Evaluation of Partial Path Queries on XML data

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

Xiaoying Wu
Dept. of CS, NJIT
xw43@njit.edu

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

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

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

ABSTRACT

XML query languages typically allow the specification of structural patterns of elements. Finding the occurrences of such patterns in an XML tree is the key operation in XML query processing. Many algorithms have been presented for this operation. These algorithms focus mainly on the evaluation of path-pattern or tree-pattern queries. In this paper, we define a partial path-pattern query language, and we address the problem of its efficient evaluation on XML data.

In order to process partial path-pattern queries, we introduce a set of sound and complete inference rules to characterize structural relationship derivation. We provide necessary and sufficient conditions for detecting query unsatisfiability and node redundancy. We show how partial path-pattern queries can be equivalently put in a canonical directed acyclic graph form. We developed two stack-based algorithms for the evaluation of partial path-pattern queries, `PartialMJ` and `PartialPathStack`. `PartialMJ` computes answers to the query by merge-joining the results of the root-to-leaf paths of a spanning tree of the query. `PartialPathStack` exploits a topological order of the nodes of the query graph to match the query pattern as a whole to the XML tree. The experimental evaluation of our algorithms shows that `PartialPathStack` is independent of intermediate results and largely outperforms `PartialMJ`.

1. INTRODUCTION

XML query languages typically allow the specification of structural patterns of elements. Finding the occurrences of such patterns in an XML tree is the key operation in XML query processing. Many algorithms have been presented for this operation. These algorithms focus mainly on the evaluation of path-pattern or tree-pattern queries. A restrictive feature of these queries is that they determine a total order for the elements in every path of the query. For instance, the path query in XPath `//year//author//title` retrieves `title` nodes from a bibliographic XML document. In this query, node `year` can only be an ancestor of node `author`, and node `author` can only be an ancestor of node `title`.

However, the standard query language for XML, XQuery, and even its core language, XPath, allow for structural patterns that do not form a complete path or tree. Consider, for instance, the following XPath query that involves reverse axes: `//title[descendant-or-self::*[ancestor-or-self::year][ancestor-or-self::author]]`. This query asks for `title` nodes in paths that also involve `year` and `author` nodes, but no specific order is required for those nodes in the path. In this sense, this is a path-pattern query where the structure of a path is partially specified in the pattern. Olteanu et al. [19, 18] show that XPath queries with reverse axes like the one shown above can be equivalently rewritten as a set of tree-pattern queries. However, they also show that this transformation may lead to an exponential blowup of the number of tree-pattern queries.

Further, in practice, there is a need to query XML data when the structure is not fully known to the user, or to query XML data sources with different structures in an integrated way [7, 11, 15, 20]. In order to deal with these problems, query languages are adopted that relax the structure of a path in a tree pattern. An extreme case are keyword-based languages for XML [9, 7, 11, 15]. Clearly, these query languages need to be accompanied with efficient evaluation techniques.

Partial path query language. In this paper, we define a query language that allows a partial specification of path patterns. Queries in this language do not require a total order for the nodes in the pattern. The language is general enough to encompass on the one side path-pattern queries and on the other side queries without structural relationships (nodes lying on the same path without order). This language can express different types of XPath expressions as, for instance, those mentioned above. It is also the constituent component of partial tree-pattern queries [20, 21].

The problem. We address the problem of efficiently evaluating partial path-pattern queries. A partial path-pattern query can be expressed equivalently by a set of path-pattern queries. There are several algorithms for evaluating path-pattern queries. For instance, Bruno et al. [2] provide an algorithm which is asymptotically optimal for path-pattern queries without repetitions of the same node label in the pattern for the stream data model. However, as mentioned earlier, this equivalent set may contain a number of path patterns, which is exponential on the number of query nodes. Clearly, this drawback does not suggest for efficient evalua-

CIKM '07 Lisboa, Portugal

tion techniques through the generation of an equivalent set of path patterns. Therefore, we focus on techniques that can directly process and match the partial path pattern to the XML data tree.

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

- Because the structure of a path may not be fully specified in a partial path-pattern query, new structural expressions can be derived from those explicitly specified in the query. These structural expressions are important in query processing. We define a sound and complete set of inference rules to fully characterize structural relationship derivation.
- Unlike path-pattern queries, partial path-pattern queries can be unsatisfiable. We provide necessary and sufficient conditions for detecting query unsatisfiability. Detecting unsatisfiable queries prevents accessing the data, which can be very large, at a small overhead.
- Partial path-pattern queries can contain redundant nodes, i.e., nodes that can be removed without affecting the meaning of the query. We provide conditions for efficiently identifying redundant nodes. We show how partial path-pattern queries can be equivalently put in a canonical form, which is a directed acyclic graphs (dag). We exploit the canonical form of queries to design query evaluation algorithms.
- We developed a stack-based algorithm `PartialMJ` for the evaluation of partial path-pattern queries. `PartialMJ` extracts a spanning tree from the query dag. It uses an extension of algorithm `PathStack` [2] to compute the results of the root-to-leaf paths of the spanning tree. These results are produced in root-to-leaf order in the XML tree and are merge-joined to compute the answer of the query. `PartialMJ` may generate *intermediate* results for the root-to-leaf paths of the spanning tree that cannot contribute to the answer of the query.
- To overcome the intermediate result problem, we developed a novel holistic stack-based algorithm `PartialPathStack` for the evaluation of partial path-pattern queries. `PartialPathStack` exploits a topological order of the nodes in the query dag, and matches the query dag as a whole to the XML tree. We analyze the complexity of `PartialPathStack`, and we show that it is independent of intermediate results.
- We implemented both algorithms, and we performed an extensive experimental evaluation. The experimental results confirm the dependence of `PartialMJ` on intermediate results and the superiority of `PartialPathStack`.

Paper outline. The next section discusses related work. Section 3 presents the XML data model and our language for partial path queries. Section 4 addresses query processing issues. In Section 5, we present our two algorithms. Section 6 shows the experimental results. We conclude and discuss future work in Section 7.

2. RELATED WORK

Previous papers focus on finding matches of binary structural relationships (a.k.a. structural joins). In [24], the authors presented the Multi-Predicate Merge Join algorithm (MPMGJN) for finding such matches. Al-Khalifa et al. [1] introduced a family of stack-based join algorithms, which are more efficient compared to MPMGJN, as they do not require multiple traversals of the XML tree. Algorithms for structural join order optimization were introduced in [22]. Structural join techniques can be further improved using various types of indexes [6, 13, 23].

