Evaluating Universal Quantification in XML

Shurug Al-Khalifa, Ben B. Liu, and H.V. Jagadish

Abstract—Queries posed to database systems often involve Universal Quantification. Such queries are typically expensive to evaluate. Although they can be handled by basic access methods, for selection, grouping, and so forth, new access methods specifically tailored to evaluate universal quantification can greatly decrease the computational cost. In this paper, we study the efficient evaluation of universal quantification in an XML database. Specifically, we develop a small taxonomy of universal quantification types and define a family of algorithms suitable for handling each. We experimentally demonstrate the performance benefits of the new family of algorithms.

Index Terms—Query processing, XML.

1 Introduction

Users often need to find the results for a query with Universal Quantification. In English, such queries typically involve the use of the word *every*. The following is an example of a Universal Quantification query.

Example 1. "Retrieve books that have the affiliation of *every* author equal to the University of Michigan."

When evaluating such a query, all authors of a book need to be checked for the affiliation with the University of Michigan. Only when all authors of a book satisfy the affiliation condition does this book become part of the query result.

Evaluating a query with Universal Quantification can be very costly. The straightforward way to evaluate the query in Example 1 is to check the authors one by one for each book. If the last author of each book is the one not affiliated with the University of Michigan, then the book has to be eliminated from the result after a great deal of work has been performed. Our goal in this paper is to devise a more efficient evaluation strategy for such queries.

In previous work, new algorithms have been proposed to handle Universal Quantification in relational databases. In the context of XML, some relational techniques can be used and others cannot. The reasons for that are as follows:

- The difference between flat relational tuples and structured XML trees. An XML document is a tree and therefore requires tree-specific algorithms to handle queries imposed on it.
- The nesting property that often occurs in XML documents complicates matters even more. In XML, an element of a specific type can be nested in another element of the same type. This property requires recursion-aware access methods that can handle nesting.

Our interest in this paper is on the universal quantification queries on XML documents stored in a database. The XQuery statement in Fig. 1 formally states the English language query of Example 1. Given a query similar to this, we would like to evaluate it efficiently. In Section 2, we develop a small taxonomy of universal quantification. We show that one type of universal quantification can readily be handled by the suitable adaptation of well-known relational techniques. Efficiently evaluating the other types will occupy us for the bulk of this paper.

The rest of the paper is organized as follows: In Section 3, we introduce our proposed algorithms for processing different flavors of Universal Quantification. An analysis of the proposed algorithms follows in Section 4. The experiments we performed to evaluate our techniques are described in Section 5. We follow that with an examination of previous work done in evaluating Universal Quantification in Section 6. Finally, we conclude this paper in Section 7.

In Fig. 2, we show a fraction of an XML document in tree format on which we will base our examples throughout this paper.

2 Flavors of Universal Quantification

Queries with Universal Quantification differ depending on where the word *every* occurs. There are four locations in a Universal Quantification query where the word *every* can go: after a predicate, before a simple predicate, before a complex predicate, and before a correlational predicate. In this section, we will discuss each flavor and introduce an algorithm to handle one of them then we extend this algorithm to handle the rest. In the following, we formally define each flavor of Universal Quantification, each followed by an example.

Definition 1: Simple predicate. *A simple predicate is a predicate of the form "A* Operator *C," where A is a node,* Operator *is one of* $\{=, \neq, <, \leq, \ldots\}$*, and C is a constant. If the* Operator *is equality or inequality, we have a* Simple Equality Predicate.

Example 1 belongs to this category of predicates.

- *S. Al-Khalifa is with King Saud University, Saudi Arabia. E-mail: shurug@umich.edu.*
- *B.B. Liu and H.V. Jagadish are with the Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI 48109. E-mail: {binliu, jag}@umich.edu.*

Manuscript received 10 May 2006; revised 9 June 2007; accepted 9 July 2007; published online 31 July 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0243-0506.
Digital Object Identifier no. 10.1109/TKDE.2007.190643.

```
FOR $b IN document("bib.xml")//book
WHERE EVERY $a IN $b//author SATISFIES
$a//affiliation = "University of Michigan"
RETURN $b
```

Fig. 1. An XQuery statement that uses Universal Quantification.

Definition 2: Complex predicate. *A complex predicate has the same form as a simple predicate, except that C needs to be computed instead of being a constant. We define* Complex Equality Predicate *similar to Definition 1.*

Example 2. "Retrieve books that have the affiliation of *every* author equal to a university that has more than 25,000 students."

Definition 3: Correlational predicate. *A correlational predicate has the same form as a simple predicate, except that C has to be computed, and its value is a function of the specific instance of left side node A.* Correlational Equality Predicate *is defined similarly as in previous definitions.*

Example 3. "Retrieve books that have the affiliation of *every* author equal to the affiliation of his/her advisor."

In all the above examples, *every* quantifies "//book/author," although we are retrieving "book" nodes.

2.1 After a Predicate

When the word *every* occurs *after* a predicate in a Universal Quantification query, the query sounds like: Find all object1 that are associated with *every* object2. The predicate is underlined and the *every* is in italic (the same for all remaining flavors). The predicate can be simple, complex, or correlational.

Example 4. "Retrieve books that have an affiliation of an author which is equal to *every* university in the state of Michigan."

