Efficient Processing of XML Twig Queries with OR-Predicates[*]

Haifeng Jiang
jianghf@cs.ust.hk

Hongjun Lu
luhj@cs.ust.hk

Wei Wang
fervvac@cs.ust.hk

Department of Computer Science
The Hong Kong University of Science and Technology
Hong Kong, China

ABSTRACT

An XML twig query, represented as a labeled tree, is essentially a complex selection predicate on both structure and content of an XML document. Twig query matching has been identified as a core operation in querying tree-structured XML data. A number of algorithms have been proposed recently to process a twig query holistically. Those algorithms, however, only deal with twig queries without OR-predicates. A straightforward approach that first decomposes a twig query with OR-predicates into multiple twig queries without OR-predicates and then combines their results is obviously not optimal in most cases. In this paper, we study novel holistic-processing algorithms for twig queries with OR-predicates without decomposition. In particular, we present a merge-based algorithm for sorted XML data and an index-based algorithm for indexed XML data. We show that holistic processing is much more efficient than the decomposition approach. Furthermore, we show that using indexes can significantly improve the performance for matching twig queries with OR-predicates, especially when the queries have large inputs but relatively small outputs.

1. INTRODUCTION

Matching twig queries is a core operation in XQuery processing. A few algorithms have recently been proposed for matching such labeled twigs. Among them, the *holistic twig join* algorithms [3, 11] have demonstrated superior performance due to their effectiveness in dampening irrelevant intermediate results and their capability to leverage indexes to minimize irrelevant data access.

Surprisingly, we found that almost all the existing work on twig query matching only considered twig queries whose sibling edges are connected by `AND` logic, such as

```
Q1 = /dblp/paper[title='XML' and
        year=2003]//author
```

which asks for *the authors of papers with title 'XML' and published in year 2003*[1]. Queries in real applications, however, may contain logical-OR operators, such as

```
Q2 = /dblp/paper[title='XML' or
        year=2003]//author
```

which selects *the authors who have papers either with title 'XML' or published in year 2003.* In general, logical-AND and logical-OR operators can be arbitrarily specified in an XQuery expression. For example, the following query

```
Q3 = /dblp/paper[title='XML' or
        (year=2003 and conf='SIGMOD')]//author
```

selects *the authors who have papers either with title 'XML' or published in SIGMOD 2003.*

We call such general twig queries as `AND`/`OR`-twig queries and denote twig queries without logical-OR operators as `AND`-twig queries.

To handle a twig query with logical-OR operators, naïvely, we can decompose it into multiple `AND`-twigs, process each `AND`-twig with some existing algorithm and then combine all the results. For example, we can evaluate query Q2 as two separate `AND`-twigs:

```
/dblp/paper[title='XML']//author
/dblp/paper[year=2003]//author
```

Although existing twig join algorithms are applied, such a decomposition-based approach has a serious disadvantage: we may scan same data multiple times, incurring more I/O and CPU cost. For example, to evaluate the two resultant `AND`-twigs for Q2, typically, we need to access the data corresponding to `dblp`, `paper` and `author` elements twice. The decomposition process is analogous to transforming an arbitrary logical expression into a logical expression in disjunctive normal form. In the worst case, the number of resultant `AND`-twigs from decomposition is exponential to the size of the twig query. While some optimization techniques may be applied, it is inevitable that certain data have to be scanned multiple times.

Motivated by the recent success in efficient holistic processing of `AND`-twigs, we present in this paper the techniques

---

[*] This work is partially supported by the Research Grant Council of the Hong Kong Special Administrative Region, China (grant AoE/E-01/99).

[1] Here, we assume that the output only contains results for the last tag in the main path, for ease of exposition. Our algorithms to be presented can output twig instances.

SIGMOD 2004 June 13-18, 2004, Paris, France.

developed to process `AND/OR`-twigs holistically without decomposing them into `AND`-twigs. The contributions of the work reported here can be summarized as follows:

- We develop a basic framework for holistic processing of `AND/OR`-twigs based on the concept of **OR-block**. With OR-blocks, an `AND/OR`-twig can be viewed as an `AND`-twig containing element nodes and OR-blocks. As a result, efficient holistic algorithms for `AND`-twigs can be leveraged.
- Novel algorithms are developed to efficiently evaluate OR-blocks, hence `AND/OR`-twig queries for XML data that are either sorted or indexed. Both the analytical and experimental results demonstrate the effectiveness and efficiency of our techniques.

The remainder of the paper is organized as follows. Section 2 gives some preliminary knowledge on twig query processing. The concept of OR-block and its role in holistic `AND/OR`-twig query processing are described in Section 3. Section 4 and Section 5 present, respectively, the merge-based and the index-based algorithms for matching `AND/OR`-twigs holistically. Section 6 presents the performance study. Some related work is presented in Section 7. Finally, Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Data Model

We model XML documents as ordered trees. Figure 1 shows an example XML data tree. Each tree node is assigned a region code (*start*, *end*, *level*) based on its position in the data tree [21, 18, 4, 10]. Each text phrase is enclosed in a rectangle and assigned a region code that has the same *start* and *end* values.

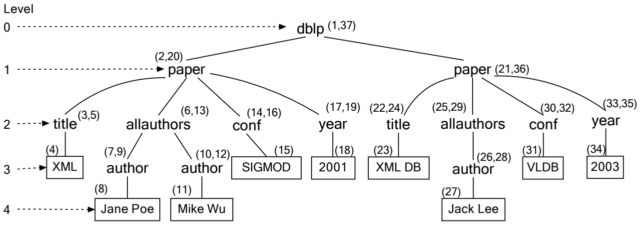


Figure 1: An example XML data tree with region codes

The region encoding supports efficient evaluation of structural relationships (i.e., ancestor-descendant or parent-child relationship) between element nodes. Formally, element u is an ancestor of element v if and only if $u.start < v.start < u.end$. For parent-child relationship, we also test whether $u.level = v.level - 1$. Based on the property of region encoding, we have COROLLARY 1:

COROLLARY 1. *Given two elements* e_i *and* e_j*, if* $e_i.end < e_j.start$*, then* e_i *is neither an ancestor of element* e_j *nor an ancestor of any element* e_x *such that* $e_x.start > e_j.start$.

2.2 Tree Representation for AND/OR-Twigs

We represent an `AND/OR`-twig query as a tree with three types of nodes: location step query node (`QNode`), logical-AND node (`ANode`) and logical-OR node (`ONode`):

- `QNode`: A location step query node in the tree stands for one location step in the original twig query. A `QNode` has the content */tag* or *//tag*, where '/' denotes a *child* location step axis, '//' denotes a *descendant* location step axis, and '*tag*' is a placeholder for the node test (i.e., the corresponding label in the twig query).
- `ANode`: A logical-AND node always takes the text '`and`' in the query tree. It connects two or more child subtrees with `AND` logic.
- `ONode`: A logical-OR node always takes the text '`or`' in the query tree. It connects two or more child subtrees with `OR` logic.

The first three trees in Figure 2 are the tree representations for queries Q1, Q2 and Q3 respectively. Each tree node is identified as n_i. An `ANode` or an `ONode` is enclosed with a rectangle.

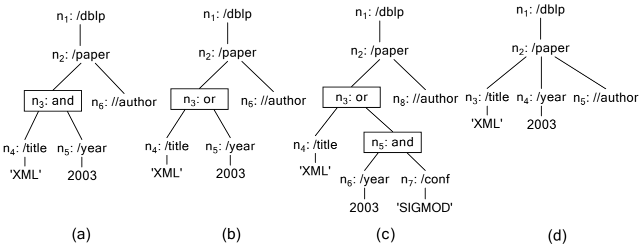


Figure 2: (a), (b) and (c) are the query trees for queries Q1, Q2 and Q3 respectively. (d) is the simplified query tree of (a).

We call a `QNode` as an **ancestor-descendant** `QNode` if its location step axis is '//' or as a **parent-child** `QNode` if its location step axis is '/'. In Figure 2(a), n_6 is an ancestor-descendant `QNode` (i.e., an ancestor-descendant relationship between n_2 and n_6 is specified) and n_4 is a parent-child `QNode` (i.e., a parent-child relationship between n_2 and n_4 is specified).

2.2.1 Query Tree Simplification

A query tree may contain redundant nodes. For example, we can simplify the query tree (a) in Figure 2 to (d), which is semantically the same as (a) but has one tree node less. We define two simplification rules that are of particular interest to this work: (1) If an `ANode` or an `ONode` n has a child node n_i of the same type, we can remove n_i and link the child nodes of n_i to n; and (2) If a `QNode` n has a child `ANode` n_i, we can remove n_i and link the child nodes of n_i to n. The simplification from Figure 2(a) to Figure 2(d) is based on rule (2). From now on, we assume that all query trees are simplified with these two rules.