One can exploit the above techniques to evaluate a path-pattern query or a tree-pattern query. The task involves the following phases: decomposing the query into binary structural relationships, then, finding their matches, and, finally, stitching together these matches. This is inefficient due to the large number of intermediate results. To deal with this problem, Bruno et al. [2] presented two stack-based join algorithms (`PathStack` and `TwigStack`) for the evaluation of path-pattern queries and tree-pattern queries, respectively. `PathStack` is optimal for path-pattern queries, while `TwigStack` is optimal for tree-pattern queries without child relationships.

Several researchers have worked on extending `TwigStack`. For example, in [16], algorithm `TwigStackList` evaluates efficiently tree-pattern queries in the presence of child relationships. Also, in [4], algorithm `Twig2Stack` can evaluate generalized tree-pattern queries including optional relationships. Chen et al. [3] proposed algorithms that handle queries over dag-structured data. Evaluation methods of tree-pattern queries with OR predicates are developed in [12]. In [14], the XR-tree index [13] is used to avoid processing input that does not participate in the answer of the query. Finally, [10] introduces algorithm `TwigOptimal`, which applies the notion of virtual cursors [23] to enhance the traversal of the XML tree during tree-pattern query evaluation.

All the above tree-pattern query evaluation techniques assume that there are access mechanisms, i.e., indexes, that efficiently return a stream of nodes in the XML tree that satisfy a given node predicate. Nodes within streams are usually represented by their positional representation [24] (see Section 3.1). Other types of streaming, e.g., Tag+Level Streaming and Prefix-Path Streaming, are suggested in [5]. In [17], instead of the region encoding positional representation, the authors used an extended Dewey labelling scheme to facilitate query evaluation.

Partially specified tree-pattern queries were introduced in [20, 21]. In these papers, partial tree-pattern queries are evaluated by generating a set of complete tree-pattern queries based on index graphs (structural summaries of data). Here, we focus on the evaluation of partial path-pattern queries. These queries are dags in their canonical form. To the best of our knowledge, no previous holistic algorithms exist for their evaluation.

3. DATA MODEL AND QUERY LANGUAGE

In this section, we discuss about the XML data model and the *region encoding* positional representation technique, and we introduce the partial path query language.

3.1 XML Data

An XML database is commonly modelled by a tree structure. Tree nodes represent and are labelled by elements, attributes, or values. Tree edges represent element-subelement, element-attribute, and element-value relationships.

Without loss of generality, we assume that the root node of an XML tree represents an element labelled by r, and no other node is labelled by r. Such a root node can always be added to a tree if not initially there.

Figure 1 shows an XML tree. The triplets next to the

nodes encode their position in the tree, and they are explained below.

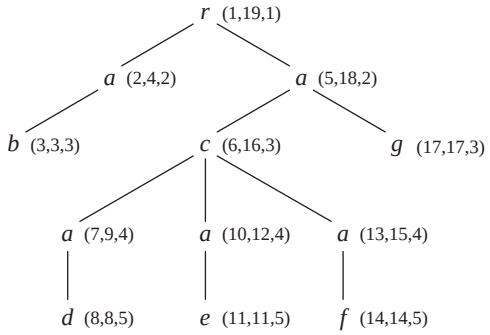


Figure 1: XML tree

Positional representation. XML query processing algorithms require an efficient technique for representing the position of nodes in an XML tree. A commonly used technique is the so called *region encoding* [8, 24, 1, 2, 14], where tree nodes are represented by triplets of the form (*begin*, *end*, *level*).

The *begin* and *end* values of a node can be determined through a depth-first traversal of the XML tree, by sequentially assigning numbers to the first and the last visit of the node. The *level* value represents the level of the node in the XML tree. For simplicity, we assume that one XML tree is processed at a time. If the database comprises multiple trees, a fourth field, $treeID$, can be used to denote its XML tree in the database.

Region encoding simplifies checking structural relationships between two nodes: node n_1 is an ancestor of node n_2 iff $n_1.begin < n_2.begin$, and $n_2.end < n_1.end$. Node n_1 is the parent of node n_2 iff $n_1.begin < n_2.begin$, $n_2.end < n_1.end$, and $n_1.level = n_2.level - 1$.

3.2 Partial Path Queries

We now introduce the syntax and semantics of partial path queries.

Syntax. A partial path query specifies a path pattern where the structure may not be fully defined.

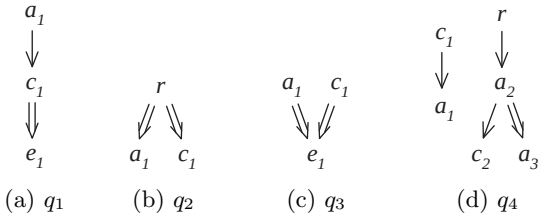
DEFINITION 3.1. Let a_i denote a variable ranging over nodes in an XML tree labelled by a, $a \neq r$. A *structural relationship* is an expression of the form r/a_i, a_i/a_j, or a_i/b_j (*child* relationship), or of the form $r//a_i$, $a_i//a_j$, or $a_i//b_j$ (*descendant* relationship). A *partial path query* is a nonempty set of structural relationships. □

Figure 2 shows four partial path queries.

$$q_1=\{a_1/c_1, c_1//e_1\}$$
$$q_2=\{r//a_1, r//c_1\}$$
$$q_3=\{a_1//e_1, c_1//e_1\}$$
$$q_4=\{c_1/a_1, r/a_2, a_2/c_2, a_2//a_3\}$$

Figure 2: Partial path queries

We can represent a query as a node-labelled graph. The nodes of the graph correspond to the variables of the query. There is a single (resp. double) arrow from node a_i to node b_j iff the structural relationship a_i/b_j (resp. $a_i//b_j$) belongs to the query. Figure 3 shows the graph representation of the queries of Figure 2. Notice that a query graph can be disconnected, e.g. query q_4 in Figure 3(d). In the following, we identify queries with their graph representation.



(a) q_1 (b) q_2 (c) q_3 (d) q_4

Figure 3: Graph representation of queries

The user can flexibly specify the structure of a path in a query fully, partially, or not at all.

