and the other three problems become computationally intractable when negation is incorporated. We propose an exact algorithm to minimize GTPQs, which is supposed to be sufficiently efficient, since the query sizes are typically small in practice.
(3) We propose a graph representation of intermediate results and a pruning approach to address notable problems in evaluating query patterns over graphs, develop an algorithm for GTPQs with ancestor-descendant edges and its extension to deal with parent-child edges (Section 4). The algorithm can largely filter nodes that cannot contribute to the final results, wisely avoid generating redundant intermediate results, and compactly represent the matches.
(4) We implement our algorithm and conduct an experimental study using synthetic and real-life data (Section 5). We find that our evaluation algorithm performs significantly better than state-of-art algorithms even for conjunctive TPQs. It also has better scalability and is robust for different queries on different graphs. The experiments also demonstrate the effectiveness of the graph representation of results and the efficiency of the pruning method.

Related work. There is a large body of research work on TPQs over tree-structured data (see [14] for a survey). However, all studies heavily relied on the relatively simple structure of trees and employed the node encoding schemes (including the interval [4], Dewey [21] and sequence [28] encodings) that are not applicable to graphs for determining structural relationships. Techniques critical for their efficiency, such as stack encoding and nodes skipping, can be only applied to tree-structured data. For some sparse graph data whose structures can be modeled by disjoint trees connected by edges, such as many XML documents with ID/IDREFs, although one can apply those existing algorithms for tree-structured data to evaluate a query over such graphs by first decomposing it to several TPQs over different trees and then merging the results of distinct queries to form the final results, it is inefficient due to large redundant intermediate results and costly merging processes.

Some studies extended the traditional TPQs by incorporating additional functions and restrictions. Chen et al. [10] included optional nodes to patterns and investigated efficient evaluation plans upon native XML database systems. The generalized tree pattern is still against tree-structured data, which differs from this work that studies TPQs over graph-structured data with logical predicates. Jiang et al. [16] proposed new holistic algorithms based on a concept of OR-blocks to process AND/OR-twigs, TPQs with OR-predicates. In the end of Section 2, we shall show that (1) our query size can be always no larger than the size of element nodes of AND/OR-twig for expressing a semantically identical query; (2) constructing OR-blocks involves converting a propositional formula to conjunctive normal form, thus taking exponential time in the worst case; (3) the proposed algorithms only support tree-structured data as input. [17] studied path queries with negation, while [29] and [20] added negation to TPQs. They cannot be applied to GTPQs either, since they are based on the classical holistic twig join algorithm [4] that only works on tree-structured data.

There has been work on pattern queries for graph-structured data. TwigStackD [7] generalized the holistic algorithms, but it takes considerable time and space without a pre-filtering process [30]. HGJoin [27] can evaluate general graph pattern queries using OPT-tree-cover [2] as the underlying reachability indexing approach. It decomposes a pattern into a set of complete bipartite graphs and generates matches for them in order according to a plan. The time cost of plan generation is always exponential since it has to produce a state graph with exponential nodes no matter for obtaining an optimal or suboptimal plan. Cheng et al. [11] proposed *R*-join/*R*-semijoin processing for the graph pattern matching problem. It relies on a cluster-based *R*-join index whose size is typically prohibitively large, as the index stores matches for every two labels derived from 2-hop indexing [12]. Unlike the plan generation of HGJoin, it adopts left-join to reduce the cost, but in the worst case the time complexity is still exponential. Since both HGJoin and *R*-join/*R*-semijoin use structural joins similar to the earlier work on tree-structured data, they typically have large intermediate results and need to perform large amounts of expensive join operations. All these three algorithms also do not directly support queries with negative/disjunctive predicates. A straightforward approach to apply them to the GTPQ processing is to decompose the query into multiple conjunctive TPQs and perform the difference and merge operations on results of the decomposed queries. However, the number of the resultant conjunctive TPQs may be exponential and large intermediate results may need to be generated and merged.

A number of studies investigated various graph pattern matching problems [13, 15, 31]. [15] proposed a graph query language GraphQL and studied graph-specific optimization techniques for graph pattern matching that combines subgraph isomorphism and predicate evaluation. While the language is able to express queries with ancestor-descendant edges and disjunctive predicates, the work focused on processing non-recursive and conjunctive graph pattern queries, where all edges of a query pattern correspond to the parent-child edges of GTPQs, specifying the adjacent relationship between desired matching nodes. [13] defined matching in terms of bounded simulation to reduce its computation complexity. [31] studied distance pattern matching, in which query edges are mapped to paths with a bounded length. Queries of [13] and [31] do not support negative/disjunctive predicates on edges and have quite different semantics with ours.

Most existing algorithms are to find all instances of patterns containing matches of all query nodes. In real-world applications, however, the answer to the query often only require matches of several but not all query nodes. Indeed, many query nodes only serve as filters for imposing structural constraints on output nodes. Our framework can avoid generating redundant matches at run time.

Satisfiability, containment, equivalence and minimization are fundamental problems for any query languages. The minimization of TPQs over tree-structured data has been investigated in several papers. Amer-Yahia et al. [3] proposed algorithms for the minimization with and without integrity constraints. Ramanan [23] studied this problem for TPQs defined by graph simulation. Chen et al. [6] used a richer class of integrity constraints for query minimization of TPQs with an unique output node. However, we are not aware of previous work on minimization as well as the other three problems for TPQs with logical predicates either over tree-structured data or over graph-structured data.

2. DATA MODEL AND GENERALIZED TREE PATTERN QUERIES

Data graphs. A data graph is a directed graph $G = (V, E, f)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is finite set of edges, in which each pair (v, v') denotes an edge from v to v'; (3) f is a function on V defining attribute values associated with nodes. For each node $v \in V$, $f(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$, where the expression $A_i = a_i (i \in [1, n])$ represents that v has a attribute denoted by A_i and its value is a constant a_i. For example, in a data graph $G = (V, E, f)$ of a DBLP document, the node properties in f may include tags, string values, typed values, and attributes specified in the elements.

Abusing notions for trees and traditional tree pattern queries, we refer to a node v_2 as a *child* of a node v_1 (or v_1 as a parent of v_2) and say they have a *parent-child* (PC) relationship if there is an edge (v_1, v_2) in E, and refer to v_2 as a descendant of v_1 (or v_1 as an *ancestor* of v_2) and say they have an *ancestor-descendant* (AD) relationship if there is a nonempty path from v_1 to v_2 in G.

Generalized tree pattern queries. A generalized tree pattern query (GTPQ) $Q = (V_b, V_p, V_o, E_q, f_a, f_e, f_s)$, where:
(1) V_b and V_p are both a finite set of nodes, called *backbone nodes* and *predicate nodes*, respectively. The complete set of query nodes is denoted as V_q, i.e., $V_q = V_b \cup V_p$.
(2) $V_o \subseteq V_b$. The nodes in V_o are called *output nodes*.
(3) $E_q \subseteq \{(u_1, u_2) | u_1, u_2 \in V_b\} \cup \{(u_1, u_2) | u_1 \in V_b \cup V_p, u_2 \in V_p\}$, is a finite set of edges. Here, (V_q, E_q) is restricted to a directed tree .
(4) f_a is a function defined on V_q such that for each node $u \in V_q$, $f_a(u)$ is an *attribute predicate* that is a conjunction of atomic formulas of the form of "A op a", in which A is an attribute name, a is a constant and op is a comparison operator in $\{<, \leq, =, \neq, >, \geq\}$.
(5) f_e is a function on E_q to specify the type of the edge. Each edge (u_1, u_2) represents either PC relationship or AD relationship.
(6) f_s is a function defined on internal nodes. For each internal node $u \in V_q$ with k children being predicate nodes, $f_s(u)$, called a *structural predicate*, is a propositional formula in k variables $p_{u'_1}, \ldots, p_{u'_k}$, each corresponding to a tree edge directing to a predicate child of u. In particular, if u has no predicate children, $f_s(u) = 1$. Each node u is associated with a distinct propositional variable denoted by p_u.

We call a GTPQ a union-conjunctive GTPQ if the structural predicates on all query nodes are negation-free, and call it a conjunctive GTPQ if the structural predicates on all the query nodes only have conjunction connectives.

Before giving the semantics of GTPQs, we add variables for non-root backbone nodes to extend the structural predicate. For an internal node u with k' backbone children, denoted by $u_1, \ldots, u_{k'}$, the *extended structural predicate* $f_{ext}(u) = p_{u_1} \wedge \ldots \wedge p_{u_{k'}} \wedge f_s(u)$.

Example 2. In Example 1, $Q_1 = (V_b, V_p, V_o, E_q, f_s, f_e, f_s)$ is a conjunctive GTPQ, in which (1) $V_b = \{u_1, u_4, u_5, u_6, u_7\}$, $V_p = \{u_2, u_3, u_8\}$, $V_o = \{u_4, u_5, u_7\}$; (2) the attribute predicate f_a for a query node is a conjunction of comparisons among tags and typed values $\big($e.g. $f_a(u_2) = (\text{tag} = \text{"author"} \wedge \text{value} = \text{"Bob"})\big)$; (3) $f_s(u_1) = p_{u_2} \wedge p_{u_3}$, and $f_s(u_6) = p_{u_8}$. The only difference between Q_2 and Q_1 is that in Q_2, $f_s(u_1) = p_{u_2} \vee p_{u_3}$. In Q_3, $f_s(u_1) = p_{u_2} \wedge \neg p_{u_3}$. As an example of extended structural predicates, for Q_2, $f_{ext}(u_1) = (p_{u_2} \vee p_{u_3}) \wedge p_{u_4} \wedge p_{u_5} \wedge p_{u_6}$. □

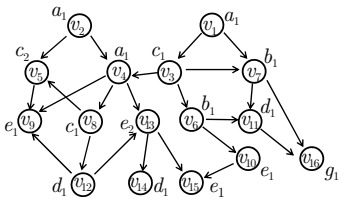
Semantics. Consider a data graph $G = (V, E, f)$ and a GTPQ $Q = (V_b, V_p, V_o, E_q, f_a, f_e, f_s)$. We say that a data node v in G *downwardly matches* a query node u in Q, denoted by $v \models u$, if the following conditions are satisfied:
(1) v satisfies the attribute predicate of u, denoted by $v \sim u$. That is, for each formula "A op a" in $f_a(u)$, there is an element $(A = a')$ in $f(v)$ such that a' op a. v is called a *candidate matching node* of u. $mat(u)$ denotes the set of candidate matching nodes of u, i.e., $mat(u) = \{v | v \in V, v \sim u\}$.
(2) If u is an internal node, the data node v determines a truth assignment to the variables of $f_{ext}(u)$ such that $f^v_{ext}(u) = 1$, where $f^v_{ext}(u)$ denotes the truth-value of f_{ext} under the assignment. For each variable $p_{u'}$, the truth-value $p^v_{u'}$ is assigned as follows: for each PC (resp. AD) child u' of u, $p^v_{u'} = 1$ if there exists a child (resp. descendant) v' of v such that $v' \models u'$; otherwise, $p^v_{u'} = 0$.