It is worth noticing that there are other rules for simplifying twig queries. We refer the interested reader to the work by Amer-Yahia *et al* on minimization of twig queries [2].

2.2.2 Operations on Query Tree Nodes

Given a query tree Q, we will use q (and its variants such as q_i and q') to denote a `QNode` in Q or the subtree rooted at q when there is no ambiguity, and use n (and its variants such as n_i and n') to refer to a node of any type in Q.

We define some operations on query tree nodes. `children`(n) returns all child nodes of n and `parent`(n) returns the parent

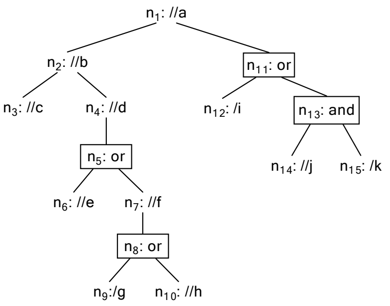


Figure 3: An example query tree

node of n. Given the query tree in Figure 3, `children`(n_1) returns $\{n_2, n_{11}\}$ and `parent`(n_{12}) is n_{11}. `Qchildren`(n) stands for the set of `QNode`s in subtree n that are reachable from n without traversing other `QNode`s, and `Qparent`(n) returns the nearest ancestor `QNode` of n. For the query tree in Figure 3, `Qchildren`(n_1) is $\{n_2, n_{12}, n_{14}, n_{15}\}$, `Qchildren`($n_4$) is $\{n_6, n_7\}$ and `Qparent`(n_{12}) is n_1.

3. MATCHING AND/OR-Twigs

We will use the following convention in this section: each `QNode` q_i (or n_i) is associated with an element node e_i (by changing 'q' or 'n' to 'e') such that $tag(e_i) = tag(q_i)$. In addition to the convention, we define a function, namely `edgeTest` which is ubiquitously used throughout the text:

Definition 1. [**edgeTest(q)** or **edgeTest**(e', e)] Let q be a `QNode` in an `AND/OR`-twig and q' be `Qparent`(q)—recall that, by convention e and e' are the associated elements of q and q' respectively. Boolean function `edgeTest`(q) or `edgeTest`(e', e) evaluates `true` if element e' is an ancestor (respectively, the parent) of element e if q is an ancestor-descendant (respectively, a parent-child) `QNode`.

Section 3.1, together with Section 3.2, gives a precise definition of a match for an `AND/OR`-twig query, based on the concept of OR-block. In preparation for presenting our algorithms, we further study the properties of an OR-block in Section 3.3.

3.1 AND/OR-Twig Matching

Before we give a formal definition of a match for an `AND/OR`-twig query, we identify a unique construct in an `AND/OR`-twig, OR-predicate:

Definition 2. [**OR-predicate**] Given a query tree Q, an OR-predicate is a subtree in Q such that the root of the subtree is an `ONode` n and `parent`(n) is a `QNode`.

The query tree in Figure 3 has three OR-predicates, rooted at n_5, n_8 and n_{11}. In particular, OR-predicate n_5 contains OR-predicate n_8.

Given the query in Figure 3, according to its semantics, we would say that element e_1 has a match for subtree n_1 if the following three conditions are met: (1) e_1 satisfies OR-predicate n_{11} (see, Section 3.2); (2) `edgeTest`(n_2) is `true`; and (3) e_2 has a match for subtree n_2. A match for an `AND/OR`-twig can be formally expressed as follows:

Definition 3. [**A Match for an `AND/OR`-twig query**] Let Q be a query tree with N nodes $n_1, n_2, \cdots, n_N$, where n_1 is the root `QNode`. By convention, e_i is the associated element of n_i if n_i is a `QNode`. We say element e_1 has a match for the query tree n_1 if the following holds for each child subtree n_{k_i} of n_1: if n_{k_i} is an `ONode`, then e_1 satisfies OR-predicate n_{k_i} (see, Section 3.2); otherwise (i.e., n_{k_i} is a `QNode`), `edgeTest`(n_{k_i}) is `true` and, element e_{k_i} has a match for subtree n_{k_i} if n_{k_i} is not a leaf node.

3.2 OR-predicate Evaluation

The challenge to OR-predicate evaluation is that, for an OR-predicate to be `true`, not all its components are required to be `true`. For the query tree in Figure 3, element e_1 satisfies OR-predicate n_{11} if either `edgeTest`(n_{12}) is `true` or both `edgeTest`(n_{14}) and `edgeTest`(n_{15}) are `true`. In a nutshell, to evaluate an OR-predicate, we consider the logical combination of the `edgeTest`(q_i) values for all `QNode`s q_i in the OR-predicate.

We introduce a new concept, namely OR-block, which is important to understanding OR-predicate evaluation.

Definition 4. [**OR-block**] Given a query tree Q, an OR-block is a tree t embedded in Q such that the root of t is an `ONode` n, `parent`(n) is a `QNode` and the leaf nodes of t are `Qchildren`(n).

In Figure 3, there are three OR-blocks rooted at n_5, n_8 and n_{11}. `QNode`s n_6 and n_7 are the leaf nodes of OR-block n_5. OR-block n_8 also has two leaf nodes, n_9 and n_{10}. OR-block n_{11} has three leaf nodes (i.e., n_{12}, n_{14} and n_{15}) and one internal node n_{13}. Different from OR-predicates, OR-blocks in a query tree are disjoint.

If we regard an OR-block as a *composite* tree node, an `AND/OR`-twig can be represented as a query tree with only `QNode`s and OR-blocks. Figure 4 shows such a representation of the query tree in Figure 3.

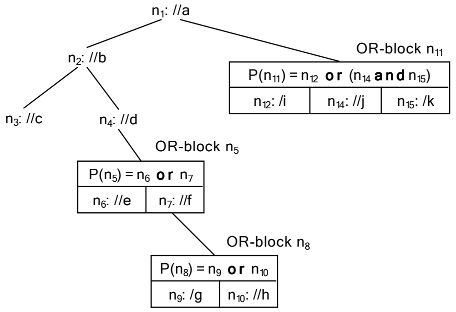


Figure 4: A representation of the query tree in Figure 3 by `QNode`s and OR-blocks

Each OR-block in Figure 4 consists of two parts. The lower part lists all the `QNode`s that belong to the OR-block and the upper part is an expression P that records the logical combination of `QNode`s in the OR-block.

OR-predicate evaluation becomes intuitive with the OR-block concept. Consider OR-predicate n_{11} in Figure 4, where $P(n_{11})$ is "n_{12} `or` (n_{14} `and` n_{15})". It is easily verifiable that element e_1 satisfies OR-predicate n_{11} (and OR-block n_{11} as well) if $P(n_{11})$ is `true` after we substitute `edgeTest`(n_i) for each `QNode` n_i in $P(n_{11})$.

We need to take more care of OR-predicates that contain more than one OR-block. For example, OR-predicate n_5

has two OR-blocks. Suppose that `edgeTest`(n_6) is `false` and `edgeTest`(n_7) is `true`. For e_4 to satisfy OR-predicate n_5, e_7 should also satisfy OR-block n_8. In brief, to evaluate an OR-predicate n, it is insufficient to simply substitute `edgeTest`(n_i) for each `QNode` n_i in $P(n)$: we should take into account whether element e_i has a match for subtree n_i if n_i is not a leaf node. Definition 5 on OR-predicate evaluation complements Definition 3.

Definition 5. [**OR-predicate Evaluation**] Let `ONode` n be the root of an OR-predicate connected to `QNode` q, whose associated element is e. We say element e satisfies OR-predicate n if $P(n)$ is `true` by replacing each `QNode` n_i in $P(n)$ with a boolean function as follows: if n_i is a leaf node, replace n_i with `edgeTest`(n_i); otherwise, replace n_i with the boolean value (`edgeTest`(n_i) **and** e_i has a match for subtree n_i).

3.3 The Logical-max QNode in an OR-block

We proceed to describe an interesting concept: the **logical-max** `QNode` in an OR-block. This concept generalizes COROLLARY 1 and considers how to judge that a given element will never satisfy an OR-block. It plays an important role in the `AND/OR`-twig join algorithms that we are going to present in Section 4 and Section 5.