Semantics. The answer of a partial path query on an XML tree is a set of tuples. Each tuple consists of tree nodes that lie on the same path and preserve the child and descendant relationships of the query. More formally:

DEFINITION 3.2. An *embedding* of a partial path query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) any node a_i in Q is mapped by M to a node of T labelled by a, and node r in Q is mapped by M to the root of T; (b) the nodes of Q are mapped by M to nodes that lie on the same path in T; (c) $\forall\ a_i/b_j$ (resp. $a_i//b_j$) in Q, $M(b_j)$ is a child (resp. descendant) of $M(a_i)$ in T. □

We call *image* of Q under an embedding M, denoted $M(Q)$, a tuple that comprises all the images of the nodes of Q under M.

DEFINITION 3.3. The *answer* of Q on T is the set of the images of Q under all possible embeddings of Q to T. □

Consider, for instance, query q_3 in Figure 3(c). The answer of q_3 on the XML tree of Figure 1 is: $\{\langle a_1$:(5,18,2), c_1:(6,16,3), e_1:(11, 11,5)$\rangle$, $\langle c_1$:(6,16,3), a_1:(10,12,4), e_1:(11, 11,5)$\rangle\}$.

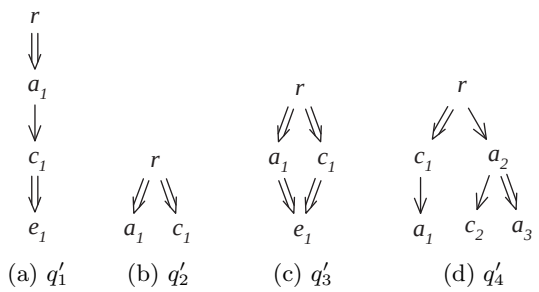
Notice that a query may include two distinct nodes a_i and a_j, e.g., c_1 and c_2 in query q_4 in Figure 3(d). The images of two such nodes under an embedding may coincide unless this is prevented from the structural relationships of the query.

Query q_1 is the only partial path query in Figure 3 which is also a mere path query, since the structural relationships in the query induce a total order for the query nodes. Query q_2 is syntactically similar to a tree-pattern query (twig). However, the semantics is different: when query q_2 is a partial path query, the images of the query nodes a_1 and c_1 should lie on the same path on the XML tree.

A partial path query may contain more than one source node (i.e. a node without incoming edges). Since, by assumption, every XML tree is rooted at a node labelled by r, we can add a node r (if not already there) and double arrows to any source node of a query without altering its meaning. This way, every query can be represented as a rooted directed graph. Figure 4 shows the four queries of Figure 3 after this transformation.

4. QUERY PROCESSING

As the structure of the path is partially specified in partial path queries, new structural relationships may be inferred



(a) q'_1 (b) q'_2 (c) q'_3 (d) q'_4

Figure 4: Queries with root node

from those explicitly specified in a query. Further, unlike path queries, partial path queries may be unsatisfiable and have redundant nodes. Derived structural relationships are necessary in detecting unsatisfiable queries and redundant nodes. In this section, we address these issues and we show how a query can be processed and put in a canonical form, which is convenient for evaluation.

4.1 Structural Relationship Inference

Consider query q'_4 in Figure 4. Since a_2 is a parent of c_2 and an ancestor of a_3, we can infer that c_2 is an ancestor of a_3 as well. Indeed, since c_2 is a child of a_2, a_3 can not be placed between a_2 and c_2. Next we formalize the inference of structural relationships.

DEFINITION 4.1. A structural relationship p is *derived* from a query Q iff for every embedding M of Q to any XML tree, M satisfies p. The *closure* of Q is the set that comprises all the structural relationships that can be derived from Q. □

In order to characterize the derivation of structural relationships and compute closures of queries, we introduce a set of inference rules shown in Figure 5. Let a_i and b_j be query nodes, and x, y, z, and w be variables ranging over query nodes. Recall that r denotes the root node of a query. We use the symbol $\vdash$ to denote that the relationships that precede it infer the relationship that follows it. The absence of expressions that precede $\vdash$ denotes an axiom.

(IR1) $\vdash r//a_i$
(IR2) $x/y \vdash x//y$
(IR3) $x//y,\ y//z \vdash x//z$
(IR4) $x/a_i,\ x//b_j, \vdash a_i//b_j$
(IR5) $a_i/x,\ b_j//x, \vdash b_j//a_i$
(IR6) $x/y,\ y/w,\ x//z,\ z//w \vdash x/z$
(IR7) $x/y,\ x//z,\ w/z,\ w//y \vdash x/z$
(IR8) $x/y,\ y/w,\ x/z \vdash z/w$
(IR9) $x//y,\ y//w,\ x/z \vdash z//w$
(IR10) $x/y,\ x/z,\ w/z \vdash w/y$
(IR11) $x//y,\ x/z,\ w//z \vdash w//y$
(IR12) $x/y,\ y/w,\ z/w \vdash x/z$
(IR13) $x//y,\ y//w,\ z/w \vdash x//z$

Figure 5: Inference rules

The next theorem states that the inference rules correctly and completely characterize the derivation of structural relationships. Let Q be a query, and p be a structural relationship not in Q. A set of inference rules is *sound* if whenever p can be produced from Q using the inference rules, p can also be derived from Q. It is *complete* if whenever p can be derived from Q, p appears in Q or can be produced from Q using the inference rules.

THEOREM 4.1. The set of inference rules of Figure 5 is sound and complete. □

Based on the closure of a query, we define the *full form* of a query.

DEFINITION 4.2. A query is in *full form* if it is equal to its closure. □

Clearly, the number of structural relationships in the closure of a query is, in the worst case, a square polynomial in the number of its nodes. In practice, only a small percentage of these relationships appears in the closure of the query. Since usually a query is much smaller than the data, the cost of computing its closure is insignificant.

4.2 Query Satisfiability

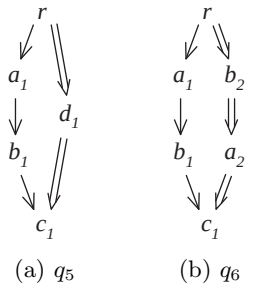
Detecting an unsatisfiable query saves execution time at a small overhead. It prevents accessing the data to get an empty answer.

DEFINITION 4.3. A partial path query is called *satisfiable* iff it has a non-empty answer on some XML tree. Otherwise, it is called *unsatisfiable*. □

In contrast to path queries, partial path queries can be unsatisfiable. Consider, for instance, the query q_5 of Figure 6(a). Clearly, this query is unsatisfiable since no XML tree path can satisfy all four structural relationships in it. The following proposition provides necessary and sufficient conditions for query satisfiability.