Let $V_b = \{u_1, \ldots, u_m\}$. A m-ary tuple $(v_1, \ldots, v_m)$ of nodes in G is said to be a match of Q on G, if the following conditions hold: (1) for each $v_i (i \in [1, m])$, $v_i \models u_i$; (2) for each edge $(u_i, u_j) \in E_q (i, j \in [1, m])$, if u_j is a PC child of u_i, v_j is a child of v_i; otherwise, v_j is a descendant of v_i.

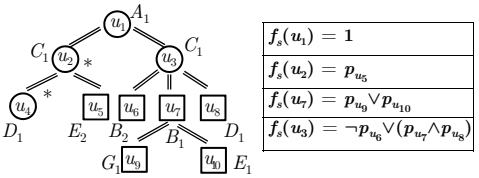
The answer $Q(G)$ to Q is a set of results in the form of tuples, where each tuple consists of the images of output nodes V_o in a match of Q. For each match, there is at least an assignment for all variables that makes the extended structural predicates of all internal backbone nodes and some of internal predicate nodes evaluate to true, which we call a certificate of the match. For a match and an assignment as a certificate of the match, an instance of Q on G is a tuple consisting of such nodes that each of them matches a distinct query node whose corresponding propositional variable is true under the assignment. In particular, an instance of conjunctive GTPQ is exactly a match of the query.

Example 3. For simplicity of presentation, a lower-case letter x_i in all figures throughout this paper denotes $f(v)$ for a data node v and a capital letter Y_j denotes $f_a(u)$ for a query node u such that $v \sim u$ if $j \leq i$ and $X = Y$.

Consider the data graph and the query shown in Fig. 2. $v_{13} \sim u_5, v_{15} \not\sim u_5$. Accordingly, $mat(u_5) = \{v_{13}\}, mat(u_{10}) =$

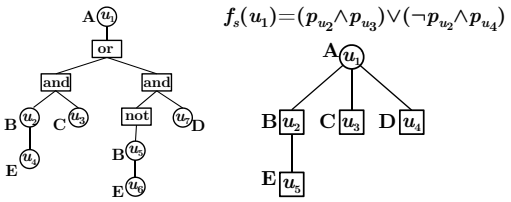


(a) Data graph G



(b) GTPQ Q on G

Figure 2: Example of a data graph and a GTPQ. We use a rectangle to represent a predicate node and a circle to represent a backbone node.



(a) B-twig query (b) GTPQ

Figure 3: Comparison between a B-twig query and a GTPQ

$\{v_9, v_{10}, v_{13}, v_{15}\}$. The answer $Q(G) = \{(v_3, v_{11}), (v_3, v_{12}), (v_3, v_{14}), (v_8, v_{12}), (v_8, v_{14})\}$. One of the query matches leading to (v_3, v_{11}) is (v_1, v_3, v_3, v_{11}), where elements are sorted in the ascending order of the subscripts of corresponding query nodes. An instance of this match is $\{u_1 : v_1, u_2 : v_3, u_3 : v_3, u_4 : v_{11}, u_7 : v_6, u_8 : v_{11}, u_9 : v_{15}\}$, where '$u : v$' means v is a match of u. Indeed, $v_3 \models u_3$, because (1) $v_3 \sim u_3$, and (2) $f_{ext}^{v_3}(u_3) = 1$ since $v_6 \models u_7$ and $v_{11} \models u_8$. Also, $v_5 \models u_3$, because v_5 cannot reach a node matching u_6 and hence $p_{u_3}^{v_5} = 0$, thereby $f_{ext}^{v_5}(u_3) = 1$. □

For simplicity of semantics, we require a query to explicitly specify backbone nodes and predicate nodes and restrict output nodes to backbone ones. The distinction between the two types of nodes is that propositional variables associated with backbone nodes are disallowed to be operands of negation and disjunction as those associated with predicate nodes, which guarantees that each backbone node has an image in a match of the query. Permitting negation and disjunction on any query nodes leads to issues that are not computationally desirable. If each query result is still required to have an image for each output node, the expressive power does not change; but to determine whether a query is valid is effectively to check whether the variables associated with output nodes are always true for all certificates of matches, which is a co-NP-complete problem. Otherwise, the output structures become not fixed. They can either be specifically defined in the query, or consist of exponential combinations of output nodes by default. Our algorithm described in Section 4 can be straightforwardly extended to process queries with multiple output structures [1].

We now compare GTPQ with the works in [29] and [5]. [29] deals with *AND/OR-twig* against tree-structured data. [5] further extends [29] to handle *B-twig*, which additionally introduces the logical-NOT operation into the query. Both represent a query by defining special types of nodes for operators, namely logical-AND nodes, logical-OR nodes and logical-NOT nodes. For each occurrence of a variable in a structural predicate of a GTPQ, the corresponding AND/OR-twig or B-twig needs to use a distinct subtree to express the structural constraints with respect to descendants as specified by the variable, since in AND/OR-twigs and B-twigs, the query nodes connected to different operator nodes are considered as distinct. The query size of AND/OR-twigs or B-twigs hence may be much larger than the size of a GTPQ for expressing complex tree patterns. In Fig. 3, the B-twig query has to use two paths u_2–u_4 and u_5–u_6 to represent the constraints that can be imposed by a single path u_2–u_5 in the semantically equivalent GTPQ. Moreover, before evaluating the query, [29] and [5] have to construct OR-blocks to normalize the twig. The normalization process is essentially a CNF conversion of propositional formulas. Since a CNF conversion can lead to an exponential explosion of the formula, the time cost of a conversion is exponential in the size of original query, and the resulting query size also becomes exponential in the worst case. Therefore, our query representation is more powerful and compact than the tree representation of [29] and [5].

3. FUNDAMENTAL PROBLEMS FOR GENERALIZED TREE PATTERN QUERIES

In this section, we study the problems of satisfiability, containment, equivalence, and minimization of GTPQs, which are important for query analysis and optimization.

3.1 Satisfiability

A GTPQ Q is *satisfiable* if there is a data graph G on which the answer $Q(G)$ to Q is nonempty. We first introduce some definitions before showing how to determine the satisfiability and establishing the property of the problem.

We say u is an *independently constraint* node if (1) the formula $\big(f_s(u')[p_u/1] \oplus f_s(u')[p_u/0]\big) \wedge f_s(u)$ is satisfiable, in which u' is the parent of u, $f_s(u')[p_u/x]$ is the formula produced by assigning x to the variable p_u ($x \in \{0, 1\}$), and $\oplus$ is the exclusive-or logical operator; (2) all ancestors of u are independently constraint nodes. Intuitively, the variables of independently constraint nodes can independently affect the resulting truth-value of the structural predicates of their parents and ancestors. Backbone nodes are clearly independently constraint nodes, if their structural predicates are satisfiable.

A *transitive structural predicate* $f_{tr}(u)$ for a node u is constructed from $f_{ext}(u)$ in a bottom-up sweep as follows. (1) For each leaf node and each non-independently constraint node u , the transitive structural predicate is the same as the extended structural predicate, i.e. $f_{tr}(u) = f_{ext}(u)$. (2) For an internal node u such that the transitive structural predicates of all children have been defined, $f_{tr}(u)$ is produced by substituting $\big(p_{u'} \wedge f_{tr}(u')\big)$ for each variable $p_{u'}$ of independently constraint node u' in $f_s(u)$.

For two non-root nodes u_1, u_2 in Q, we say that u_2 is *similar* to u_1, denoted by $u_1 \lhd u_2$, if the following conditions hold. (1) For each formula "A op a_1" in $f_a(u_1)$, there is a formula "A op a_2" in $f_a(u_2)$ such that (a) if op $\in \{\leq, <\}$, $a_2 \leq a_1$, (b) if op $\in \{\geq, >\}$, $a_2 \geq a_1$, (c) if op $\in \{=, \neq\}$, $a_1 = a_2$. We use $u_2 \vdash u_1$ to denote that u_1 and u_2 satisfy this condition. (2) For each PC (resp. AD) child u_1' of u_1 such that u_1' is an independently constraint node, there is a PC child (resp. a descendant) u_2' of u_2 such that $u_1' \lhd u_2'$. (3) The formula $f_{tr}(u_2) \rightarrow f_{tr}(u_1)[u_1 \mapsto u_2]$ is a tautology, where $f_{tr}(u_1)[u_1 \mapsto u_2]$ is a formula transformed from $f_{tr}(u_1)$ by replacing $p_{u'}$ with $p_{u''}$ for each pair (u', u'') such that (a) u' is a descendant of u_1, (b) u'' is a descendant of u_2 and (c) $u' \trianglelefteq u''$. We say that u_1 is *subsumed* by u_2, denoted by $u_1 \trianglelefteq u_2$, if (1) $u_1 \lhd u_2$, and (2) the parent of u_1 is the lowest common ancestor u_{lca} of u_1 and u_2, and (a) if u_1 is a PC child of u_{lca}, u_2 is also a PC child of u_{lca}; (b) otherwise u_2 is a descendant of u_{lca}.