We make a few assumptions in the following discussions. All `QNode`s in the query Q are ancestor-descendant `QNode`s (see, Section 2.2). Each `QNode` $q_i \in Q$ is associated with an element node list T_{q_i}. Element nodes in T_{q_i} are encoded with region codes and sorted by the *start* field in ascending order. Element $e_i \in T_{q_i}$ is currently associated with q_i and we are only allowed to move forward in T_{q_i}.

Consider a `QNode` q in a query tree Q and an OR-block n connected to q. Suppose that OR-block n contains k `QNode`s $q_1, q_2, \cdots, q_k$ and among these k `QNode`s, the element node e_{min} of q_{min} has the smallest *start* value. It is obviously the case (by COROLLARY 1) that if $e.end < e_{min}.start$, then e does not satisfy OR-block n and will not satisfy OR-block n no matter how T_{q_i} is forwarded.

We are interested in finding the *largest possible* threshold value v ($\geq e_{min}.start$) such that if $e.end < v$ then e does not satisfy OR-block n and will not satisfy OR-block n no matter how T_{q_i} is forwarded. The main application of such a threshold value v is that it enables effective element skipping in our algorithms to be presented in the next sections. The `ORBlockMax`(n) algorithm shown in Algorithm 1 returns a `QNode` q_{max} in OR-block n such that $e_{max}.start$ is the threshold value v desired.

Algorithm 1 ORBlockMax(n)

```
 1: if $n$ is a QNode then
 2:     return $n$;
 3: else
 4:     for each $n_i \in$ children($n$) do
 5:         $q_i$ = ORBlockMax($n_i$);
 6:     end for
 7:     if $n$ is an ANode then
 8:         return $\arg\max_{q_i}\{e_i.start\}$, for $q_i$ initialized at line 5;
 9:     else
10:         return $\arg\min_{q_i}\{e_i.start\}$, for $q_i$ initialized at line 5;
```

In Algorithm 1, function $\arg\max_{x_i}\{\mathrm{f}(x_i)\}$ returns a variable x_m, among all variables x_i in consideration, such that $\mathrm{f}(x_m)$ is no smaller than any $\mathrm{f}(x_i)$. Function $\arg\min_{x_i}\{\mathrm{f}(x_i)\}$ returns a variable x_m such that $\mathrm{f}(x_m)$ is no greater than any $\mathrm{f}(x_i)$. Ties are broken arbitrarily.

The intuitive explanation of `ORBlockMax` is that, for `QNode`s connected by `AND` logic, we pick the `QNode` with the *maximum start* value, while for `QNode`s connected by `OR` logic, we pick the `QNode` with the *minimum start* value. Take the query tree in Figure 3 as an example. Suppose that the element *start* value for n_i is i, then `ORBlockMax`(n_5) returns n_6 while `ORBlockMax`(n_{11}) returns n_{12}.

THEOREM 1. *Let q be a `QNode` in a query tree and n be an OR-block connected to q. Assume that each `QNode` q_i in OR-block n is associated with a forward-only element node list T_{q_i}. Element nodes in T_{q_i} are assigned region codes and sorted by the start field in ascending order. Element e corresponds to q and $e_i \in T_{q_i}$ corresponds to q_i. Let q_{max} = ORBlockMax(n). If $e.end < e_{max}.start$, then element e does not satisfy OR-block n and will never satisfy OR-block n after any T_{q_i} is forwarded.*

PROOF. Our proof consists of two parts. In part 1, we prove the correctness of THEOREM 1 when the logical expression $P(n)$ for OR-block n is in disjunctive normal form (DNF). In part 2, we show that, when $P(n)$ is not in DNF, OR-block n can be transformed into a new OR-block n' with the same set of leaf `QNode`s as OR-block n such that (1) $P(n')$ is in DNF; (2) $P(n')$ is equivalent to $P(n)$; and (3) `ORBlockMax`(n') is the same as `ORBlockMax`(n).

Part 1: Since $P(n)$ is in DNF, let us assume that $P(n)$ has k disjuncts and each disjunct D_i ($1 \leq i \leq k$) is a conjunction of t_i `QNode`s. Formally, $P(n) = \bigvee_{i=1}^{k} D_i$, where each $D_i = \bigwedge_{j=1}^{t_i} q_{ij}$. An example OR-block n whose $P(n)$ is in DNF is illustrated in Figure 5.

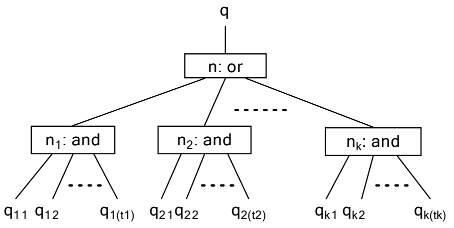


Figure 5: An example OR-block n (connected to q) such that $P(n)$ is in disjunctive normal form (DNF)

How `ORBlockMax`(n) works for the query tree in Figure 5 becomes clear: it calls `ORBlockMax` for each `ANode` subtree n_i and among all the k returned `QNode`s, the `QNode` with the minimum *start* value is returned. Suppose that q_{xy} = `ORBlockMax`(n), which means that each disjunct D_i has a `QNode` $q_{x_i y_i}$ such that $e_{x_i y_i}.start \geq e_{xy}.start$. By the given condition $e.end < e_{xy}.start$ in THEOREM 1, we have $e.end < e_{x_i y_i}.start$, which implies that `edgeTest`($q_{x_i y_i}$) is `false` and will keep to be `false` when $q_{x_i y_i}$ is forwarded. For element e to satisfy OR-block n, there must exist at least one `ANode` subtree n_x such that for each `QNode` q_{xj} ($1 \leq j \leq t_x$), `edgeTest`(q_{xj}) is `true`. This contradicts the given condition. In conclusion, THEOREM 1 is proved when $P(n)$ is in DNF.

Part 2: We are left to prove that we can transform an arbitrarily shaped OR-block n into an equivalent, normalized OR-block n' as shown in Figure 5 while keeping `ORBlockMax`(n') the same as `ORBlockMax`(n).

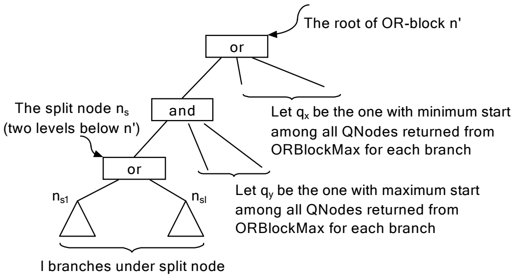


Figure 6: In a split operation, we always pick an `ONode` that is two levels below the root of OR-block n'—such an `ONode` must exist if $P(n')$ is not in DNF.

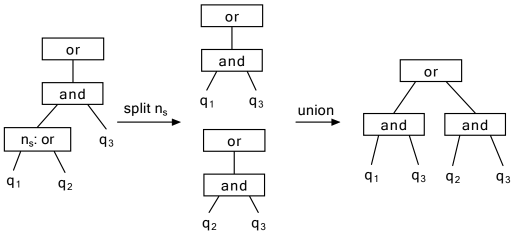


Figure 7: The transformation from an arbitrary OR-block n into an equivalent OR-block n' such that $P(n')$ is in disjunctive normal form (DNF)

Let OR-block n' initially be OR-block n. The transformation from OR-block n into OR-block n' is a repeated application of the following three steps until $P(n')$ is in DNF.

1. **Split:** First, get an `ONode` n_s such that n_s is two levels below root n' as shown in Figure 6. Such an `ONode` must exist according to the simplification rules described in Section 2.2.1. Assume that n_s has l child nodes $n_{s_1}, n_{s_2}, \cdots, n_{s_l}$. Second, make l copies of n', denoted as $n'_1, n'_2, \cdots, n'_l$, and replace subtree n_s in n'_i $(1 \leq i \leq l)$ with subtree n_{s_i}. The left part of Figure 7 illustrates one split step.

2. **Union:** Combine the resultant OR-blocks n'_i at their roots. The right part of Figure 7 shows a union process.

3. **Simplify:** Apply the simplification rules to n'.

With the described transformation process, it is sufficient to prove that, `ORBlockMax`(n') keeps unchanged before and after each three-step process.

Consider the parameters shown in Figure 6. Let q_{s_i} be `ORBlockMax`(n_{s_i}). According to the definition of `ORBlockMax`, we know that, `ORBlockMax`(n') is $\min(\max(\min_{i=1}^{l} q_{s_i}, q_y), q_x)$ initially. After the three-step process, `ORBlockMax`(n') becomes $\min_{i=1}^{l} \min(\max(q_{s_i}, q_y), q_x)$. These two min-max formulae can be shown to be equivalent.