THEOREM 4.2. A partial path query is unsatisfiable iff its full form comprises a trivial cycle, i.e. two structural relationships of the form $a//b$ and $b//a$.

Consider the queries q_5 and q_6 of Figure 6. These queries are unsatisfiable. One can see that the full form of both queries comprises trivial cycles. For instance, they both comprise the trivial cycle $r//a_1$ and $a_1//r$.



(a) q_5 (b) q_6

Figure 6: Two unsatisfiable queries

Checking query satisfiability amounts to checking the full form of the query for trivial cycles. This is in the worst case a square polynomial in the number of the query nodes. Given that the size of a query is not expected to be comparable to the size of the XML database, the cost of checking query satisfiability is insignificant.

4.3 Redundant Nodes in Queries

Some nodes in a query can be removed without affecting the meaning of the query. We call these nodes redundant:

DEFINITION 4.4. A node in a partial path query is *redundant* iff in any tuple of any answer of the query it has the same value as another (not necessarily the same) node of the query. □

Redundant nodes can be detected based on the following theorem:

THEOREM 4.3. A node w in a partial path query is *redundant* iff the full form of the query comprises one of the following sets of structural relationships:

(a) x/w and x/y, where x and y are query nodes and w and y have the same label.
(b) w/x and y/x, where x and y are query nodes and w and y have the same label.
(c) x/y^1, y^1/y^2, ..., y^k/z, $x//w$, $w//z$, $k \geq 1$, where x, y^1, ... y^k, z, and w are query nodes and the label of w is the same as the label of one of y^1, ..., y^k. □

Figures 7(a), 7(b), and 7(c) graphically display the three conditions of Theorem 4.3.

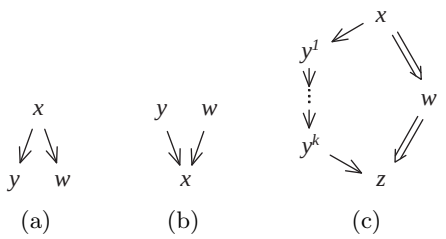


Figure 7: Query patterns with redundant node w

Clearly, identifying redundant nodes in a query can be performed efficiently.

4.4 Canonical Form of Queries

For query evaluation purposes, it is convenient to introduce a "normal form" for queries called *canonical form*.

DEFINITION 4.5. A partial path query Q is in *canonical form* iff its set $\mathcal{P}$ of structural relationships contains exactly all the structural relationships of the closure of $\mathcal{P}$ except those that can be inferred by $\mathcal{P}$ using inference rules IR2 and IR3. □

Since a satisfiable query does not comprise cycles in its full form, it has a unique canonical form. This canonical form can be represented as a rooted directed acyclic graph.

Queries q'_1, q'_2, and q'_3 of Figure 3 are already in canonical form. Figure 8(a) repeats query q'_4 of Figure 3 and Figure 8(b) shows its canonical form.

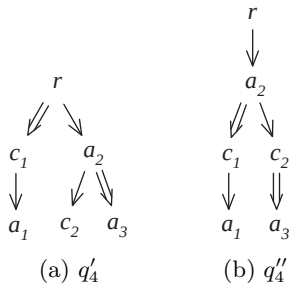


Figure 8: Query q''_4 is the canonical form of q'_4

Computing a canonical form for a query can be done efficiently by removing from its full form edges in any order that can be inferred from other edges using IR2 or IR3, until no more edges can be removed.

In the following, we assume that queries are satisfiable, in canonical form, without redundant nodes. A notable feature of this representation is that there is a topological ordering of the nodes of a query that satisfies its structural relationships (both child and descendant). We exploit this feature in the next section in designing the PartialPathStack algorithm.

5. PARTIAL PATH QUERY EVALUATION ALGORITHMS

In this section, we present two stack-based algorithms for the evaluation of partial path queries: `PartialMJ` and `PartialPathStack`.

5.1 Preliminaries

Let q be a partial path query in canonical form and n be a node in q. Function `nodes(`q`)` returns all nodes of q. Function `isRoot(`n`)` returns true if n does not have incoming edges in q, and false otherwise. Function `isSink(`n`)` returns true if n does not have outgoing edges in q, and false otherwise. Function `parents(`n`)` returns all nodes in q with outgoing edges to n.

Each query node n labelled by l is associated with a stream T_n of all nodes (positional representation) labelled by l in the XML tree. To sequentially access the nodes in T_n, we maintain a cursor C_n, initially pointing to the first node in T_n. For simplicity, C_n may alternatively refer to the node pointed by pointer C_n in T_n. Operation `advance(`C_n`)` moves C_n to the next node in T_n. Function `eos(`C_n`)` returns true if C_n has reached the end of T_n. C_n`.begin` denotes the *begin* field in the positional representation of node C_n (see Section 3.1).

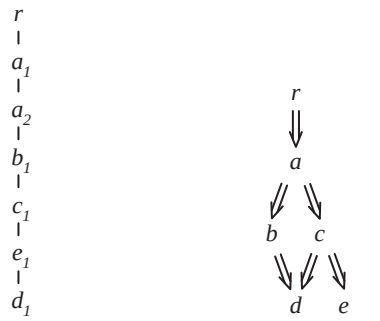
A stack S_n is associated with each query node n. In the case of algorithm `PartialMJ`, each entry of S_n is a pair of a node from stream T_n and a pointer to an entry in the stack of a parent of n in the query. In the case of algorithm `PartialPathStack`, each entry of S_n is a pair of a node from stream T_n and a set of pointers to entries in the stacks of all the parents of n in the query.

Function `empty(`S_n`)` returns true if stack S_n is empty, and false otherwise. Operation `push(`S_n`,`*entry*`)` pushes *entry* on top of stack S_n. Operation `pop(`S_n`)` pops the top entry from stack S_n. Functions `bottom(`S_n`)` and `top(`S_n`)` return the position of the bottom and top entry in stack S_n, respectively. At every point during the execution of the algorithms (a) each node in a stack entry is a descendant in the XML tree of all nodes in the entries below it, and (b) all nodes in a stack lie on the same root-to-leaf path in the XML tree.

5.2 Algorithm PartialMJ

Given a partial path query, algorithm `PartialMJ` extracts a spanning tree of the query graph. Then, it finds matches for all root-to-leaf paths of the spanning tree in the XML tree by using an extension of the path matching algorithm `PathStack` [2]. The results for each path of the spanning tree are tuples produced in a sorted root-to-leaf order in the XML tree. These tuples are merge-joined by guaranteeing that (a) they lie on the same path in the XML tree, and (b) they satisfy the structural relationships that appear in the query graph and not in the spanning tree.