We finally define *complete structural predicates* to characterize the whole structural constraints of a GTPQ. For a node u, the complete structural predicate $f_{cs}(u)$ is created from the corresponding transitive structural predicate $f_{tr}(u)$ by performing the following operations: (1) for each descendant u' of u, if its attribute predicate is unsatisfiable, $f_{cs}^{new}(u) = f_{cs}^{old}(u)[p_{u'}/0]$, where $f_{cs}^{old}(u)$ is the old formula before this transformation and $f_{cs}^{new}(u)$ is the newly generated formula; (2) for every two nodes u_1 and u_2 in two distinct subtrees of u such that $u_2 \trianglelefteq u_1$, $f_{cs}^{new}(u) = f_{cs}^{old}(u) \wedge \big(\neg p_{u_1} \vee (p_{u_2} \wedge f_{ext}(p_{u_2}))\big)$, where $f_{cs}^{old}(u)$ and $f_{cs}^{new}(u)$ have the same meaning as above in (1).

Theorem 1 shows that the satisfiability of a GTPQ is equivalent to the satisfiability of the complete structural predicate of the root, if given that the attribute predicate of the root is satisfiable. If the query is a conjunctive or union-conjunctive GTPQ, the problem of satisfiability can be solved in linear time. When negation is added into the query, the satisfiability becomes NP-complete.

Theorem 1. *A GTPQ Q is satisfiable if and only if for the root node u of Q, $f_a(u)$ and $f_{cs}(u)$ are both satisfiable.* □

Theorem 2.

1. *The satisfiability of a union-conjunctive GTPQ can be determined in linear time.*
2. *The satisfiability of a GTPQ is NP-complete.* □

Example 4. Consider the query in Fig. 2(b). All query nodes are independently constraint nodes. Replacing p_{u_7} with $p_{u_7} \wedge (p_{u_9} \vee p_{u_{10}})$ in $f_{ext}(u_3)$, we have $f_{tr}(u_3) = \neg p_{u_6} \vee (p_{u_7} \wedge (p_{u_9} \vee p_{u_{10}}) \wedge p_{u_8})$. Since there are no two nodes u and u' such that $u \trianglelefteq u'$, $f_{cs}(u_1) = f_{tr}(u_1) = p_{u_5} \wedge p_{u_4} \wedge p_{p_5} \wedge p_{u_3} \wedge \big(\neg p_{u_6} \vee$

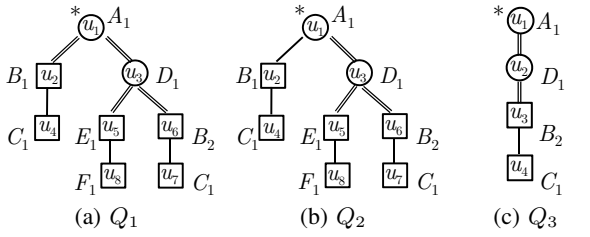


Figure 4: Examples for four fundamental problems of GTPQs

$(p_{u_7} \wedge (p_{u_9} \vee p_{u_{10}}) \wedge p_{u_8}))$. Due to the satisfiability of $f_{cs}(u_1)$, we see that the query is satisfiable. Indeed, we can get a nonempty answer by posing Q on G in Fig. 2(b) as shown in Example 3.

Let us turn to Q_1 and Q_2 depicted in Fig. 4. The following table presents structural predicates of internal nodes for Q_1 and Q_2.

$f_s(u_1) = \neg p_{u_2}$	$f_s(u_2) = p_{u_4}$	$f_s(u_5) = p_{u_8}$
$f_s(u_3) = (p_{u_5} \wedge p_{u_6}) \vee (\neg p_{u_5} \wedge p_{u_6})$		$f_s(u_6) = p_{u_7}$

For both queries, u_5 and u_8 are two non-independently constraint nodes. In Q_1, we have $u_2 \trianglelefteq u_6$, because (1) $u_6 \vdash u_2$, (2) $u_4 \trianglelefteq u_7$, (3) $f_{tr}(u_6) \rightarrow f_{tr}(u_2)[u_2 \mapsto u_6] = p_{u_7} \rightarrow p_{u_7}$, which is a tautology, (4) u_2 is an AD child of u_1 which is an ancestor of u_6. In contrast, for Q_2, $u_2 \not\trianglelefteq u_6$, since now u_2 is a PC child of u_1 but u_6 is not. Suppose attribute predicates of all nodes are satisfiable. Then for Q_2, $f^2_{cs}(u_1) = \neg(p_{u_2} \wedge p_{u_4}) \wedge p_{u_3} \wedge \big((p_{u_5} \wedge p_{u_6} \wedge p_{u_7}) \vee (\neg p_{u_5} \wedge p_{u_6} \wedge p_{u_7})\big)$, which is satisfiable; but for Q_1, $f^1_{cs}(u_1) = f^2_{cs}(u_1) \wedge \big(p_{u_6} \rightarrow (p_{u_2} \wedge p_{u_4})\big)$, which is unsatisfiable. Therefore, we know that Q_2 is satisfiable and Q_1 not. □

3.2 Containment and Equivalence

For two GTPQs Q_1 and Q_2, Q_1 is *contained* in Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for any data graph G, $Q_1(G) \subseteq Q_2(G)$. Q_1 and Q_2 is *equivalent*, denoted by $Q_1 \equiv Q_2$, if $Q_1(G) \subseteq Q_2(G)$ and $Q_2(G) \subseteq Q_1(G)$.

Homomorphism. Given two GTPQs Q_1 with query nodes V^1_q and Q_2 with query nodes V^2_q, a homomorphism from Q_1 to Q_2 is a mapping λ from V^1_q to $V^2_q \cup \{\perp\}$ such that (1) the two sets of output nodes of Q_1 and Q_2 are bijective; (2) for any non-independently constraint node $u \in V^1_q$, $\lambda(u) = \perp$; (3) for any independently constraint node u_1 in V^1_q, (a) for any PC (resp, AD) child node u'_1 of u_1 such that u'_1 is also an independently constraint node, $\lambda(u'_1)$ is a PC child (resp, a descendant) of $\lambda(u_1)$, and (b) $\lambda(u_1) \vdash u_1$; (4) the formula $f_{cs}(u^2_{root}) \rightarrow f_{cs}(u^1_{root})[u^1_{root} \mapsto \lambda(u^1_{root})]$ is a tautology, where u^1_{root} is the root node of Q_1 and $f_{cs}(u^1_{root})[u^1_{root} \mapsto \lambda(u^1_{root})]$ is a formula transformed from $f_{cs}(u^1_{root})$ by replacing $p_{u'}$ with $p_{\lambda(u')}$ for each independently constraint node $u' \in V^1_q$.

Theorem 3 yields a decision procedure for containment and equivalence between two GTPQs. Theorem 4 states the intractability of the two problems of containment and equivalence.

Theorem 3. *For two GTPQs Q_1 and Q_2, $Q_1 \sqsubseteq Q_2$ iff there exists a homomorphism from Q_2 to Q_1.* □

Theorem 4. *Containment of GTPQs is co-NP-hard.* □

Example 5. Recall the queries in Fig. 4. We now assume $f_s(u_1) = p_{u_2}$ and others the same as in Example 4. Let Q_3 be a conjunctive GTPQ, and u^j_i denote u_i in Q_j to distinguish nodes in different queries. We have that $Q_2 \sqsubseteq Q_3$, $Q_2 \sqsubseteq Q_1$ and $Q_1 \equiv Q_3$. Indeed, there is a homomorphism $\lambda_{3,2}$ from Q_3 to Q_2, where $\lambda_{3,2}(u^3_1) = u^2_1, \lambda_{3,2}(u^3_2) = u^2_3, \lambda_{3,2}(u^3_3) = u^2_6, \lambda_{3,2}(u^3_4) = u^2_7$. There is also $\lambda_{1,3}$ from Q_1 to Q_3, in which $\lambda_{1,3}(u^1_i) = \perp$ $(i = 5, 8), \lambda_{1,3}(u^1_j) = u^3_3 (j = 2, 6), \lambda_{1,3}(u^1_k) = u^3_4 (k = 4, 7), \lambda_{1,3}(u^1_1) = u^3_1, \lambda_{1,3}(u^1_3) = u^3_2$. We can also derive $\lambda_{3,1}$ and $\lambda_{1,2}$. □

Algorithm 1: minGTPQ

Input: GTPQ $Q = (V_b, V_p, V_o, E_q, f_a, f_e, f_s)$ with the root u_r.
Output: A minimum equivalent GTPQ Q_m of Q.

1. construct an equivalent query Q_m from Q by removing subtrees rooted at a node whose attribute predicates are unsatisfiable and assigning the variables of the removed nodes to 0 for respective structural predicates
2. check each structural predicate to determine for each node whether it is an independently constraint node and remove all non-independently constraint nodes followed by assigning the variables of them to 0 for respective structural predicates
3. compute the complete structural predicate $f_{cs}(u)$ for each node u in Q_m in bottom-up order
4. **for each** $u \in V^m_q$ in bottom-up order do **do**
5. **if** $f_{cs}(u)$ is unsatisfiable **then**
6. $f_s\big(parent(u)\big) := f_s\big(parent(u)\big)[p_u/0]$
7. remove the whole subtree rooted at u from Q_m
8. **for each** node $u \in V^m_q$ **do**
9. **if** the formula $f_{cs}(u_r) \rightarrow p_u$ is a tautology **then**
10. **for each** u' such that $u' \trianglelefteq u$ **do**
11. $f_s\big(parent(u')\big) := f_s\big(parent(u')\big)[p_{u'}/1]$
12. **for each** output node u_o in the subtree rooted at u' **do**
13. **if** there exists u'' such that $u_o \triangleleft u''$ and the subtree query pattern rooted at u'' and that rooted at u_o are isomorphic **then**
14. remove u_o from the set of output nodes and add u'' into it
15. remove nodes in the subtree rooted at u' from Q_m that are not ancestors of any output nodes and corresponding edges they connect
16. **else if** the formula $f_{cs}(u_r) \rightarrow \neg p_u$ is a tautology **then**
17. **for each** pair $(u, u') \in S$ **do**
18. $f_s\big(parent(u')\big) := f_s\big(parent(u')\big)[p_{u'}/0]$
19. remove the whole subtree rooted at u' from Q_m
20. **return** Q_m

3.3 Minimization

Since the efficiency of processing a query depends on the size of it, it is necessary to identify and eliminate redundant nodes. For a GTPQ with query nodes V_q, we define its size as $|Q| = |V_q|$.