Based on part 1 and part 2, THEOREM 1 is proved. ☐

4. A MERGE-BASED ALGORITHM

In this section, we present `GTwigMerge`, an algorithm for finding all matches (see, Definition 3) of an `AND`/`OR`-twig query against an XML document. It is worth noticing that, although `GTwigMerge` shares similarity with the `TSGeneric` algorithm in the previous work [11], it makes important extensions to handle `AND`/`OR`-twigs.

We will first introduce some data structures and notations, in addition to those described in Section 2.2.2, to be used by the `GTwigMerge` algorithm.

4.1 Data Structures and Notations

Function `isLeaf`(n) evaluates `true` if a node n is a leaf node and `isRoot`(n) returns `true` if node n is a root node. Function `subtreeQNodes`(q) returns all `QNode`s in subtree q (inclusive). `Qsibling`(q) denotes the set of `QNode`s q_i such that $q_i \neq q$ and `Qparent`(q_i) = `Qparent`(q). For Figure 3, `Qsibling`$(n_2) = \{n_{12}, n_{14}, n_{15}\}$.

We assume each `QNode` q is associated with a list T_q of element nodes, which are encoded with (*start*, *end*, *level*) and sorted in ascending order of the *start* field. Typically, element node lists are retrieved through a *tag index*, which returns a list of element nodes for a given tag. If q has value predicates, element nodes are generally retrieved from a B-tree index. For example, a composite B-tree index on (*text*, *start*) can process value selections efficiently.

The `GTwigMerge` algorithm keeps two data structures during execution: a cursor C_q and a stack S_q for each `QNode` q.

The cursor C_q points to the current element in T_q. When we refer to Section 3, C_q also acts as the associated element of q, unless explicitly specified. Function `end`(C_q) tests whether C_q is at the end of T_q. We can access the attribute values of C_q by $C_q \rightarrow start$, $C_q \rightarrow end$ and $C_q \rightarrow level$. The cursor can be forwarded to the next element in T_q with $C_q \rightarrow$`advance()`. Initially, C_q points to the head of T_q.

Stack S_q may cache some elements before C_q such that each element is a descendant of the element below it. Each element node in S_q keeps a pointer to its nearest ancestor (i.e., the one with the largest *level* value) in $S_{\texttt{Qparent}(q)}$. With the pointer, cached elements in stacks represent the partial results that might be extended to full results. Stack S_q is initially empty.

4.2 The GTwigMerge Algorithm

The main algorithm for `GTwigMerge` is shown in Algorithm 2 and the procedure `GetQNode` is shown in Algorithm 3.

Algorithm 2 The Main Algorithm of GTwigMerge

```
while not end($root$) do
    $q$ = GetQNode($root$); {Algorithm 3}
    if not isRoot($q$) then
        cleanStack($S_{\texttt{Qparent}(q)}, C_q$);
    cleanStack($S_q, C_q$);
    if isRoot($q$) or (not empty($S_{\texttt{Qparent}(q)}$)) then
        if not isLeaf($q$) then
            push($S_q, C_q$, isRoot($q$)?$-1$ : top($S_{\texttt{Qparent}(q)}$));
        if $q$ is inside OR-predicate(s) then
            Reevaluate OR-predicates; {Section 4.2.3 }
        else if $q$ has no QNode children then
            outputPathSolutions($C_q$);
    $C_q \rightarrow$advance();
end while
mergePathSolutions();
PROCEDURE cleanStack($S_p, C_q$)
    pop all elements $e_i$ from $S_p$ such that $e_i.end < C_q \rightarrow start$;
FUNCTION end($q$)
    $\forall q_i \in$ subtreeQNodes($q$): isLeaf($q_i$) $\wedge$ end($C_{q_i}$);
PROCEDURE push($S_p, C_q, ptr$)
    push the pair ($C_q$, $ptr$) onto stack $S_p$;
```

Algorithm 3 GetQNode(q)

```
1: if isLeaf($q$) then
2:     return $q$;
3: for each $q_i \in$ Qchildren($q$) do
4:     $q'$ = GetQNode($q_i$);
5:     if $q' \neq q_i$ then
6:         return $q'$;
7: end for
8: $q_{max}$ = getMaxQChild($q$);
9: while $C_q \rightarrow end < C_{q_{max}} \rightarrow start$ do
10:     $C_q \rightarrow$ advance();
11: end while
12: $q_{min} = \arg\min_{q_i}\{C_{q_i} \rightarrow start\}$, $q_i \in$ Qchildren($q$);
13: if hasExtension($q$) and $C_q \rightarrow start < C_{q_{min}} \rightarrow start$ then
14:     return $q$;
15: else
16:     return $q_{min}$;
FUNCTION getMaxQChild($q$)
1: for each $n_i \in$ children($q$) do
2:     if $n_i$ is a QNode then
3:         $q_i = n_i$;
4:     else
5:         $q_i$ = ORBlockMax($n_i$); {Algorithm 1}
6: end for
7: return $\arg\max_{q_i}\{C_{q_i} \rightarrow start\}$, for $q_i$ initialized at lines 3, 5;
FUNCTION hasExtension($q$)
1: return true if $C_q$ has a match (see, Definition 3) by regarding
   all QNodes in the subtree $q$ as ancestor-descendant QNodes (see,
   Section 2.2); otherwise, return false.
```

`GTwigMerge` operates in two phases. In the first phase, it repeatedly calls the `GetQNode` algorithm with the query root as the parameter to get the next `QNode` for processing and outputs path solutions. In the second phase, the individual path solutions are merged to compute the matching instances for the `AND`/`OR`-twig query.

In Section 4.2.1, we explain the `GetQNode`(q) algorithm. Section 4.2.2 describes the main algorithm in more detail. Techniques for handling OR-predicates containing parent-child `QNode`s are presented in Section 4.2.3.

4.2.1 The GetQNode Algorithm

`GetQNode`(q) is a procedure called in Algorithm 2. It returns a `QNode` q_x with three properties: (1) q_x has an **extension** (see, Algorithm 3) and $C_{q_x} \rightarrow start < C_{q_i} \rightarrow start$ for all $q_i \in$ `Qchildren`(q_x), if any; (2) If $q_x \neq q$, then $C_{q_x} \rightarrow start < C_{q_j} \rightarrow start$, for all $q_j \in$ `Qsibling`(q_x), if any; and (3) If $q_x \neq q$, then $C_{q_x} \rightarrow start < C_{\texttt{Qparent}(q_x)} \rightarrow start$. These properties guarantee the correctness of the main algorithm in processing C_{q_x}.

At lines 3-7 in Algorithm 3, we invoke `GetQNode` for each $q_i \in$ `Qchildren`(q). If any returned node q' is not equal to q_i, we can return q' outright (line 6). Otherwise, since each child `QNode` q_i has an extension, we will try to locate an extension for q by skipping over elements in T_q (line 10) based on q_{max}, which is returned by `getMaxQChild`(q). By THEOREM 1, the skipped elements do not contribute to new output results.

It is incorrect to use $q'_{max} = \arg\max_{q_i}\{C_{q_i} \rightarrow start\}$ to skip element nodes in T_q at line 10. Consider an `AND`/`OR`-twig query `//a[.//b or .//c]//d` and its element node lists shown in Figure 8, where the regions of element nodes are represented as intervals. Suppose the cursors for the four `QNode`s are (a_1, b_1, c_3, d_1). Here, q'_{max} is n_4 (i.e., $C_{q'_{max}} = c_3$). If we use c_3 to skip elements in T_{n_1}, we will reach a_7. But both a_5 and a_6 have matches if C_{n_3} is forwarded to b_3.

Example 1. Consider Figure 8. Suppose the four cursor elements are initially at (a_1, b_1, c_1, d_1). At the first call of `GetQNode`(n_1), C_{n_1} is forwarded—by element d_1 of n_5 returned by `getMaxQChild`(n_1)—to element a_5, which is the first element in T_{n_1} whose *end* is not smaller than d_1 (recall, lines 9-11 in Algorithm 3), and n_3 is returned. Before the next call of `GetQNode`(n_1), the cursors are (a_5, b_2, c_1, d_1). The next three calls of `GetQNode`(n_1) return n_3 once and n_4 twice to consume b_2, c_1 and c_2. After that, the cursors are at (a_5, b_3, c_3, d_1). The next call of `GetQNode`(n_1) will return n_1, which has an extension because both b_3 and d_1 are descendants of a_5 (i.e., C_{n_1}).