Figure 9(b) shows the graph of a query q and Figure 11(a) shows a spanning tree q_s of q. Edge $c//d$ of q is missing from q_s. Any two results from the two root-to-leaf paths of q_s that are on the same path of the XML tree can be merged to produce a result for q if they satisfy the identity conditions on r and a and the structural condition $c//d$ (see Figure 11(b)).



(a) XML Tree (b) Query q

Figure 9: Example of a tree path and a query

Algorithm `PartialMJ` is shown in Figure 10. In this algorithm, each entry of a stack S_n is a pair of (a) a node from stream T_n and (b) a pointer ptr to the entry of its lowest ancestor in the XML tree appearing in S_m, where m is the parent of n in the spanning tree of the query. Function `isLeaf(`n`)` returns true if n is a leaf node in q_s, and false otherwise.

In lines 01-08, the algorithm scans the streams, and finds matches for the root-to-leaf paths in the spanning tree of the query. Line 02 determines the next query node n to be processed. Line 03 pops out of the stacks all nodes that do not lie on the same root-to-leaf path in the XML tree as the stream node C_n currently processed. Stream node C_n is pushed on stack S_n only if the stacks of the parents of node n in the query are not empty (lines 04-05). This way, we avoid stacking and processing stream nodes which do not contribute results to the answer. When we push a node C_n on stack S_n, we also add a pointer to the top entry in stack S_m, where m is the parent of n in the spanning tree of the query. Line 06 checks if node n is a leaf in the spanning tree. If this is the case, line 07 calls procedure `showResults` to produce the results for the path of the spanning tree ending to node n. These results are sorted in a root-to-leaf order in the XML tree so that they can be easily merge-joined to compute results for the query. Procedure `showResults` is similar to procedure `showSolutionsWithBlocking` [2], which uses a blocking technique to produce results for a path query sorted in a root-to-leaf order. The results for each path of the spanning tree are merge-joined in line 09. This join involves checking that (a) the results are on the same path in the XML tree, (b) matchings for the common nodes of the paths in the query are identical, and (c) structural relationships in the query that do not appear in the spanning tree are satisfied. All these conditions can be checked in a straightforward way using the positional representation for the nodes in the XML tree.

Figure 11(c) shows the state of the stacks after the evaluation of query q of Figure 9(b) on the single-path XML tree of Figure 9(a). Figure 11(a) shows the spanning tree

q: partial path query
q_s: a spanning tree of q
E: the set of edges in q which do not appear in q_s

Algorithm PartialMJ()
01 while (¬end())
02 n = getNextQueryNode()
03 cleanStacks(C_n)
04 if (isRoot(n) or $\forall\ m \in$ parents(n): ¬empty(S_m))
05 moveToStack(n)
06 if (isLeaf(n))
07 showResults(S_n, top(S_n))
08 advance(C_n)
09 joinPathSolutions()

Function end()
 return $\forall\ n \in$ nodes(q): isSink(n) $\Rightarrow$ eos(C_n)

Function getNextQueryNode()
 return $n \in$ nodes(q) such that C_n.begin is minimal

Procedure cleanStacks(C_n)
01 for m in nodes(q)
02 pop all entries in S_m whose nodes are not ancestors of C_n in the XML tree

Procedure moveToStack(n)
01 ptr = pointer to top of S_m, where m is the parent of n in q_s
02 push(S_n, (C_n, ptr))

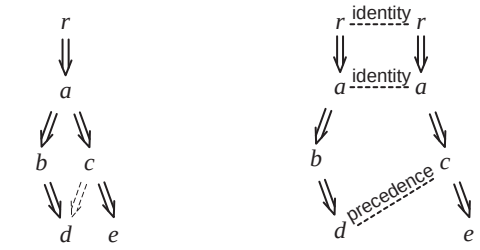
Procedure joinPathSolutions()
01 order the root-to-leaf paths of q_s in descending order of the level of their lowest branching node
02 merge-join the solutions of the root-to-leaf paths of q_s that are on the same path of the XML tree and satisfy the structural relationships in E

Figure 10: Algorithm PartialMJ

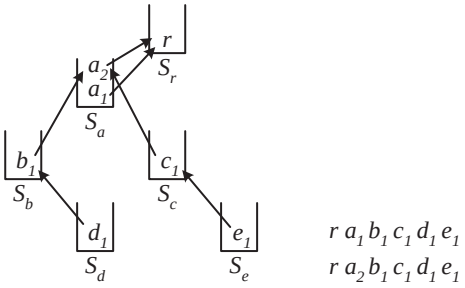
q_s of q used in the evaluation of q. Since the structural relationship $c//d$ of q does not appear in q_s, there are no pointers from stack S_d to stack S_c. The results for the left root-to-leaf path of q_s are $\{ra_1b_1d_1, ra_2b_1d_1\}$, and those for the right root-to-leaf path of q_s are $\{ra_1c_1e_1, ra_2c_1e_1\}$. One can see that from the four possible pairs of results of the two paths only two can be merge-joined, and are shown in Figure 11(d).

Child relationships. The algorithm presented in Figure 10 is designed for the evaluation of queries that do not include child relationships. In the presence of child relationships, two changes need to be done. First, whenever a node C_b from stream T_b is processed, and a/b is a child relationship in the query, C_b is pushed on stack S_b only if its parent node in the XML tree appears in (the top position of) stack S_a. Second, in the computation of the results of a root-to-leaf path of the spanning tree of the query (using procedure `showResults`), a node in stack S_b appears only in results for this path that also include from S_a its parent in the XML tree.

Analysis of PartialMJ. Algorithm `PartialMJ` fills the stacks in a single pass of the input streams. Further, it uses



(a) Spanning tree q_s (b) Join conditions on the spanning tree q_s



(c) Stacks

$r\ a_1\ b_1\ c_1\ d_1\ e_1$

$r\ a_2\ b_1\ c_1\ d_1\ e_1$

(d) Results

Figure 11: PartialMJ example

procedure `showResults` to produce results for every root-to-leaf path in the spanning tree of the query. This procedure is shown to be asymptotically optimal for the evaluation of path queries [2]. However, there may be combinations of results from the root-to-leaf paths in the spanning tree that cannot be merged to form a result of the query. We call these combinations *intermediate results*. Because of the intermediate results, the algorithm is not asymptotically optimal. Clearly, if the query is a path-pattern query, algorithm `PartialMJ` is asymptotically optimal.