Minimization. Given a GTPQ Q, the minimization problem is to find another GTPQ Q_m such that (1) $Q \equiv Q_m$, (2) $|Q_m| \leq |Q|$, and (3) there exists no other such Q' with $|Q'| < |Q_m|$.

From Theorem 3, we have that for a GTPQ Q, there is a minimal equivalent GTPQ of Q whose query nodes are a subset of query nodes of Q. We say two GTPQs Q_1 and Q_2 are isomorphic, if there is a homomorphism between them that is a one-to-one mapping. The following proposition shows that the minimal equivalent query of a GTPQ is unique up to isomorphism.

Proposition 5. *Let GTPQs Q_1 and Q_2 be minimal and equivalent. Then Q_1 and Q_2 are isomorphic.* □

Algorithm 1 shows how to minimize a GTPQ. Due to space limit, we omit the description and instead give an example to illustrate it.

Example 6. In Fig. 4, the query Q_3 is a minimum equivalent query of Q_1 with structural predicates given in Example 5. (1) Since we suppose all attribute predicates are satisfiable, there are no nodes to be removed in this step, and $Q_m = Q_1$ (line 1). (2) All nodes except u_5 and u_8 are independently constraint nodes, hence we remove u_5 and u_8 and assign 0 to p_{u_5} in $f_s(u_3)$, thereby having that $f_s(u_3) = p_{u_6}$ (line 2). In this step, all propositional formulas of structural predicates are *simplified* to equivalent formulas with minimum variables. (3) There are no nodes whose complete structural predicates are unsatisfiable, and so none is removed (line 4–7). (4)

The formula $f_{cs}(u_1) \rightarrow p_{u_6}$ is a tautology and $u_2 \trianglelefteq u_6$, so u_2 and its child u_4 is removed, and we have $f_s(u_1) = 1$, thereby generating the query Q_3 (line 8–19). This step is to remove subtrees which can be *semantically subsumed* by others. □

The correctness can be proved based on Theorem 3. Since the algorithm involves solving SAT problems, the worst-case time complexity is exponential in the query size. In fact, Theorem 6 shows that the minimization problem is NP-hard and hence it is difficult to find a polynomial-time algorithm. Nevertheless, because there are many high-performance algorithms for SAT and the query size is not much large in practice, it is still worth minimizing a GTPQ considering the benefits of efficiency of evaluation.

Theorem 6. *The minimization problem for GTPQs is NP-hard.* □

4. EVALUATING GENERALIZED TREE PATTERN QUERIES

4.1 Framework

Recall that two major problems that impair the efficiency of algorithms for processing TPQs over graphs are large intermediate results and expensive join operations on them. In the following, we propose two new techniques to address them.

Graph representation of intermediate results. To reduce the cost of storing intermediate results and avoid merge-join operations, we represent intermediate results as a graph rather than sets of tuples. Each match for a path or a substructure of the query pattern can be embedded into the tree pattern and hence naturally can be represented as a tree. By grouping all the candidate matches by the corresponding matched query nodes and adding an edge to connect a pair of data nodes whenever there's an edge between the corresponding pair of query nodes in the query pattern, we can represent the intermediate and final results as graphs. In such a graph representation, each data node exists at most once, in contrast to the tuple representation in which a data node may be in multiple tuples. Also, the AD or PC relationship between two nodes is exactly represented by only one edge, while in the tuple form the corresponding two nodes may be put as an element in more than one tuple to repeatedly and explicitly represent their relationship. Since the size of the intermediate matches may be huge, even exponential in both the query size and the data size in the worst case, the graph representation is much more compact with at most quadratic space cost. Moreover, to enumerate all resulting matches of a pattern query, we only need to perform one single graph traversal on a presumably small graph instead of multiple merge-join operations over large intermediate results.

It is worth noting that such a way of representing intermediate results can be also applied to algorithms for other graph pattern queries to boost their evaluation. For TPQs, it is particularly optimal because we can enumerate matches directly from the graph. However, for graph pattern queries, additional matching operations including joins may be unavoidable because it is difficult to locally determine which nodes should be traversed to form a match. The additional matching operations are in essence an easier evaluation of a pattern matching on a smaller graph, such a technique can thus still be expected to speed up the whole processing.

Reachability index enhanced effective pruning. Since the number of data nodes to be processed significantly affects the efficiency of pattern query evaluation, it is desirable to perform effective pruning to reduce the number of candidate matching nodes. In the literature, [7] and [11] have developed two pruning approaches for reachability query pattern matching. TwigStackD [7] proposed a pre-filtering approach that can select nodes guaranteed to be in final matches. Since it has to perform two graph traversals on the data graph, it is likely unfeasible for large-scale real-world graphs. The work [11] on pattern queries over labeled graphs proposed another pruning process, namely *R*-semijoin, using a special index called cluster-based *R*-join index. It can filter nodes that cannot possibly contribute to partial matches for an AD edge between two labeled query nodes. However, (1) the selected nodes may be still redundant since the nodes only satisfy the reachability condition imposed by one edge and the global structural satisfaction is not checked. (2) It is highly costly to construct and store the *R*-join index for a large data graph since the index essentially precomputes and stores all matches for pairwise labels and the index size is quadratic in the graph size. (3) It cannot be used to perform pruning for queries that have expressive attribute predicates rather than a fixed set of labels associated with nodes. Since predicates of query nodes are often not fixed and predictable, the index actually cannot be precomputed and this approach cannot be used.

We explore the potentials of existing reachability index for effective pruning. It is interesting to note that most reachability indexing schemes follow a paradigm. They first utilize a relatively simple reachability index which often assigns two or three labels to each node in order to cover the reachability of a substructure, called a cover, such as tree-cover in [2, 26], path-tree in [18], and chain-cover in [9, 19]. To cover the remaining reachability information, each node keeps one or two lists where complete or just a portion of ancestors and descendants are stored. When answering whether a node can reach another, the algorithms typically use nodes stored in the lists as the intermediate to determine the reachability.

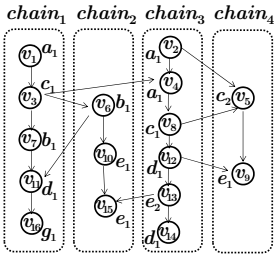
When it comes to answer a number of reachability queries between two sets of nodes, the following two observations are helpful: (1) the lists of different nodes often share a number of nodes, (2) the nodes in different lists have rich reachability information. If we merge the lists of a set of nodes by eliminating the duplicates and those whose reachability information can be derived from others, the merged list "subsumes" all the reachability information in the original lists of the node set but the size will not be much larger, and possibly even much smaller, than the list size of any individual node. Using the merged list, reachability patterns are likely to be evaluated more efficiently.

For example, considering a reachability pattern u_A—u_B, we want to filter data nodes in $mat(u_A)$ that cannot reach any nodes in $mat(u_B)$. Instead of performing $|mat(u_A)| \times |mat(u_B)|$ pairwise reachability queries to check for each node $v \in mat(u_A)$ whether it can reach a node $v' \in mat(u_B)$, (1) we merge all index lists of $mat(u_B)$ to a single list of the minimum size that preserves all the reachability information saved in the original lists; and (2) for each $v \in mat(u_A)$, use the list of v and the merged list rather than individual lists for $mat(u_B)$ to holistically determine whether v reaches some node in $mat(u_B)$. Intuitively, we can regard the set $mat(u_B)$ as a single dummy node which is reachable from all nodes that are ancestors of nodes in $mat(u_B)$.

In this paper, we use 3-hop [19] as the underlying reachability index scheme, as 3-hop has both a very compact index size and reasonable query processing time. As different labeling schemes are often preferable to different graph structures, it is also very flexible for our framework to use other labeling schemes to efficiently process different types of graphs.

We restrict our attention to in-memory processing and do not address the issues relating to disk-based access methods and physical representation of graph data.

Algorithm outline. Our GTPQ evaluation algorithm (referred to as GTEA) is outlined as follows. First, it prunes candidate matching nodes that do not satisfy downward structural constraints (i.e. not satisfy the subtree pattern query rooted at the corresponding



node	L_{in}	L_{out}
v_3		v_4
v_5	v_8	
v_6	v_3	v_{11}
v_9	v_{12}	
v_{15}	v_{13}	

Figure 5: Chain decomposition and 3-hop index

query node). Second, it performs the second round pruning process on a carefully selected subtree pattern, called prime subtree, to remove nodes not satisfying upward structural constraints (i.e. not reachable from any candidate nodes of the root). Third, the prime subtree is further shrunk if possible, and GTEA generates the matches of the shrunk prime subtree while representing the intermediate results as a graph, from which the final results can be efficiently obtained. We begin with focusing on evaluating GTPQs with AD edges only and show how to extend the algorithm to process PC edges in Section 4.4.

4.2 Pruning Candidate Matching Nodes

We use a two-round pruning process to filter unqualified data nodes. The first round selects data nodes that satisfy downward structural constraints of the query pattern for each query node. At the second round, we then obtain a minimum subtree that contains all output nodes having more than one candidate matching node, and select necessary edges from this subtree to find nodes satisfying upward structural constraints.