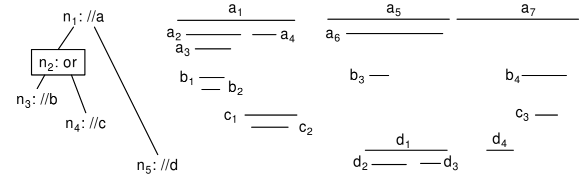


Figure 8: An example query tree and the element node lists associated with its `QNodes`

4.2.2 The Main Algorithm of GTwigMerge

Algorithm 2 shows the main algorithm for `GTwigMerge`. It repeatedly calls `GetQNode`(*root*) to get the next `QNode` q to process, as described next.

First of all, we pop elements from the `QNode` parent stack $S_{\texttt{Qparent}(q)}$ and stack S_q (lines 4-5). The popped elements do not contribute to new outputs according to COROLLARY 1 and the three properties of `QNode` q returned by `GetQNode`(*root*).

We continue to process C_q if q is either a root node or C_q has ancestor elements in the parent stack $S_{\texttt{Qparent}(q)}$, which means there is a potential for it to contribute to final matches. The code at lines 9-12 is unique compared to the previous algorithms for `AND`-twigs because we need to refine the definition for an **output twig instance** of an `AND`/`OR`-twig query. In the previous algorithms, an output twig instance contains elements from *all* `QNode`s in the query. It becomes problematic if we adopt the same output model because a match of an `AND`/`OR`-twig query may be contributed by only some of the `QNode`s in the query. Here, we adopt a simple yet intuitive output model: *Each output twig instance for an `AND/OR`-twig query comprises of elements from `QNodes` that are not inside any OR-predicate* and OR-predicates only serve as filters. The `QNode`s for output are called **output nodes**.

We will explain the work done at line 10 in Section 4.2.3. For now, simply regard line 10 as a black-box.

If q is a **leaf output node**, all path solutions for element C_q are output (line 12). A `QNode` is a leaf output node if it is an output node and has no `QNode` children. For example, `QNode`s n_3 and n_4 in Figure 3 are leaf output nodes, and they correspond to root-to-leaf paths $\{n_3 \leftarrow n_2 \leftarrow n_1\}$ and $\{n_4 \leftarrow n_2 \leftarrow n_1\}$ respectively.

Two points are worth noticing on outputting path solutions. First, if there are parent-child `QNode`s in a path solution, we should discard the path solution if the corresponding edges do not satisfy parent-child relationship. Second, path solutions for each root-to-leaf path should be sorted in root-to-leaf order as required by a merge process that follows. Since path solutions for current C_q could be *larger* than those for some later C_q in root-to-leaf order, we need to *block* the output when necessary. We refer the interested reader to the previous work [3, 11] for more detail.

After all possible path solutions are generated, they are merged to compute the output twig instances of the `AND/OR`-twig query (line 15). Merging multiple lists of sorted path solutions is a simple practice of a multi-way merge join.

4.2.3 OR-predicates with Parent-child QNodes

Suppose that, in Algorithm 2, `GetQNode`(*root*) returns a `QNode` q with k OR-predicates $n_1, n_2, \cdots, n_k$. If all `QNode`s in the k subtrees are ancestor-descendant `QNode`s, then C_q satisfies all these OR-predicates by definition because q has an extension. On the other hand, if some subtree n_x contains parent-child `QNode`s, C_q might not satisfy OR-predicate n_x. It is even possible that this particular element C_q could not satisfy OR-predicate n_x at all. This causes a potential problem if q is an output node, because C_q may participate in path solutions.

The following observation is important to solving the identified problem above: for any element e_i in a stack, all possible extensions in which element e_i participates must have been returned by `GetQNode`(*root*) and examined in Algorithm 2 before e_i is popped. Based on the observation, for each output node q_j with OR-predicates, we can keep evaluating OR-predicates for each element e_i in S_{q_j} and know whether e_i satisfies all OR-predicates (in some extension examined) when it is popped. The evaluation results, in the form of (q_j, e_i), can be reported to the merge routine for validating path solutions involving q_j.

The Data Structure

We introduce the data structure required to evaluate OR-predicates for stack elements.

Given a `QNode` q inside an OR-predicate, each element e_i in S_q keeps a hash table H_{e_i} that maps a `QNode` to a boolean value as follows. For each $q_j \in$ `Qchildren`(q), if there exists some element $e'_j \in T_{q_j}$ such that e'_j has a match and `edgeTest`(e_i, e'_j) is `true`, then $\mathrm{H}_{e_i}[q_j]$ is set to `true`. Note that, e'_j may have already been processed and no longer stay in stack S_{q_j}.

In addition, for an output node q with OR-predicates, we also keep a hash table H_{e_i} for each element $e_i \in S_q$ except that we are only interested in hash entries $\mathrm{H}_{e_i}[q_j]$ where $q_j \in$ `Qchildren`(q) and q_j is inside an OR-predicate. For example, for `QNode` n_1 in Figure 3, hash tables for elements in S_{n_1} will have entries for n_{12}, n_{14} and n_{15} but not for n_2, which is an output node itself.

With the hash table structure, it is straightforward to evaluate OR-predicates based on Definition 5. Take Figure 3 as an example. Suppose we are going to pop an element e_x from S_{n_1}. To evaluate whether there has ever been an extension in which element e_x satisfies OR-predicate n_{11}, we only need to replace the `QNode`s in $P(n_{11})$ with their hash values in H_{e_x} (i.e., $\mathrm{H}_{e_x}[n_{12}]$, $\mathrm{H}_{e_x}[n_{14}]$ and $\mathrm{H}_{e_x}[n_{15}]$).

Maintenance of Hash Tables

When an element is pushed onto a stack (line 8, Algorithm 2), its hash table is initialized to be empty. An empty hash table returns `false` for any `QNode`. All other maintenance of hash tables is carried out at line 10 as described below.

If q is not a leaf `QNode`, nothing needs to be done. Otherwise, we update the hash tables of the elements in the stacks along the path from q up to the first output node encountered. Example 2 shows an update process for a simple case where there is only one element in each stack. It is easy to generalize it to the case when stacks have multiple elements.

Example 2. Consider Figure 3. Suppose `GetQNode`(*root*) in Algorithm 2 returns n_9. Also suppose that stack S_{n_7} contains element e_7 and stack S_{n_4} contains element e_4 such that both `edgeTest`(e_7, C_{n_9}) and `edgeTest`(e_4, e_7) are `true`. Assume that all element hash tables are empty. The update involves the following steps: (1) Set $\mathrm{H}_{e_7}[n_9]$ to `true`; (2) Since e_7 now shows to have a match and `edgeTest`(e_4, e_7) is `true`, we set $\mathrm{H}_{e_4}[n_7]$ to `true` as well; and (3) Report (n_4, e_4) because e_4 has been proved to satisfy its only OR-predicate n_5.

We conclude Section 4.2 with the following theorem, which asserts the correctness of the `GTwigMerge` algorithm:

THEOREM 2. *Given an `AND/OR`-twig query Q against an XML database D, the `GTwigMerge` algorithm correctly returns all the output twig instances for Q on D.*

4.3 Cost Analysis of GTwigMerge

We now analyze the worst-case I/O cost for the `GTwigMerge` algorithm. In the interest of space, we omit the analysis on the worst-case CPU cost.

The I/O cost of `GTwigMerge` consists of two parts: the I/O cost for accessing element node lists and the I/O cost for outputting and joining path solutions. Since we always advance the cursors and never backtrack, it is obvious that accessing elements requires only linear worst-case I/O cost. If all `QNode`s are ancestor-descendant `QNode`s, then every individual path solution takes part in at least one final output. Thus, the I/O cost for outputting and joining path solutions is linear to the total size of output twig instances. Hence, we have the following theorem:

THEOREM 3. *Given an `AND/OR`-twig query Q containing only ancestor-descendant `QNodes` and an XML document D, `GTwigMerge` has worst-case I/O cost linear to $|input|+|output|$ and worst-case CPU cost linear to $|Q| \cdot |input| + |output|$, where $|Q|$ is the size of the query, $|input|$ is the total size of the element node lists associated with `QNodes`, $|output|$ is the output size.*

Twigs with Parent-child QNodes

Algorithm `GTwigMerge` still works correctly when `AND/OR`-twig queries contain parent-child `QNode`s. However, the optimality in terms of worst-case I/O and CPU cost is no longer guaranteed.