5.3 Algorithm PartialPathStack

To overcome the problem of intermediate results of `PartialMJ`, we developed a novel holistic stack-based algorithm for the evaluation of partial path queries. In contrast to `PartialMJ`, `PartialPathStack` does not decompose a query into paths, but tries to match the query graph to an XML tree as a whole.

The key feature of algorithm `PartialPathStack` is that it employs a topological order of the query nodes, i.e., a linear ordering of nodes which respects the partial order induced by the structural relationships of the query. Algorithm `PartialPathStack` is shown in Figure 12.

Algorithm `PartialPathStack` manages streams and stacks as `PartialMJ`. The only difference is that, in the case of `PartialPathStack`, each entry in a stack S_n is a pair of a node from stream T_n and a set of pointers to entries in the stacks of all the parents of n in the query.

Whenever a stream node C_n of a query sink node n is pushed on a stack, the algorithm checks whether results can be generated. Output is produced in a reverse topological order, so that the stack of a query node is processed after the stacks of its children nodes in the query have been processed. To avoid redundantly reproducing results, the algorithm outputs at this point only results that include the stream node C_n. Procedure `outputResults` combines nodes from all stacks to produce the query results. No other node from the stack S_n of node n is used at this point to produce new results for the query (lines 07-08). In contrast, all nodes from the other sink node stacks can be used to form results (lines 09-11). All the nodes can be used from non-sink node stacks if they (or nodes higher in the stack) are pointed by nodes in the stacks of all the children of n in the query (lines 12-15).

Figure 13(b) shows the state of the stacks after the evaluation of query q of Figure 9(b) on the single-path XML tree of Figure 9(a). When node d_1 is pushed on stack S_d, new results can be produced that include d_1, which are produced according to the topological order shown in Figure 13(a). The results of the query are shown in Figure 13(c).

To process queries with child relationships, we need to modify the stacking of the nodes and the output of solutions as we did with `PartialMJ`.

Analysis of PartialPathStack. One can observe that (a) each stream node C_n can be pushed on stack S_n only after its ancestors in the XML tree have been considered (procedure `getNextQueryNode`), and (b) a node is not popped from a

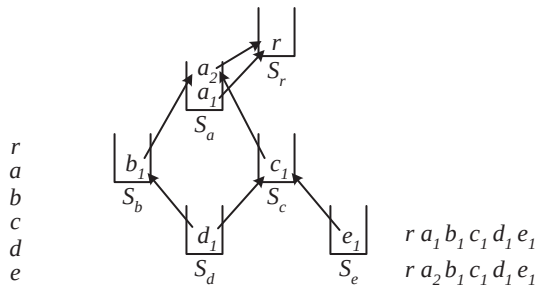
```
q: partial path query with N nodes

Algorithm PartialPathStack()
01 extract a topological order 1..N of the query nodes
02 while (¬end())
03     n = getNextQueryNode()
04     cleanStacks(C_n)
05     if (isRoot(n) or ∀ m ∈ parents(n): ¬empty(S_m))
06       moveToStack(n)
07       if (isSink(n) and ∀ m ∈ nodes(q):
                              isSink(m) ⇒ ¬empty(S_m))
08         if (n == N)
09           outputResults(n, N, top(S_N))
10         else
11           for i = bottom(S_N) to top(S_N)
12             outputResults(n, N, i)
13     advance(C_n)

Procedure moveToStack(n)
01 ptrs = pointers to top of all parents(n) in q
02 push(S_n, (C_n, ptrs))

Procedure outputResults(sinkNode, m, stackPos)
01 solution[m] = stackPos
02 if (m = 1)    //node m is the root of the query
03     output (S_1[solution[1]],...,S_N[solution[N]]
04 else
05     for a in parents(m)
06       S_a.pointerFrom[m] = S_m[stackPos].pointer_to_a
07     if (m-1 == sinkNode)
08       outputResults(sinkNode, m-1, top(S_{m-1}))
09     else if (isSink(m-1))
10       for i = bottom(S_{m-1}) to top(S_{m-1})
11         outputResults(sinkNode, m-1, i)
12     else
13       maxHeight = minarg_d {S_{m-1}.pointerFrom[d]}
14       for i = bottom(S_{m-1}) to maxHeight
15         outputResults(sinkNode, m-1, i)
```

Figure 12: Algorithm PartialPathStack



(a) Topological order (b) Stacks (c) Results

Figure 13: PartialPathStack example

stack unless the current node C_n appears in a different path in the XML tree (procedure `cleanStacks`). Based on these observations, we can show the following lemma:

LEMMA 5.1. The stream nodes in a result of a query appear in the stacks of the `PartialPathStack` algorithm simultaneously at some point during its execution. □

We use the above lemma to show the correctness and completeness of `PartialPathStack`:

THEOREM 5.1. Given a partial path query q and an XML tree T, algorithm `PartialPathStack` correctly returns all the answers of q on T. □

Given a partial path query q and an XML tree T, let *input* denote the sum of sizes of the input streams, *output* denote the size of the results of q on T, *indegree* denote the maximum number of incoming edges to a query node, *outdegree* denote the maximum number of outgoing edges from a query node, and *maxpath* denote the maximum length of a root-to-leaf path in T.

THEOREM 5.2. Algorithm `PartialPathStack` has worst-case I/O and CPU time complexities O(*indegree* * *input* + *outdegree* * *output*). The worst-case space complexity of `PartialPathStack` is O(*indegree* * min(*input*, *maxpath*)).

Based on the previous theorem, `PartialPathStack` is asymptotically optimal if the *indegree* and *outdegree* of the query are bound by a constant. Clearly, for the case of queries whose graph is a tree, only the *outdegree* needs to be bound by a constant for `PartialPathStack` to be asymptotically optimal. In any case, `PartialPathStack` does not generate any intermediate results.