4.2.1 Preliminary: Merging 3-hop index

3-hop is a recent graph reachability indexing scheme well-known for its compact index size and reasonable query time. It follows the indexing paradigm mentioned in Section 4.1. It uses the chain-cover which consists of a set of disjoint chains covering all nodes in the graph. Each node in the graph is assigned a chain ID cid and its sequence number sid on its chain. For two nodes v and v' on the same chain (i.e., $v.cid = v'.cid$), $v \leq_c v'$, if $v.sid \leq v'.sid$. In particular, if $v.sid < v'.sid$, we say v is *smaller* than v'. Obviously, reachability on the chain-cover can be answered using chain IDs and sequence numbers. To encode the remaining reachability information outside chain-cover, 3-hop records a successor list $L_{out}(v)$ $\big($resp. predecessor list $L_{in}(v)\big)$ of "entry" (resp. "exit") nodes to (resp. from) other chains for each node v. The entry (resp. exit) node to (resp. from) a chain is the smallest (resp. largest) one on that chain that v reaches (resp. reaches v). See [19] for details of 3-hop index construction. For answering the reachability between two nodes v_1 and v_2 on different chains, 3-hop takes the following steps. (1) Collect the smallest nodes on any other chain that v_1 can reach through exit nodes of chain $v_1.cid$. That is, we get a set of nodes $X_{v_1} = \{x | x \in \bigcup_{v_1 \leq_c v'} L_{out}(v')$ and $\forall v' \geq_c v_1, x \leq_c L_{out}^{x.cid}(v')\} \cup \{v_1\}$, where $L_{out}^{x.cid}(v')$ is the entry node of v' on chain $x.cid$. We call X_{v_1} the *complete successor list* of v_1. (2) Collect the largest nodes on any chain that can reach v_2 through entry nodes of chain $v_2.cid$. In this step, we get a set of nodes $Y_{v_2} = \{y | y \in \bigcup_{v' \leq_c v_2} L_{in}(v')$ and $\forall v' \leq_c v_2, L_{in}^{y.cid}(v') \leq_c y\} \cup \{v_2\}$, where $L_{in}^{y.cid}(v')$ is the exit node of v' on chain $y.cid$. We call Y_{v_2} the *complete predecessor list* of v_2. (3) If there is a pair $(x, y)(x \in X_{v_1}, y \in Y_{v_2})$ such that $x \leq_c y$, then we can conclude that v_1 can reach v_2.

Example 7. Fig. 5 gives a chain decomposition of G of Fig. 2(a) and the corresponding 3-hop index. Chain IDs and sequence numbers are omitted. As an example, $v_3.cid = v_{11}.cid = 1$, $v_{11}.sid = 4$ and $v_3.sid = 2$. Because $v_3.sid < v_{11}.sid$, $v_3 \leq_c v_{11}$ and v_{11} is reachable from v_3. To answer whether v_3 can reach v_9, we collect the entry nodes in $L_{out}(v_i)(i = 3, 7, 11, 16)$ into $X_{v_3} = \{v_3, v_4\}$. Then we look up the exit nodes in $L_{in}(v_j)(j = 9, 5)$ and get $Y_{v_9} = \{v_9, v_{12}\}$. Since there is a pair (v_4, v_{12}) such that $v_4 \in X_{v_3}, v_{12} \in Y_{v_9}$, and $v_4 \leq_c v_{12}$, we say v_3 can reach v_9. □

Procedure 2: MergePredLists

Input: A set of nodes S.
Output: The predecessor contour C^p of S.
1. **for each** node $v \in S$ **do**
2. **if** $C^p[v.cid] < v.sid$ **then** $C^p[v.cid] := v.sid$
3. $v' := v$
4. **repeat**
5. **for each** index node $v'' \in L_{in}(v')$ **do**
6. **if** $C^p[v''.cid] < v''.sid$ **then**
7. $C^p[v''.cid] := v''.sid$
8. $v' :=$ prev(v')
9. **until** $v' = null$ or $visited_{v'.cid} \geq v'.sid$
10. **if** $visited_{v.cid} < v.sid$ **then** $visited_{v.cid} := v.sid$
11. **return** C^p

Note that to obtain the complete predecessor (resp. successor) lists, the original 3-hop needs to visit all larger (resp. smaller) nodes. We can assign a forward (and backward) tracing pointer to each node which points to the smallest larger (resp. largest smaller) node whose L_{out} (resp. L_{in}) list is nonempty so that nodes with empty lists can be skipped. We define two operations *next*(v) and *pre*(v) on each node v, which return the node that the forward and the backward tracing pointer points to respectively. For example, since v_6 is the largest smaller node that has a non-empty L_{in} w.r.t. v_{15}, prev$(v_{15}) = v_6$.

A basic operation of the pruning process is merging the complete predecessor/successor lists for a given set of data nodes (denoted by S). For the 3-hop case, it picks the largest (resp. smallest) nodes on each chain from the complete predecessor (resp. successor) list and we call the resultant list *predecessor contour* C^p (resp. *successor contour* C^s). A node v is said to reach (resp. be reachable from) S if v reaches (resp. is reachable from) at least one node in S. We have the following proposition.

Proposition 7. *A data node v reaches $mat(u)$ iff there is a pair $(x, y) \in X_v \times C^p$ such that $x \leq_c y$, while $mat(u)$ reaches v iff there exists a pair $(x, y) \in C^s \times Y_v$ such that $x \leq_c y$.* □

Procedure 2 sketches the process of calculating the predecessor contour C^p, where $visited_i$ records the largest node on chain i whose predecessor list has been looked up. For each node $v \in S$, MergePredLists processes v and those smaller nodes whose predecessor lists have not been looked up as follows. For each node v' to be processed and each exit node v'' in $L_{in}(v')$, it compares v'' with the nodes in C^p on the same chain of v'', and update C^p if v'' is larger (line 4–9). To retrieve nodes from C^p efficiently, C^p can be implemented as a map that uses chain IDs as keys and the sequence numbers as values.

Example 8. We show how to compute the predecessor contour of $mat(u_{10})$ for the query Q of Fig. 2. Example 3 have given that $mat(u_{10}) = \{v_9, v_{10}, v_{13}, v_{15}\}$. The procedure collects the complete predecessor lists for each of $mat(u_{10})$ one by one, but no predecessor list is repeatedly visited. For example, assume that v_{10} is read before v_{15}. When collecting $Y_{v_{15}}$, although prev(v_{15}) points to v_6, MergePredLists needs not look up $L_{in}(v_6)$, because the list has been looked up when collecting $Y_{v_{10}}$. The predecessor contour of $mat(u_{10})$ is $\{v_3, v_9, v_{13}, v_{15}\}$. It can be easily verified that the size of this predecessor contour is a half of the total size of the four individual complete lists of v_9, v_{10}, v_{13} and v_{15}. Note

Procedure 3: PruneDownward

Input: 3-hop index L_{out}, a GTPQ Q.
Output: Updated candidate matching nodes satisfying downward structural constraints.

1. **for each** node $u \in V_q$ **do** $mat(u) := \{x | x \in V, x \sim u\}$
2. **for each** leaf node u' in V_q **do** $C^p_{u'} :=$ MergePredLists$\big(mat(u')\big)$
3. $V'_q = V_q \backslash \{u' | u' \text{ is a leaf node}\}$
4. **for each** $u \in V'_q$ in bottom-up order **do**
5. **for each** $v \in mat(u)$ **do** $chain_{v.cid} := chain_{v.cid} \cup \{v\}$
6. **for each** $chain_i$ that is not empty **do**
7. **for each** child u' of u **do** $val[p_{u'}] := 0$
8. **for each** node $v_i \in chain_i$ **do**
9. **for each** child u' of u *s.t.* $val[p_{u'}] = 0$ **do**
10. **if** v_i reaches $mat(u')$ **then** // using Proposition 7
11. $val[p_{u'}] := 1$
12. **if** $f_s(u)$ evaluates to false with the valuation val **then**
13. $mat(u) := mat(u) \backslash \{v_i\}$
14. $C^p_u :=$ MergePredLists$\big(mat(u)\big)$

Procedure 4: PruneUpward

Input: 3-hop index L_{in}, the prime subtree (V_t, E_t).
Output: Updated candidate matching nodes satisfying upward structural constraints.

1. $C^s_{u_{root}} :=$ MergeSuccLists$\big(mat(u_{root})\big)$
2. $V_t := V_t \backslash \{u_{root}\}$
3. **for each** node $u \in V_t$ in top-down order such that $|mat(u)| > 1$ **do**
4. **for each** child u' of u such that $|mat(u')| > 1$ **do**
5. **for each** node $v \in mat(u')$ **do**
6. $chain_{v.cid} := chain_{v.cid} \cup \{v\}$
7. $Group_v := Group_v \cup \{u'\}$
8. **for each** node v_i in a nonempty $chain_i$ **do**
9. **if** $mat(u')$ do not reach v_i **then** // using Proposition 7
10. **for each** $u' \in Group_{v_i}$ **do**
11. $mat(u') := mat(u') \backslash \{v_i\}$
12. **else break**
13. **for each** non-leaf child u' of u **do**
14. $C^s_{u'} :=$ MergeSuccLists$\big(mat(u')\big)$

that the size of a predecessor contour is bounded by the number of chains. This example actually gives the worst case but still has a high compression rate (50%). □

Time complexity. The time complexity of the procedure is $O(|S| + |L_{in}|)$, where $|L_{in}|$ is the total size of all predecessor lists in 3-hop index. It can be observed from the fact that no index node in a predecessor list has been ever repeatedly visited.

Following the same line of MergePredLists, we develop MergeSuccLists that calculates the successor contour of a node set with time complexity of $O(|S| + |L_{out}|)$, where $|L_{out}|$ is the total size of all successor lists in 3-hop index.