There are two reasons for the sub-optimality. First, if some output nodes are parent-child `QNode`s, a path solution (with element nodes from parent-child `QNode`s) may turn out not to join with any other path solutions to form an output twig instance. Thus, irrelevant I/O access is caused. This point is elaborated in the previous work [11]. Second, if some OR-predicates in an `AND/OR`-twig contain parent-child `QNode`s, a path solution may contain an element node that eventually turns out not to satisfy all its OR-predicates. Such path solutions are another source of irrelevant I/O and CPU cost.

5. AN INDEX-BASED ALGORITHM

Although `GTwigMerge` only requires one scan of input element node lists, such linear cost might be practically unsatisfiable for *selective* queries, where a large part of the input data does not contribute to final outputs. It is most desirable to avoid accessing the data without matches. In this section, we present an algorithm that processes `AND`/`OR`-twigs using available indexes. For our algorithm to be independent from a specific index implementation, we assume that two new cursor methods are provided to access element node lists through indexes:

1. $C_q \rightarrow$`fwdToAncestorOf`(C_p) forwards C_q to the first ancestor of C_p. If no such ancestor exists, C_q is set to the first element e such that $e.start > C_p \rightarrow start$.
2. $C_q \rightarrow$`fwdBeyond`(C_p) forwards C_q to the first element e such that $e.start > C_p \rightarrow start$.

5.1 The GTwigIndex Algorithm

The `GTwigIndex` algorithm shares the same main algorithm (i.e., Algorithm 2) with `GTwigMerge` except that it replaces the merge-based algorithm `GetQNode` with an index-based one, namely `GetQNodeIdx` which extends `GetQNode` and exploits available indexes on element node lists through the two additional cursor methods: `fwdToAncestorOf` and `fwdBeyond`.

The `GetQNodeIdx` algorithm addresses the limitations of the previous work, which also considers skipping elements with indexes but only works for `AND`-twigs.

5.1.1 The GetQNodeIdx Algorithm

Algorithm 4 shows the `GetQNodeIdx` algorithm. It takes different actions depending on whether S_q is empty or not.

Algorithm 4 GetQNodeIdx(q)

```
1: if isLeaf($q$) then
2:    return $q$;
3: if not empty($S_q$) then
4:    for each $q_i \in$ Qchildren($q$) do
5:       $q'$ = GetQNodeIdx($q_i$);
6:       if $q' \neq q_i$ then
7:          return $q'$;
8:    end for
9:    $q_{max}$ = getMaxQChild($q$); {Defined in Algorithm 3}
10:   $C_q \rightarrow$fwdToAncestorOf($C_{q_{max}}$);
11: else
12:   LocateExtension($q$); {Algorithm 5}
13: $q_{min} = \arg\min_{q_i}\{C_{q_i} \rightarrow start\}$, $q_i \in$ Qchildren($q$);
14: if hasExtension($q$) and $C_q \rightarrow start < C_{q_{min}} \rightarrow start$ then
15:   return $q$;
16: else
17:   return $q_{min}$;
```

If S_q is not empty, the process is similar to that in `GetQNode` except that we use `fwdToAncestorOf` to skip elements (line 10).

There is another opportunity for skipping elements: if S_q is empty, we can directly locate an extension for q (line 12). The rationale is that, when stack S_q is empty, for any $q_j \in$ `subtreeQNodes`(q) to contribute a new path solution, it must participate in an extension involving some element node in T_q. Next, we explain the `LocateExtension` algorithm in detail, with the emphasis on techniques to meet new challenges presented by `AND`/`OR`-twigs.

5.1.2 The LocateExtension Algorithm

The purpose of `LocateExtension`(q) is to locate an extension for a given `QNode` q, particularly, using indexes.

For a `QNode` q in an `AND`-twig query, we can do the following to locate its extension: (1) Pick an edge $(q_i =$ `parent`$(q_j), q_j)$ in subtree q, such that C_{q_i} is not an ancestor of C_{q_j}; (2) Forward C_{q_i} and C_{q_j} appropriately until C_{q_i} is an ancestor of C_{q_j}; and (3) Repeat (1) until no such edge can be found.

When it comes to `AND`/`OR`-twig queries, matching each `QNode` edge $(q_i =$ `Qparent`$(q_j), q_j)$ *in isolation* may lose correct results if q_j lies inside an OR-block connected to q_i. The reason is that element nodes $e_i \in T_{q_i}$ that do not have matches with any element in T_{q_j} may match with element nodes in T_{q_x}, where q_x is some other `QNode` in the OR-block. Given Figure 8, suppose we need to locate an extension for `QNode` n_1 and the cursors are (a_2, b_1, c_1, d_1). If we pick the `QNode` edge (n_1, n_4) to process, we will eventually move their cursors to (a_7, c_3). As a result, a_5 and a_6 are erroneously skipped.

In other words, an OR-block is an *atomic processing unit* whose components cannot be matched separately. We give the definition for a **broken edge**, which can be matched in isolation:

Definition 6. [**Broken Edge**] Given an edge (q, n) in a query tree Q, where q is a `QNode` and $n \in$ `children`(q), we say that the edge (q, n) is *broken* if it satisfies one of the following: (1) n is a `QNode` and C_q is not an ancestor of C_n; and (2) n is an `ONode` and C_q does not satisfy OR-block n.

Algorithm 5 shows the details of `LocateExtension`. It keeps matching broken edges until q has an extension. Each time, we pick the broken edge with the highest priority for processing. In the following, we discuss how to fix a broken edge and how to calculate the **priority of a broken edge**.

Algorithm 5 LocateExtension(q)

```
while not hasExtension($q$) and not end($q$) do
   Let BrokenEdges be all broken edges in subtree $q$;
   for each $(p_i, n_i) \in$ BrokenEdges do
      $priority_i$ = calculated priority of edge $(p_i, n_i)$;
   end for
   $(q_x, n_x) = \arg\max_{(p_i, n_i)}\{priority_i\}$;
   if $n_x$ is a QNode then
      fixEdge($q_x, n_x$);
   else
      fixBlock($q_x, n_x$);
end while
PROCEDURE fixBlock($q, n$)
while $q$ does not satisfy OR-block $n$ do
   $q_{max}$ = ORBlockMax($n$); {Algorithm 1}
   $C_q \rightarrow$fwdToAncestorOf($q_{max}$);
   if not end($C_q$) then
      $q_{min} = \arg\min_{q_i}\{C_{q_i} \rightarrow start\}$, $q_i \in$ Qchildren($n$);
      $C_{q_{min}} \rightarrow$fwdBeyond($C_q$);
end while
PROCEDURE fixEdge($q, n$)
while $(q, n)$ is broken and not (end($C_q$) or end($C_n$)) do
   if $C_q \rightarrow start < C_n \rightarrow start$ then
      $C_q \rightarrow$fwdToAncestorOf($C_n$);
   else
      $C_n \rightarrow$fwdBeyond($C_q$);
end while
```

Fixing a Broken Edge

In a broken edge (q, n), the node n could be either a `QNode` or an `ONode` (i.e., the root of an OR-block). There are two procedures (i.e., `fixEdge` and `fixBlock`) defined in Algorithm 5 to deal with these two types of broken edges respectively.

Consider the `fixEdge`(q, n) procedure. It skips ancestor element nodes and descendant element nodes alternately until C_q is an ancestor of C_n or either of the two element node lists (T_q and T_n) is exhausted.

The procedure `fixBlock`(q, n) tries to make q satisfy OR-block n by forwarding the cursors of q and $q_i \in$ OR-block n appropriately in each while loop: (1) An appropriate `QNode` q_{max} is picked and used to skip element nodes in T_q (lines 2-3); and (2) C_q is used to skip elements in $C_{q_{min}}$ (lines 5-6).

Priority of a Broken Edge

In the `LocateExtension` algorithm, we pick the broken edge that has the highest priority. In the perfect case, we should have a priority calculation function such that the overall cost for running the `LocateExtension` algorithm is minimal. This is essentially a query optimization problem. Realistically, it is very difficult to find such an optimal priority assignment function. We consider using statistics for computing the priority of a broken edge.

The *maximum distance* (MD) heuristic in the previous work [11] assigns the priority to a broken edge (q_i, q_j) according to the estimated average distance $AvgDist_{q_i \lhd q_j}$ between pairs of matches for edge (q_i, q_j) and picks the edge with the largest $AvgDist$ value. The basic idea is that we should choose a broken edge whose next match is the farthest from the current cursors so that we can hopefully skip the most number of elements when matching other broken edges. This is essentially a local optimization solution but it has been shown to be quite robust.