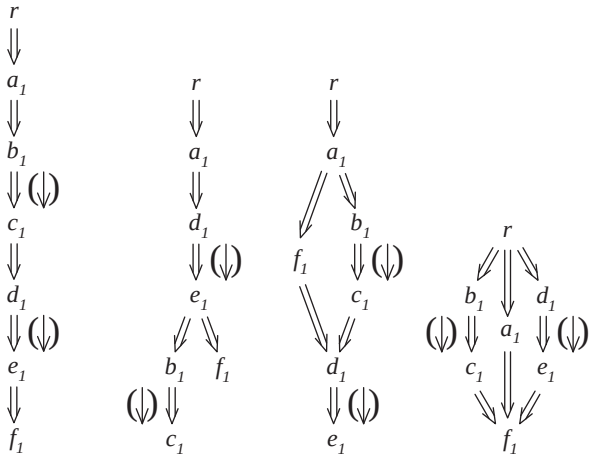
6. EXPERIMENTAL EVALUATION

We ran a comprehensive set of experiments to measure the performance of `PartialMJ` and `PartialPathStack`. In this section, we report on their experimental evaluation.

Setup. We evaluated the performance of the algorithms on both benchmark and synthetic data. For benchmark data, we used the Treebank XML document[1] Treebank's XML tree consists of around 2.5 million nodes having 250 distinct element tags and its maximum depth is 36. It also has deep recursive data. Synthetic data is random XML trees. We generated such trees using IBM's AlphaWorks XML generator[2]. In all the experiments, the parameter $MaxRepeats$ (that determines the maximum number a node appears as a child of its parent node) was set to 4, and the parameter $numLevels$ (that determines the maximum number of tree levels) was set to 14. The number of distinct element tags used in all trees was fixed to 11. For each measurement on synthetic data, 10 different XML trees of the same number of nodes were used. Each displayed value in the plots is the average over these 10 measurements.

Figure 14 shows the types of queries used in our experiments. Queries Q_1 to Q_4 include only descendant relationships, while queries Q_5 to Q_8 include child relationships as well. The labels of the query nodes, however, are appropriately modified so that the queries can always produce results in the different XML trees used in the experiments. Our query set comprises a full spectrum of partial path queries, from simple path-pattern queries to complex non-tree graph queries.

We implemented all algorithms in C++, and ran our experiments on a dedicated Linux PC (AMD Sempron 2600+) with $2GB$ of RAM.



(a) Q_1 (Q_5) (b) Q_2 (Q_6) (c) Q_3 (Q_7) (d) Q_4 (Q_8)

Figure 14: Types of partial path queries used in the experiments.

Execution time on fixed datasets. We measured the execution time of `PartialMJ` and `PartialPathStack` for evaluating all queries in Figure 15 on both Treebank and synthetic data. For queries Q_1 and Q_5, which are path-pattern queries, we also measured the execution time of algorithm `PathStack` [2]. The synthetic XML trees used in this experiment consist of 2.5 million nodes.

Figures 15(a) and 15(b) present the evaluation results. Figure 15(c) shows the number of results obtained per query. Algorithm `PartialPathStack` is more efficient than `PartialMJ`.

Regarding queries Q_1 and Q_5, `PartialPathStack` performs as fast as `PathStack`. This is expected, since `PartialPathStack` reduces to `PathStack` in case of path-pattern queries.

Execution time varying the input size. We measured the execution time of `PartialMJ` and `PartialPathStack` for evaluating queries Q_2, Q_3 and Q_7 of Figure 14 over synthetic XML trees of various sizes. Figures 16, 17 and 18 present the results obtained for XML trees whose node stream sizes

[1] http://www.cis.upenn.edu/ treebank

[2] www.alphaworks.ibm.com/tech/xmlgenerator

vary from 1 to 3 million nodes. Clearly, in every case, `PartialPathStack` is more efficient than `PartialMJ`.

In the experimental evaluation of query Q_2, an increase in the input size results in an increase in the output size (Figure 16(b)). When the input and the output size goes up, the execution time of `PartialMJ` and `PartialPathStack` increases (Figure 16(a)). This confirms the complexity results that show dependency of the execution time on the input and output size. However, the increase in the execution time of `PartialMJ` is slightly sharper than that of `PartialPathStack`. The reason is that `PartialMJ` is also affected by the increase in the number of the intermediate results shown in Figure 16(c). In contrast, `PartialPathStack` is independent of the size of the intermediate results.

In the experimental evaluation of query Q_3, the output size (Figure 17(b)) is comparable to the output size of query Q_2 (Figure 16(b)). The execution time of `PartialPathStack` for the evaluation of Q_3 (Figure 17(a)) is comparable to the execution time for the evaluation of Q_2 (Figure 16(a)). This again confirms the worst-case complexity results. The number of intermediate results in Q_3 (Figure 17(c)) is larger than the number of intermediate results in Q_2 (Figure 16(c)) for all input sizes used in the experiments. This increase is reflected in the execution time of `PartialMJ` which increases sharper than `PartialPathStack`.

Query Q_7 used in the experiment shown in Figure 18 is more restrictive than query Q_3 since it involves two child relationships not present in Q_3. Clearly, the number of intermediate results (Figure 18(c)) and the output size (Figure 18(b)) from the evaluation of Q_7 is less than those of Q_3, and the same holds for the execution time of both algorithms (Figure 18(a)). The reduction is more intense for `PartialMJ` due to the strong decrease in the number of intermediate results. In all cases, `PartialPathStack` largely outperforms `PartialMJ`.

7. CONCLUSION

We defined a partial path-pattern query language which represents a class of XPath expressions, and is useful for querying multiple XML data sources with unknown or different structures. We studied the problem of efficiently evaluating partial path-pattern queries. In order to process partial path-pattern queries, we introduced a set of sound and complete inference rules to characterize structural relationship derivation, we provided necessary and sufficient conditions for detecting query unsatisfiability and node redundancy, and we showed partial path-pattern queries can be equivalently put in a canonical dag form. We developed two stack-based algorithms for the evaluation of partial path-pattern queries, `PartialMJ` and `PartialPathStack`. `PartialMJ` evaluates a query dag by decomposing it while `PartialPathStack` is a holistic one. An analysis and experimental evaluation showed that `PartialPathStack` is independent of intermediate results and largely outperforms `PartialMJ`. We plan to extend out work, studying partial tree-pattern queries and developing techniques for their evaluation.

8. REFERENCES

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE*, 2002.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. of ACM SIGMOD*, 2002.

[3] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB*, 2005.

[4] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. of VLDB*, 2006.

[5] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. of SIGMOD*, 2005.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, 2002.

[7] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A semantic search engine for XML. In *Proc. of VLDB*, 2003.

[8] M. P. Consens and T. Milo. Algebras for querying text regions. In *Proc. of ACM PODS*, 1995.