4.2.2 *Pruning process for downward structural constraints*

Procedure 3 describes the first round of the pruning process. In the procedure, val refers to a valuation for variables associated with query nodes. PruneDownward first collects $mat(\cdot)$ sorted in the descending order of sequence numbers for each query node and calculates the predecessor contours for leaf nodes (line 1–2). Then, it processes each non-leaf query node u following a bottom-up fashion (line 4–14). For each node u, it first groups nodes $mat(u)$ by chain ID (line 5). Then for each candidate matching node v_i of u on each chain i, PruneDownward checks whether v_i satisfies downward structural constraints (line 8–13). To do this, (1) it first assigns a valuation to $p_{u'}$ for each child node u' of u according to the reachability from v_i to $mat(u')$ (line 9–11) , (2) and then remove v_i from $mat(u)$ if the structural predicate $f_s(u)$ of u evaluates to false under the valuation (line 12–13). Note that when processing the next node on the same chain, the valuation for the previous node is inherited due to the transitive property of transitive closure in a chain. Therefore, no predecessor list is repeatedly looked up. After all candidate matching nodes for u have been processed, the remaining data nodes in $mat(u)$ must satisfy the downward structural constraints. Then the predecessor contour for u is computed (line 14), and used in the pruning process of the parent node of u. The procedure terminates after the root is processed.

Example 9. We first show how procedure PruneDownward prunes $mat(u_3)$ of Fig. 2. In a bottom-up fashion, before pruning $mat(u_3)$, PruneDownward first processes its non-leaf child u_7. No nodes in $mat(u_7)$(i.e. $\{v_6, v_7\}$) are removed, because v_6 can reach both $mat(u_9)$ and $mat(u_{10})$ while v_7 can reach $mat(u_{10})$. The predecessor contour for $mat(u_7)$ is then computed and $C^p_{u_7} = \{v_6, v_7\}$. For determining whether v_5 should be removed from $mat(u_3)$, PruneDownward checks the reachability between v_5 and $mat(u_6)$, $mat(u_7)$, $mat(u_8)$ respectively by using the predecessor contours. One can verify that v_5 cannot reach $mat(u_6)$, which means $val[p_{u_6}] = 0$ and the structural predicate $f^{v_5}_s(u_3)$ evaluates to true. Thus, v_5 remains in $mat(u_3)$. Because the other two nodes v_3 and v_8 are in different chains, they do not inherit the valuation determined by v_5 and PruneDownward needs to check pairwise reachability between $\{v_3, v_8\}$ and $\{mat(u_6), mat(u_7), mat(u_8)\}$. Only v_8 is subsequently removed, because $p_{u_8} = 1, p_{u_6} = p_{u_7} = 0$ and $f^{v_8}_{ext}(u_3)$ evaluates to false. Finally, after this pruning round, $mat(u_3) = \{v_3, v_5\}$.

When PruneDownward refines $mat(u_1)$ and reads v_2, the assignments of p_{u_2} and p_{u_3} are directly inherited from the result computed in the previous step of processing v_4 and $f^{v_2}_{ext}(u_1)$ immediately evaluates to true without any index lookups.

PruneDownward gets the following refined candidate matching nodes which satisfy the downward structural constraints: $mat(u_2) = \{v_3, v_8\}, mat(u_3) = \{v_3, v_5\}$. □

Time complexity. Since no successor list is repeatedly checked, the 3-hop index is looked up for at most $|E_q||L_{out}|$ times, where $|E_q|$ is the number of edges in the tree pattern. MergePredLists is invoked $(|V_q| - 1)$ times to compute predecessor contours for each non-root query node, and the total time cost is $O(|V_{mat}| + |V_q||L_{in}|)$, where $|V_q|$ is the number of query nodes and $|V_{mat}|$ is the total size of initial candidate matching nodes (i.e. $|V_{mat}| = \Sigma_i |mat(u_i)|$). Therefore, PruneDownward is in $O(|V_q|(|L_{in}| + |L_{out}|) + |V_{mat}|)$ time.

4.2.3 *Pruning process for upward structural constraints*

After the fist-round pruning process, for each backbone node u, the remaining nodes in $mat(u)$ satisfy all the structural constraints imposed by predicates. Because the results of the query should consist of matches of output nodes only, the matches for predicate nodes are no longer useful and do not need to be considered. Moreover, some backbone nodes may not contribute to determining which candidate matching output nodes are in the same instance and hence can be also discarded. With these two observations, the structural constraints of a backbone subtree are enough to derive the relationships among candidate matching nodes for the output query nodes. Such a subtree, we call the *prime subtree*, can be induced by the paths from the query root to all such output nodes that $|mat(\cdot)| > 1$. The next pruning step only needs to consider this subtree pattern which in essence is reduced to a conjunctive GTPQ.

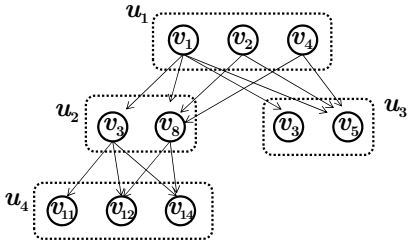


Figure 6: Example of the maximal matching graph for Q over G depicted in Fig. 2

In the opposite direction to PruneDownward, procedure PruneUpward (Procedure 4) traverses down the prime subtree. For each query node u, it filters the candidate matching nodes of each child u' of u (line 3–14). All the candidate nodes to be processed are first clustered and merged into duplicate-free sets according to their chain IDs, where the order of nodes is reversed (line 4–7). As a data node can match multiple query nodes, the algorithm uses $Group_v$ to record the corresponding query nodes that v matches (line 7) in order to update $mat(\cdot)$ when a reachability condition is determined (line 10–11). Then, for each node $v_i \in mat(u')$ on a nonempty $chain_i$, v_i should be removed if $mat(u)$ cannot reach v_i according to Proposition 7. Observe that once a node is confirmed to satisfy the condition of the incoming edge, all other larger nodes do not need to be checked since they must also satisfy the condition.

Example 10. In this example, assume that u_2 and u_3 are output nodes of Q of Fig. 2. The prime subtree is induced by u_1, u_2 and u_3. PruneUpward starts from u_1 to refine $mat(u_2)$ and $mat(u_3)$. After grouping distinct data nodes into $chain$, it gets $chain_1 = \{v_3\}$, $chain_3$=$\{v_8\}$, and $chain_4 = \{v_5\}$. v_3 is in both $mat(u_2)$ and $mat(u_3)$, but the procedure only stores one copy in $chain$ to avoid processing it repeatedly when checking reachability with $mat(u_1)$. After the two query nodes whose matching candidate nodes have the identical v_3 are inserted to $Group_{v_3}$, $Group_{v_3} = \{u_2, u_3\}$. Because $mat(u_1)$ reaches v_3, v_3 is not removed from either $mat(u_2)$ or $mat(u_3)$. Similarly, it can be verified that $mat(u_1)$ can reach v_8 and v_5. In the end, none is removed from $mat(u_2)$ and $mat(u_3)$ after this pruning round. □

Time complexity. The time complexity is $O(|V'_{mat}| + (|L_{in}| + |L_{out}|)|V'_t|)$, where $|V'_t|$ is the number of internal nodes in the prime subtree and $|V'_{mat}|$ is the total size of the remaining candidate matching nodes after the first pruning round.

4.3 Computing Final Results

Shrunk prime subtree. As a result of the pruning process, the matching output nodes are guaranteed to be in the answer. The left to do is to identify how they form the final results by computing the matches of edges in the prime subtree. Given a prime subtree, assume that u is the lowest common ancestor of all output nodes. We can further shrink the subtree by (1) removing the ancestors of u if u is not the root, and (2) removing all such nodes u' that $|mat(u')| = 1$. If the removing process leads to disjoint subtrees, we just compute results for each subtree, do a Cartesian product of them and add the candidate matching nodes of removed output nodes to assemble the whole final results. From now on, we only need to compute edge matches for the shrunk prime subtree(s).

Example 11. The shrunk prime subtree of Q of Fig. 2 is induced by u_2 and u_4. Even if we change the query to mark u_5 also as an output node, the shrunk prime subtree is *still* the same since $|mat(u_5)| = |\{v_{13}\}| = 1$ and v_{13} must be in every answer. □

Maximal matching graph. The full matches of the shrunk prime subtree can be represented by a *maximal matching graph* $Q_g(G) = (V_r, E_r)$, where (1) $V_r \subseteq V$ such that $v \in V_r$, if there is a query node $u \in V_q$ such that $v \models u$; (2) $E_r \subseteq V_r \times V_r$ such that $(v_1, v_2) \in E_r$, if (v_1, v_2) is a match of an edge $(u_1, u_2) \in E_q$.

Procedure 5: CollectResults

Input: The maximal matching graph $MaximalGraph$, a query node u and one of its candidate matching node v.
Output: the answer to the subGTPQ rooted at u and dominated by v.

```
1.  if v is a leaf node then return {u : v}
2.  else
3.      results := ∅
4.      for each branch list bch of v do
5.          branchResults := ∅
6.          for each node v' that a pointer in bch points to do
7.              branchResults := branchResults ∪
                  CollectResults(MaximalGraph, v')
8.          results := results × branchResults
9.      if u is an output node then results := {u : v} × results
10.     return results
```

We group the nodes and edges in the graph according to what query nodes and edges they match. Specifically, in an implementation, each node v has several branch lists, each of which corresponds to the child of the query node that v matches and includes pointers pointing to nodes matching the child.

Example 12. Recall the GTPQ Q and data graph G in Fig. 2. Let u_2, u_3 and u_4 be output nodes. Fig. 6 shows the corresponding maximal matching graph. As an example, v_1 has two branch lists corresponding to the two incident query edges, denoted by bch_1 and bch_2 respectively. $bch_1 = \{ptr_{v_3}, ptr_{v_8}\}$, and $bch_2 = \{ptr_{v_3}, ptr_{v_5}\}$, where $ptr_{v_i} (i = 3, 5, 8)$ is pointer to v_i. □

Computing the maximal matching graph. Since the nodes of the maximal matching graph have been obtained after the pruning process, we only need to compute matches for each query edge whose head and tail both have more than one matching node. Given a query edge (u_1, u_2), a straightforward way is to check the reachability between nodes in $mat(u_1)$ and $mat(u_2)$ using 3-hop index. The time complexity is $O((|L_{in} + L_{out}|)|E_q||V_{mat}|^2_{max})$, with $|V_{mat}|_{max}$ being the maximal size of the candidate matching nodes after the pruning process. Since in practice many queries are highly selective and $|V_{mat}|_{max}$ is presumably pretty small, the straightforward way is expected to be fast and practical.