The MD heuristic is applicable for our `LocateExtension` algorithm except when a broken edge involves an OR-block. We adopt the idea of logical-max `QNode` for OR-blocks and extend the MD heuristic to handle a broken edge involving an OR-block:

Definition 7. [**Priority of an OR-block Broken Edge**] Given a `QNode` q and an OR-block n connected to q. Assume that OR-block n contains k `QNode`s $q_1, q_2, \cdots, q_k$. Let $AvgDist_{q \lhd q_i}$ be the estimated average distance between pairs of matches for the `QNode` edge (q, q_i). Then, $AvgDist_{q \lhd n}$ is defined as the value of $AvgDist_{q \lhd q_{max}}$, where q_{max} is the `QNode` returned by `ORBlockMax`(n) if we use the $AvgDist_{q \lhd q_i}$ value instead of $e_i.start$ in the `ORBlockMax` algorithm (lines 8 and 10).

5.2 Cost Analysis of GTwigIndex

The worst-case I/O cost for `GTwigIndex` depends on how the cursor methods are implemented for the indexed element node lists. However, since each cursor method always drives the cursor forward, assuming a reasonable implementation, we can draw the conclusion that `GTwigIndex` is as efficient as `GTwigMerge` in terms of worst-case I/O and CPU cost (see, THEOREM 3).

On the other hand, our experimental study will show that `GTwigIndex` outperforms `GTwigMerge` with its sub-linearity performance characteristics, particularly for highly selective twig queries.

6. PERFORMANCE EVALUATION

This section presents experimental results on the performance of the algorithms we proposed, in comparison with the existing state-of-the-art algorithms.

6.1 Experimental Setup

6.1.1 Data Preparation

We used synthetic data for our experiments to control the structure and join characteristics of the XML data. The DTD is shown in Figure 9. We used the IBM XML data generator in Java to generate the structural part of the XML data (i.e., without text values) with default parameters [1]. The generated XML document roughly takes 100MB and contains about 2 million element nodes. We assigned a random number between 1 and 1000 to each text element node as its value.

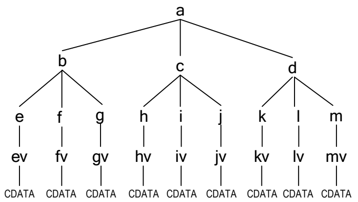


Figure 9: The DTD of the synthetic XML data

6.1.2 Query Generation

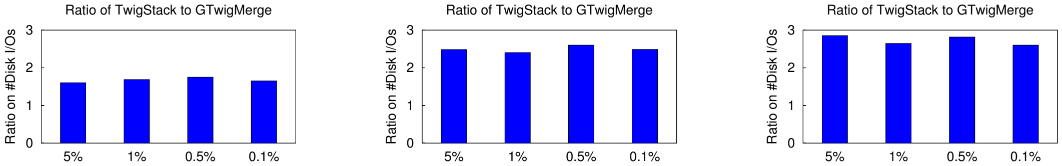
We generated a set of complex `AND`/`OR`-twig queries based on the DTD. To generate an `AND`/`OR`-twig query, we applied a *random walk* starting from the root a of the DTD tree as follows. Whenever we are at some node q of the DTD tree, we randomly pick up some children of q. If only one child is picked, it becomes the child `QNode` of q. If there are two child nodes picked, they could be either directly connected to q or linked to q through an `ONode`. When there are three child nodes picked, we randomly choose one from all possible ways to connect these nodes to q.

In our experiments, we used queries whose query trees contain one, two or three `ONode`s and classified them into three query sets QS1, QS2 and QS3 such that queries in QSi contain i `ONode`(s).

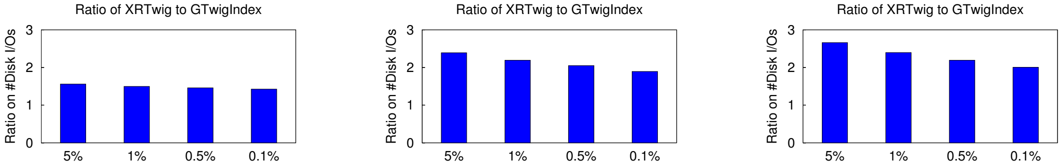
It is obvious that, if every element node participates in a match, there is no opportunity for index-based algorithms to skip elements. Therefore, we applied selection value predicates on text nodes in the queries tested. Value predicates on queries were assigned in two different ways. In one way, all value predicates in a query have the same **selectivity**, which is defined as the percentage of element nodes satisfying a value predicate. The value predicates assigned in this way could have a selectivity of 0.1%, 0.5%, 1% or 5%. In the other way, each value predicate in a query has a selectivity randomly picked from 0.1%, 1%, 5% and 10%. Table 1 shows the average number of output twig instances for a query in each query set with different value predicates.

Table 1: Average Number of Output Twig Instances

	0.1%	0.5%	1%	5%	vary
QS1	23	121	250	1590	1135
QS2	42	216	440	2567	1891
QS3	160	796	1583	7607	3438



(a) QS1 (b) QS2 (c) QS3



(d) QS1 (e) QS2 (f) QS3

Figure 10: Disk I/O performance comparison between the decomposition-based and holistic algorithms for the three query sets with same-selectivity value predicates: X-axis shows the selectivity and Y-axis shows the ratio on disk I/Os performed by the two specified algorithms in each sub figure.

6.1.3 *Evaluation Metrics*

We will use the following three metrics to compare the performance of different algorithms tested in our experiments.

- *number of elements scanned.* This metric indicates the total number of elements scanned during a join. It reflects the ability of an algorithm to skip elements.
- *number of disk I/Os.* This metric keeps the total number of disk pages accessed during a join.
- *CPU time.* The CPU time of an algorithm is obtained by averaging the running times of several consecutive runs with *hot* buffers.

6.1.4 *The Testbed*

We implemented a prototype system using C++. The system includes a storage manager, an LRU buffer manager for measuring disk I/Os, and index modules used by join algorithms. The twig join algorithms tested in our experiments were implemented on top of the test system. All the experiments were conducted on a Pentium 1.0GHz PC with 512M RAM and a 80G IBM hard disk running Windows XP. We used the file system as the storage.

6.2 Decomposition-based vs. Holistic

The first set of experiments investigates the performance advantage of the new holistic algorithms for `AND/OR`-twigs over the naïve decomposition-based approach. By decomposition, we mean that each `AND/OR`-twig is decomposed into a set of `AND`-twigs and we use existing twig join algorithms to evaluate these `AND`-twigs. Most queries in QS1, QS2 and QS3 are decomposed into 2, 4 and 6 `AND`-twigs respectively.

Specifically, we compare `GTwigMerge` with the decomposition-based approach that uses the merge-based algorithm `TwigStack` [3] and compare `GTwigIndex` with the decomposition-based approach that uses the index-based algorithm `XRTwig` [11].

Figure 10 shows the experimental results on disk I/O performance. Since the results in other metrics are qualitatively similar, we omit them in the interest of space.

An immediate observation from the figure is that holistic processing is much more efficient than the decomposition-based approach. In particular, the decomposition-based approach could perform 100% more disk I/Os than those required by the holistic algorithms (Figure 10(c) and 10(f), selectivity = 5%). We expect an even greater ratio for `AND/OR`-twigs with more `ONode`s.

Note that the performance advantage of `GTwigMerge` over `TwigStack` is almost independent of the selectivity of value predicates. In contrast, the relative performance of `XRTwig` improves when the selectivity gets smaller (i.e., the value predicates become more selective). This observation reveals that skipping elements through indexes is most effective when there are no `ONode`s in twig queries. Nevertheless, the benefit of more efficient skipping is compromised by the large number of `AND`-twigs resulted from decomposition and `GTwigIndex` is still the winner even when the value predicate is very selective.

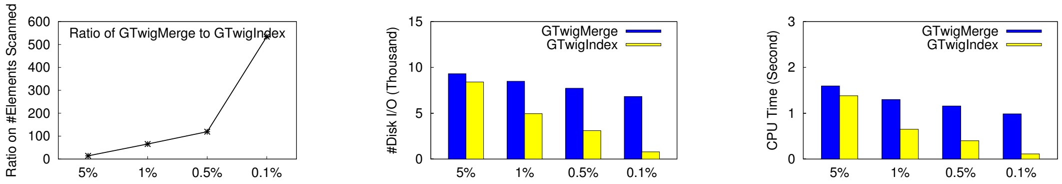
6.3 GTwigMerge vs. GTwigIndex

This part of experiments compares the performance between `GTwigMerge` and `GTwigIndex`. We are particularly interested in the effectiveness of `GTwigIndex` in skipping elements through indexes. We will study the experimental results for both cases when the value predicates in each query have either the same selectivity or varying selectivities.