[9] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into xml query processing. *Computer Networks*, 2000.

[10] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *Proc. of CIKM*, 2005.

[11] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *Proc. of ICDE*, 2003.

[12] H. Jiang, H. Lu, and W. Wang. Efficient processing of xml twig queries with or-predicates. In *Proc. of ACM SIGMOD*, 2004.

[13] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *Proc. of ICDE*, 2003.

[14] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *Proc. of VLDB*, 2003.

[15] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *Proc. of VLDB*, 2004.

[16] J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: A look-ahead approach. In *Proc. of CIKM*, 2004.

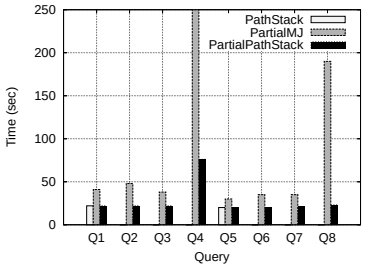
[17] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *Proc. of VLDB*, 2005.

[18] D. Olteanu. Forward node-selecting queries over trees. *ACM Trans. Database Syst.*, 2007.

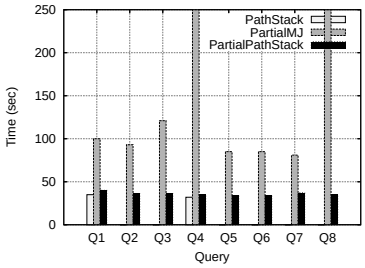
[19] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *Proc. of the XMLDM, MDDE, and YRWS Workshops*, 2002.

[20] D. Theodoratos, T. Dalamagas, A. Koufopoulos, and N. Gehani. Semantic querying of tree-structured data sources using partially specified tree patterns. In *Proc. of ACM CIKM*, 2005.

[21] D. Theodoratos, S. Souldatos, T. Dalamagas, P. Placek, and T. Sellis. Heuristic containment check of partial tree-pattern queries in the presence of index graphs. In *Proc. of ACM CIKM*, 2006.



(a) Execution time (Treebank)

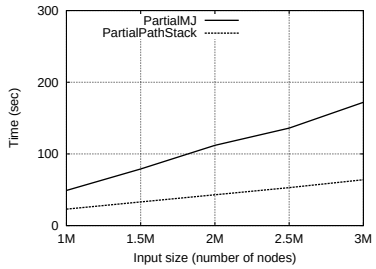


(b) Execution time (Synthetic data)

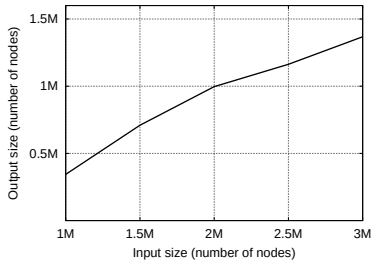
Query	Treebank	Synth. data
Q_1	86360	251112
Q_2	103313	145591
Q_3	8	149006
Q_4	2883598	1178937
Q_5	13714	130242
Q_6	12658	73065
Q_7	2	66179
Q_8	44233	443691

(c) Num. of results per query

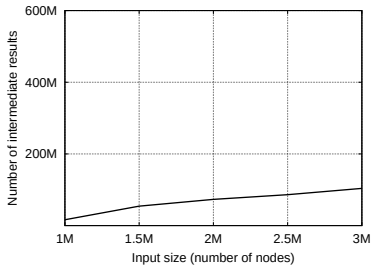
Figure 15: `PartialMJ` **vs** `PartialPathStack` **for fixed data sets.**



(a) Execution time

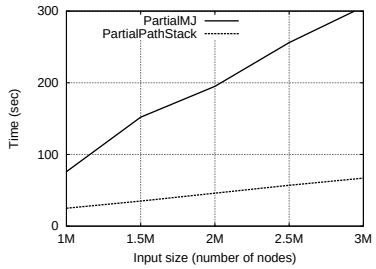


(b) Number of results

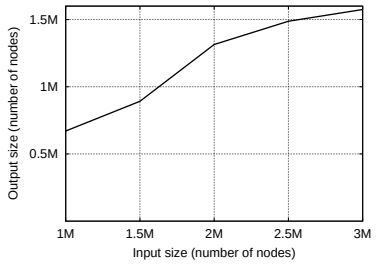


(c) Number of intermediate results

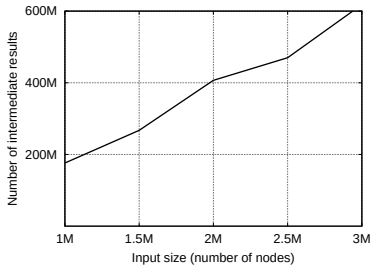
Figure 16: `PartialMJ` **vs** `PartialPathStack` **for** Q_2**, varying the size of the XML tree.**



(a) Execution time

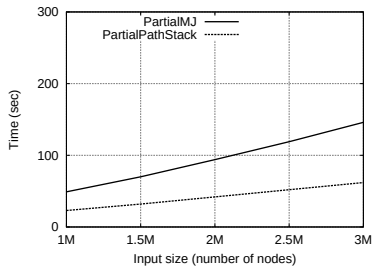


(b) Number of results

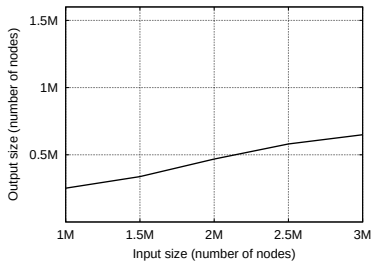


(c) Number of intermediate results

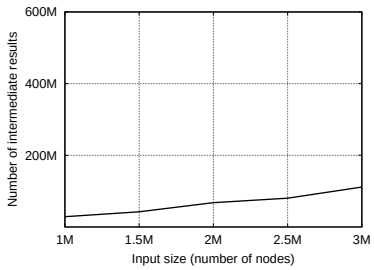
Figure 17: `PartialMJ` **vs** `PartialPathStack` **for** Q_3**, varying the size of the XML tree.**



(a) Execution time



(b) Number of results



(c) Number of intermediate results

Figure 18: `PartialMJ` **vs** `PartialPathStack` **for** Q_7**, varying the size of the XML tree.**

[22] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proc. of ICDE*, 2003.

[23] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. Beyer. Virtual cursors for XML joins. In *Proc. of ICDE*, 2004.

[24] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of ACM SIGMOD*, 2001.