A more sophisticated approach that we choose is to utilize the similar technique used in procedure PruneUpward. Observe that the loop from line 9 to 12 in PruneUpward is to determine whether a data node matching some child of u is reachable from $mat(u)$. By replacing C^s_u with the successor list of a node v, we can simultaneously get all edges from v in the maximal matching graph in $O(|L_{in}| + |L_{out}| + |E_v|)$, where $|E_v|$ is the out-degree of v in the resulting graph. The total time complexity then is $O((|L_{in}| + L_{out})|V^{inter}_{mat}|+|E_{mg}|)$, where $|V^{inter}_{mat}|$ is the number of candidate matching nodes for internal query nodes and $|E_{mg}|$ is the number of edges in the resulting maximal matching graph.

Enumerating results. We next present procedure 5, referred to as CollectResults, which derives final results from the maximal matching graph. Each result is in a tuple format. To avoid ambiguity in presentation, we explicitly specify in the tuple which query node a data node matches. Specifically, each element in a tuple is of the form $u : v$, which means v is an image of u in a match.

Procedure CollectResults traverses down the maximal graph. For a leaf node, since its corresponding query node must be an output node, the procedure returns a tuple with only an element of it (line 1). For an internal node, it collects results from each child for every branch list, and then does a Cartesian product of them (line

4-8). If the query node it matches is an output node, it is inserted into each result (line 9). The final answer to the query is the union of the results of those nodes matching the query root. When query nodes in the shrunk prime subtree are all output nodes, no redundant intermediate results would be produced. Note that no existing algorithms for pattern queries on graphs can achieve this. When there are non-output query nodes in the shrunk prime subtree, our algorithm is not duplicate free. Recall Example 12. The results obtained from v_1 are the same as those obtained from v_3, since u_1 is not an output node and v_1 can reach v_3. However, the duplicate intermediate tuples are a subset of the counterpart of other works, because (1) the prime subtree we pick is a minimum subtree of the original query pattern that contains all output nodes, (2) for non-output nodes, the algorithm merges the intermediate partial results in advance (line 7).

4.4 Evaluating Queries with PC Edges

In the context of graph database, the research on pattern queries often focuses on reachability patterns. Indeed, the reachability pattern query is an important building block for other queries. Adding PC edges to a pattern significantly increases the complexity of evaluation. Even for tree-structured data, [25] has theoretically demonstrated the difficulty of handling TPQs with arbitrary combination of PC and AD edges. [25] has proved that no holistic algorithms can achieve optimality as for queries with AD edges only. For graph-structured data, the evaluation of conjunctive pattern queries whose edges all represent PC relationship is essentially a computationally-hard labeled graph isomorphism problem. Nevertheless, we can use the similar idea of our framework to support GTPQs with PC edges.

When processing a node u in PruneDownward: (1) if u has only PC outgoing edges, we merge the set of parents of $mat(u')$ for each child u' of u into $P_{u'}$, instead of computing the predecessor contours. Then we sort $mat(u)$ and each $P_{u'}$, and check for each node v in $mat(u)$ whether it is in some $P_{u'}$ in a multiway merge-sort style. If yes, then $val[p_{u'}] := 1$, otherwise $val[p_{u'}] := 0$. (2) If u has both AD and PC edges, we process these two type of edges separately to refine $mat(u)$. Similarly, when performing PruneUpward, we collect sets of children of $mat(u)$ instead of computing the successor contour.

After the pruning stage, all candidate matching nodes are guaranteed to be in final results. To compute the maximal matching graph, we can either do nested joins to check the adjacent relationships, or perform multiway merge-join to derive the adjacent edges in the resulting graph. Other operations including determining the prime subtree and enumerating final results are the same.

Alternatively, we can also use another strategy to deal with PC edges. Regarding PC edge as a special type of AD edge, we can first process PC edges in the same way with AD edges in the process of pruning, except those whose tail's structural variable is the operand of a negation operator and which need to be processed as stated before. The prime subtree becomes a minimum subtree that contains all output nodes and those PC edges that are regarded as AD edges when pruning. After computing the maximal matching graph, we check whether the two incident nodes of the corresponding edge in the maximal matching graph are adjacent in the data graph and remove them if not. Next, the unsatisfied nodes are removed in a top-down fashion, followed by enumerating final results. We use this strategy in our implementation.

5. EXPERIMENTAL EVALUATION

In this section, we present an experimental study using both real-life and synthetic data to evaluate (1) the efficiency and scalability of our algorithm, (2) the effectiveness of representing intermediate results as graphs, and (3) the efficiency of the pruning process.

Table 1: Statistics of XMark datasets

Scaling factor	0.5	1	1.5	2	4
Dataset size (MB)	55	111	167	223	447
Nodes (Million)	0.64	1.29	1.94	2.52	5.17
Edges (Million)	0.77	1.54	2.32	3.09	6.20

We only give the experimental results for conjunctive TPQs with all query nodes being output nodes (i.e. the traditional TPQs). We found that our algorithm has better performance than other algorithms even for them. Since there has been no other algorithms designed for GTPQs and the decomposition-based approach that may be applied on top of them to process GTPQs incurs high overhead as analyzed in Related work and empirically demonstrated in prior studies [16] and [29], our algorithm can do even far better for general GTPQs than those algorithms, compared to the results reported here. Additional experimental results concerning I/O cost and the results on GTPQs with disjunctive and negative predicates can be found in [1].

Implementation. We have implemented the algorithm proposed in Section 4 (GTEA), TwigStack [4], Twig^{2}Stack [8], TwigStackD [7] and HGJoin [27]. TwigStack is the classical holistic twig join algorithm. Twig^{2}Stack is the latest algorithm for evaluating TPQs on tree-structured data which has a distinct feature of representing results in hierarchical stacks. Other algorithms for tree-structured data that can support disjunction and/or negation, such as BTwigMerge [5] and TwigStackList$\neg$ [29], are in essence the same as TwigStack with respect to the conjunctive TPQs and hence are not included in our experiments. TwigStackD can evaluate conjunctive TPQs over graph-structured data. In our implementation, we fixed the problems in the original paper [30]. HGJoin is a hash-based structural join algorithm for processing graph pattern queries. We did not implement the query plan generation in the original algorithm which relies on selective estimation techniques [22] and takes exponential time in the query size; instead, for each query, we generated all valid plans and took evaluation on each. The minimum query processing time on the best plan is reported; thus, the time presented in this paper is always smaller than the real time of the original HGJoin. This version is denoted by HGJoin+. By representing intermediate results as graphs, we have also implemented another version denoted by HGJoin*. All experiments are performed on a 2.4GHz Intel-Core-i3 CPU with 3.7 GB RAM.

5.1 On XMark Data

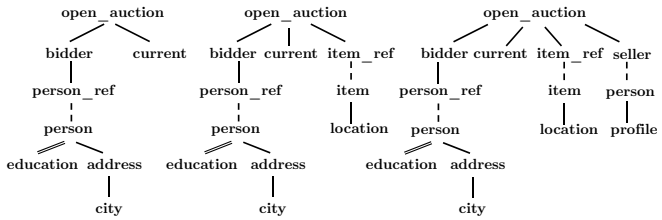
In this set of experiments, we use large synthetic XMark data [24] to evaluate the efficiency and scalability of various algorithms. As mentioned in Section 1, many graph-structured XML database can be modeled by a special form of graphs consisting of trees connected by cross edges (ID/IDREF links). In this case, we can use existing twig join algorithms to process conjunctive TPQs by decomposing them into a set of subqueries on separative trees. We use TwigStack and Twig^{2}Stack to investigate the efficiency of applying this approach.

Datasets. We generated five XMark datasets with the scaling factors from 0.5 to 4. For each dataset, we generate a graph, where nodes correspond to XML elements and edges represent the internal links (parent-child) and ID/IDREF links. The attribute for graph nodes is the tag of elements except for nodes corresponding to person, item elements, for each type of which we randomly classify them into ten groups to represent different properties. A label is assigned to each node according to the tag or the group it belongs to. Distinct labels indicate different attribute values. The details of the generated documents and graphs are presented in Table 1.

Queries. Three types of queries we used for experiments are depicted in Fig. 7, where dotted edges refer to ID/IDREF links in

Table 2: The average size of query results on XMark

Queries	55M	111M	167M	223M	447M
Q_1	368	762.8	1115.8	1496.8	2986.8
Q_2	34.6	75.8	117.8	150.3	297.2
Q_3	1.9	4.1	5.8	6.1	17.1



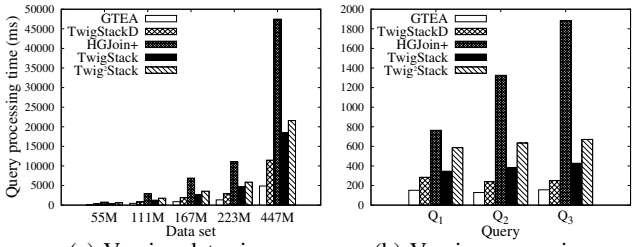
(a) Q_1 (b) Q_2 (c) Q_3

Figure 7: Queries for XMark data

the original data. For each query type, we generated ten queries by randomly choosing a label for each of person and item nodes representing a different attribute predicate. The average is reported.