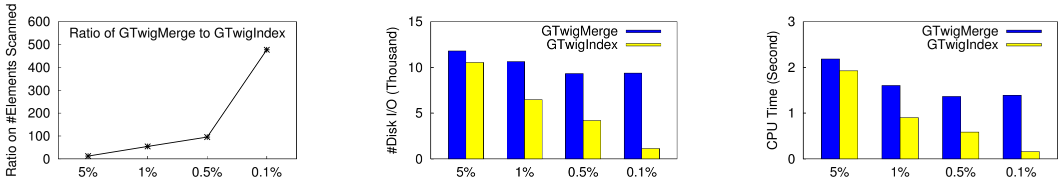
6.3.1 *Same-selectivity Value Predicates*

Figure 11 shows the experimental results when all value predicates in a query have the same selectivity. An overall impression from the results is that `GTwigIndex` *is* able to take advantage of indexes to skip elements so that sublinearity performance can be achieved.

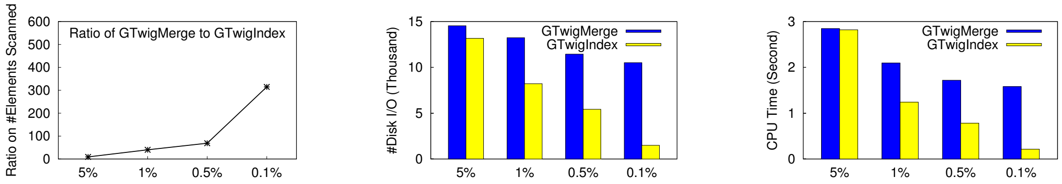
It can be observed from the figure that the performance of `GTwigIndex` improves uniformly when the value predicates become more and more selective while `GTwigMerge` performs similarly no matter how the selectivity changes (`GTwigMerge` achieves small performance gain when the selectivity goes down not because of element skipping but because of the early stop of the algorithm when an element node list is exhausted). For the case when the selectivity is 0.1%, `GTwigMerge` accessed a few hundred times more elements and performed up to 7 times more disk I/Os than did `GTwigIndex`. Note that the number of elements scanned is not necessarily proportional to the number of disk I/Os



(a) QS1 (b) QS1 (c) QS1



(d) QS2 (e) QS2 (f) QS2



(g) QS3 (h) QS3 (i) QS3

Figure 11: Performance comparison between `GTwigMerge` and `GTwigIndex` for the three query sets with same-selectivity value predicates: X-axis shows the selectivity and Y-axis shows the metrics compared between the two algorithms.

performed: an element access will not cause a disk read if the corresponding disk page is already in the buffer.

From Figure 11 (a, d and g) about ratios on elements scanned, it is noticeable that the total number of `ONode`s in queries could have a negative impact on the effectiveness of `GTwigIndex` in skipping elements. This is reasonable because more `ONode`s in a query generally mean that the query is less selective.

6.3.2 Varying-selectivity Value Predicates

Figure 12 shows the results for the case when the value predicates in each query have varying selectivities. The results indicate that `GTwigIndex` is able to take advantage of selective nodes even when there are less selective value predicates in queries.

6.4 Summary

According to the experimental results, we draw the following two conclusions. First, the holistic join algorithms proposed in this paper should be used to evaluate `AND/OR`-twig queries because they have obvious performance advantage over the decomposition-based approaches. Second, using indexes to skip elements during a join keeps to be an important source of speedup even in the presence of OR-predicates in twig queries, especially when there are selective value predicates in the queries.

7. RELATED WORK

With the increasing popularity of XML, query processing and optimization for XML databases has attracted a lot of research interest. The work on Lore [17, 14, 15], Timber [9] and Natix [6] has considered various aspects of managing such data. In particular, twig query matching is identified as a core operation in querying tree-structured XML data. Therefore, there is a rich set of literatures on matching twig queries efficiently. Below, we describe these literatures with the notice that all the existing work deals with `AND`-twigs.

A *structural join* finds all element node pairs from two element node lists T_a and T_d such that each node pair satisfies the given structural relationship. Existing algorithms include MPMGJN [21], $\mathcal{EE}/\mathcal{EA}$-Join [13], Stack-Tree-Desc/Anc [18], B+ [4] and XR-stack [10]. The first three algorithms are merge-based. In particular, the Stack-Tree-Desc/Anc algorithm employs a stack to cache some ancestor nodes in T_a and achieves the best overall performance among the three. B+ and XR-stack leverage special index structures on the data and are shown to achieve sub-linear performance for selective queries.

Different from the *holistic* twig join algorithms, some existing work [20, 8] adopts a traditional cost-based approach: first, the twig query is decomposed into binary edges; second, an execution order for evaluating these edges is chosen so that the total cost is minimal. In particular, [8] introduces an *unnest* operation for evaluating a structural join. The possible disadvantage of these cost-based approaches is that irrelevant intermediate result sizes could get very large, even when the input and final output are manageable in their sizes.

Wang *et al* recently proposed the a dynamic index method called *Vist* for matching twig queries [19]. Vist transforms XML data and queries into structure-encoded sequences. To match a query is to find all occurrences of the query sequence in the indexed data sequences. However, Vist only supports a limited class of twig queries. For example, if two nodes under a branch have the same tag in a twig query, Vist needs to disassemble the query, match sub queries separately and then *join* the results.

There has been much research on constructing efficient

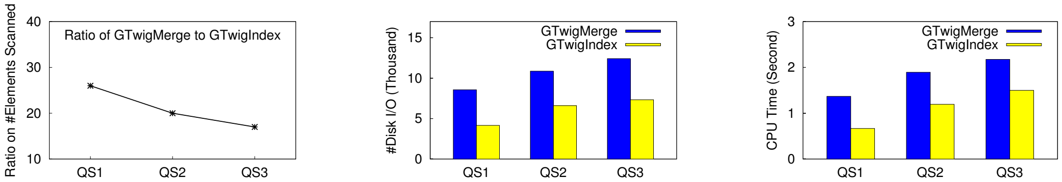


Figure 12: Performance comparison between `GTwigMerge` and `GTwigIndex` for three query sets with varying-selectivity value predicates: X-axis shows the query sets and Y-axis shows the metrics compared between the two algorithms.

structure indexes for matching path expressions. Dataguide [7] and 1-Index [16] can be used to answer simple path queries without branches. The Index Fabric [5] indexes all values by their root-to-leaf paths in a disk-based Patricia trie and can support path queries (with value predicates) efficiently. The above three indexes do not support twig queries effectively. Kaushik *et al* showed that the F&B index *covers* branching path expressions [12]. But F&B index does not support queries with value predicates gracefully because if we take values into consideration, the F&B index could be too large to be practically useful.

8. CONCLUSIONS

Twig query matching has been identified as a core operation in querying tree-structured XML data. In this paper, we proposed *holistic* join algorithms for processing twig queries that contain OR-predicates (i.e., `AND/OR`-twigs). Although the idea of *holistic* twig join processing is not new, applying it for `AND/OR`-twigs is nontrivial. To address the challenges presented by `AND/OR`-twigs, we identified the concept of OR-blocks and studied properties of OR-blocks. With OR-block, an `AND/OR`-twig can be viewed as an `AND`-twig with elements and OR-blocks. As a result, existing holistic-processing techniques for `AND`-twigs can be extended gracefully to handle `AND/OR`-twigs. Experimental studies showed that our new holistic join algorithms are much more effective in handling `AND/OR`-twigs compared to the existing algorithms. In particular, the index-based algorithm has the best overall performance.

9. REFERENCES

[1] IBM XML data generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, pages 497–508, 2001.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, pages 263–274, 2002.

[5] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, 2001.

[6] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

[7] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[8] A. Halverson, J. Burger, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode XML query processing. In *VLDB*, pages 225–236, 2003.

[9] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.

[10] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *ICDE*, pages 253–264, 2003.

[11] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[12] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[14] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[15] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[16] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[17] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. D. Ullman, and J. L. Wiener. LORE: A lightweight object REpository for semistructured data. In *SIGMOD*, pages 549–549, 1996.

[18] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, 2002.

[19] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[20] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, pages 443–454, 2003.

[21] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.