Experimental results. Fig. 8(a) shows the query evaluation time for Q_1 on datasets varying the data size. The results for Q_2 and Q_3 are quite similar. The results reveal the following. (1) GTEA constantly outperforms all other algorithms. Specifically, GTEA is three times to more than one order of magnitude faster than TwigStack and Twig^{2}Stack, five times to more than two orders of magnitude faster than HGJoin, and in the best cases three times faster than TwigStackD. When data size becomes larger, the performance gain by GTEA becomes more significant. (2) TwigStackD also has very good performance in this set of experiments with the following reasons. (a) It utilizes SSPI, a reachability index with pretty small size and good querying time for tree-like graphs. (b) Its basic idea is extended from the holistic twig join algorithms, and so TwigStackD also has the advantages taken by the stack encoding and the blocking method for path results [4]. (c) Although TwigStackD has to buffer every nodes in pools (a special structure used to store nodes popped from stacks) and large amounts of the operations of checking edge conditions with all nodes in pools have to be done (indicated as reasons of inefficiency in [27] and [11]), the pre-filtering process it uses can filter redundant nodes and relieve the cost of the above operations. Indeed, without the pre-filtering process, TwigStackD is slower by orders of magnitude [30]. (3) It is sort of surprising that TwigStack has slightly better performance than Twig^{2}Stack. The reason is that although Twig^{2}Stack can avoid generating path matches (as a primary reason for the efficiency in [8]), the overhead brought by merging stack trees and maintaining the hierarchical structures overrides the benefits in the experiments. The fact that the depth of XMark graphs is small (with an average of 5), also make the hierarchical stack encoding have not a strong advantage. Besides, the enumeration of path matches (as a reason for inefficiency for TwigStack in [8]) can be done fast using the blocking technique. (4) HGJoin has the worst performance, mainly because (a) the structural-join way has to generate a large number of (largely redundant) intermediate results for small substructures and (b) non-trivial merge-join operations on them have to be done even with the best plan. The query processing time increases significantly when the size of data graphs increases.

Fig. 8(b) shows the results on the XMark dataset of scale 0.5 for different queries. (1) The query processing time of GTEA nearly maintains the same as the query size increases. In particular, the time cost for evaluating Q_2 is smaller than that for Q_1. It is because the size of the results of Q_2 is much smaller than that for Q_1 as presented in Table 2, resulting in smaller cost for enumerating the final results. (2) The processing time of TwigStack and Twig^{2}Stack does



(a) Varying data size (b) Varying query size

Figure 8: Performance results on XMark data

not increase significantly over Q_1, Q_2 and Q_3, although they have to evaluate a increasing number of subqueries and perform a growing number of merge operations. Indeed, as shown in Table 2, the sizes of the results of Q_1 and Q_2, which are a subquery of Q_2 and Q_3 respectively, are small and thus the extra cost for evaluating Q_2 and Q_3 is very limited. (3) However, HGJoin is much more sensitive to the increase of the query size, which is due to the impact of the redundant intermediate results and expensive sort operations involved in performing multi-structural joins. The results for HGJoin highlight the crucial importance of using a pruning process to reduce the size of intermediate results not contributing to the answer.

5.2 On arXiv Data

In this set of experiments, we used a real-life graph to evaluate the performance of GTEA, TwigStackD and HGJoin for general graph data, verify the effectiveness of graph representation of intermediate results and the efficiency of the pruning process.

Dataset. We generated a graph from the HEP-Th database[1], originally derived from the arXiv[2]. There are paper nodes and author nodes, each associated with multiple properties. For simplicity, we assigned a label to each author node according to the email domain, and a label to each paper node based on its area and journal it is published in, to represent the attributes. The edges of the graph represent author or citation relationships. The graph has 9562 nodes, 28120 edges, and 1132 distinct labels.

Query generator. We designed a query generator to randomly produce meaningful queries. Each query node is associated with a label randomly chosen from the data graph to represent attribute predicates. Two groups of queries are generated: one has a small size of results between 2 and 50, the other has a large size between 200 and 1200. For each group, five sets of queries were generated with query size varying from 5 to 13. We generated fifteen different queries for each size scale and report the average. The average time can reflect the average case performance of each algorithm, since the queries are generated in a random way. The results for queries of distinct sizes in the same group are comparable, because the differences of the result sizes of the queries have little impact on the query processing time and the number of query results for each size scale follow a close distribution as illustrated in Fig. 9(a).

Experimental results. Fig. 9(b) and (c) report the results for the two groups of queries. They tell us the following. (1) GTEA has the best query processing time, significantly smaller than the processing time of other algorithms (more than one order of magnitude in most cases). It also has the best scalability in both two groups of experiments. (2) TwigStackD no longer has good performance as on XMark data. In fact, it has the longest querying time for queries with size of 5 to 9. The arXiv graph is much denser and deeper than XMark data, causing the inefficiency of the pool structure as well as SSPI. The problem of TwigStackD is highlighted by Fig. 9(c) where it fluctuates sharply for queries with large results. The results

[1] http://kdl.cs.umass.edu/data/hepth/hepth-info.html
[2] http://arxiv.org/

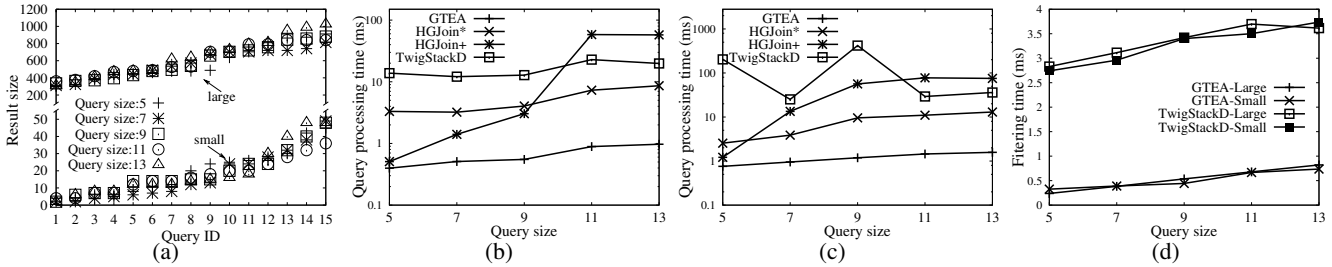


Figure 9: Performance results on arXiv data. (a) Distribution of the result sizes. (b) Query processing time on the queries with small sizes of results. (c) Query processing time on the queries with small sizes of results. (d) Comparison with the pre-filtering process.

reflect that TwigStackD has rather poor performance for particular queries. In contrast, GTEA is most robust since it always maintains good performance for all experiments. (3) HGJoin+ is not quite scalable similar to the performance on the XMark data. Yet it now has better performance than TwigStackD when the query size is smaller than 11. (4) The revised HGJoin (i.e. HGJoin*) has better scalability than HGJoin+. For the group of queries with large results, the query processing time of HGJoin* is smaller than that of HGJoin+ when the query size is larger than 7, compared to 11 for the group of queries with small results. This observation demonstrates that graph representation of intermediate results can improve the performance and achieve better scalability especially when there are many intermediate/final results and when the query size is large. The reason why the revised one takes more time than the original one for processing the queries of small sizes is that HGJoin* incurs costs for dynamically and recursively deleting unqualified nodes (not exist in our algorithm though), which offset the benefits taken by avoiding merge-join operations on tuples.

Fig. 9(d) evaluates the efficiency of our pruning process and the pre-filtering algorithm in TwigStackD, which clearly shows that our pruning method greatly outperforms the counterpart and also has better scalability with the query size. It is because the pre-fltering algorithm in TwigStackD requires two traversals of the data graph.

6. CONCLUSIONS

We have proposed the GTPQ, a new class of tree pattern queries on graph-structured data, which incorporates structural predicates defined in terms of propositional logic to specify structural conditions. We studied several fundamental problems, and established a general framework for evaluating GTPQs using a graph representation of graphs and a pruning approach. An algorithm has been developed for evaluating GTPQs, which can achieve a small size of intermediate results due to the effective pruning process and largely avoid generating redundant matches by dynamically shrinking the tree pattern during pruning and enumerating processes.

Acknowledgement. This work is supported by the National Science Foundation of China (61075074).

7. REFERENCES

[1] Full version. *CoRR*, abs/1109.4288, 2011.
[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, 1989.
[3] S. Amer-yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.
[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.
[5] D. Che, T. Ling, and W. Hou. Holistic boolean-twig pattern matching for efficient XML query processing. *TKDE*, PP(99):1, 2011.
[6] D. Chen and C.-Y. Chan. Minimization of tree pattern queries with constraints. In *SIGMOD*, 2008.
[7] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, 2005.
[8] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}stack: bottom-up processing of generalized tree pattern queries over XML documents. In *VLDB*, 2006.
[9] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.
[10] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In *VLDB*, 2003.
[11] J. Cheng, J. X. Yu, and P. S. Yu. Graph pattern matching: A join/semijoin approach. *TKDE*, 23:1006–1021, 2011.
[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, 2002.
[13] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.
[14] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *TKDE*, 19(10):1381–1403, October 2007.
[15] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.
[16] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with or-predicates. In *SIGMOD*, 2004.
[17] E. Jiao, T. W. Ling, and C. yong Chan. Pathstack¬: A holistic path join algorithm for path query with not-predicates on XML data. In *DASFAA*, 2005.
[18] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1), 2011.
[19] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.
[20] J. Lu, T. W. Ling, Z. Bao, and C. Wang. Extended XML tree pattern matching: Theories and algorithms. *TKDE*, 23(3):402 –416, 2011.
[21] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In *VLDB*, 2005.
[22] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for XML twigs. In *ICDE*, 2004.
[23] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.
[24] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: a benchmark for XML data management. In *VLDB*, 2002.
[25] M. Shalem and Z. Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *ICDE*, 2008.
[26] H. Wang, H. He, J. Yang, P. Yu, and J. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.
[27] H. Wang, J. Li, J. Luo, and H. Gao. Hash-base subgraph query processing method for graph-structured XML documents. *PVLDB*, 1(1):478–489, 2008.
[28] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, 2003.
[29] T. Yu, T. Ling, and J. Lu. Twigstacklist¬: A holistic twig join algorithm for twig query with not-predicates on XML data. In *DASFAA*, 2006.
[30] Q. Zeng and H. Zhuge. Comments on "stack-based algorithms for pattern matching on DAGs". *PVLDB*, 5(7):668–679, 2012.
[31] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.