When evaluating such a query, we need to first get the set of all universities in the state of Michigan. Then, the authors of a book need to be checked for the affiliation with the established set. Only when at least one author of a book satisfies the affiliation condition does this book become part of the query result.

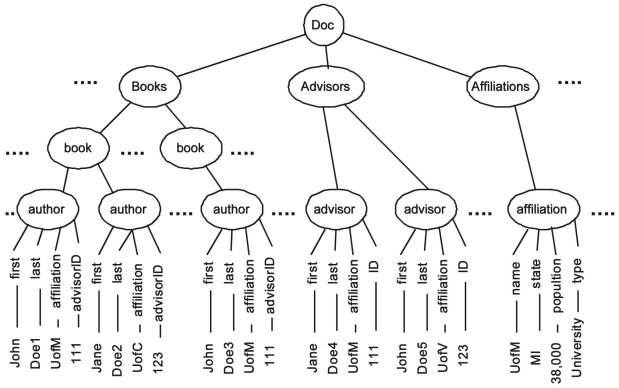


Fig. 2. A fraction of an XML document in tree format.

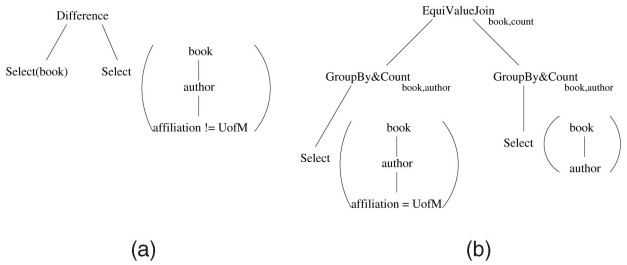


Fig. 3. Two plans that can be used to evaluate Universal Quantification query in Fig. 1. (a) A set difference plan, whereas (b) is a grouping by and counting plan.

This case is handled efficiently by relational databases, and the techniques used to evaluate it are discussed in [1], in which multiple algorithms are compared against each other. The winner is a hash-division algorithm that places all universities into a hash table and uses a bitmap with each author to find out whether or not the author is affiliated with all universities. For this class of queries, the relational technique carries over to XML in a straightforward way, and we have little new information to add. Hence, we will not further discuss this type of universal quantification or techniques used to evaluate it.

2.2 Before a Simple Predicate

When the word *every* occurs *before* a simple predicate in a Universal Quantification query, the query sounds like: Find all object1 that have *every* subobject2 satisfying a simple predicate. Example 1 discussed earlier belongs to this category. We begin with a description of known relational techniques.

2.2.1 Traditional Techniques

Logically, a Universal Quantification query can be converted into a set difference with negated condition. Continuing with Example 1, the query can be evaluated by performing a set difference between all books and books with at least one author *not* affiliated with the University of Michigan. This is different from the original query in that the Universal Quantification has been converted into Existential Quantification, which is more natural and easier to support using existing access methods. Another way of evaluating Universal Quantification is using aggregation and counting. The idea is to count all the authors of a book (without considering their affiliation) and compare the count with that of all authors of the same book that have an affiliation with the University of Michigan. If the two counts are equal, the book is part of the query result. Fig. 3a shows the set difference plan, whereas Fig. 3b shows the aggregation and counting plan.

The set difference plan may be efficient if the number of authors not affiliated with the University of Michigan is very small compared to the total number of authors. However, in our example, this is not likely to be the case. We believe that most of the time, the Universal Quantification query would have a condition that is very selective. Therefore, the negation of that condition will be true for most objects in the database, which renders the plan

inefficient. The aggregation and counting plan is obviously inefficient. In addition to performing two group by's and two pattern matches, it performs a value join to make sure that the counts of the same book are equal.

Both techniques described above can be easily adjusted to deal with XML (and, indeed, we do this to provide a baseline for our experiments later on). However, some characteristics of XML data make the problem more challenging. For example, XML data is in a tree form with complex relationships. In consequence, data materialization costs are higher than for relational tuples, and computing set difference becomes more expensive. Another issue present in XML is the possibility of nesting, in which elements can be nested in other elements of the same type. For example, in XML, a department element may be nested under another department element. This complicates matters because we need to consider both (or may be more) elements. As indicated in Section 5, performance of the two techniques is usually not very good on XML data, leaving us to seek a better method. We will discuss proposed algorithms for this case in Section 3.2.

2.3 Before a Complex Predicate

When the word *every* occurs *before* a complex predicate in a Universal Quantification query, the query sounds like: Find all object1 that have *every* subobject2 satisfying complex predicate involving object3. This predicate is considered complex because of the association with another object in the database. This object might be a result of another query.

We consider the example query 2, where every author of a book has to be checked to see if he/she is affiliated with a university that has the population subelement greater than 25,000. The university might be a result of another query that finds the names of universities with more than 25,000 students. Traditionally, this query, just like the previous flavor, can be answered by either using set difference or grouping and counting. The complex predicate in a query can be reduced to a simple predicate through full materialization, a prequery step.

To evaluate a query such as the one in Example 2, we can materialize the set of universities with more than 25,000 students and compare each author's affiliation with the list. An author with an affiliation belonging to this list passes. Hashing can be used to speed up things. This becomes similar to the second flavor, and the algorithm used there can be easily extended to include a hash table instead of a constant value. This will be discussed and evaluated in more detail later in the paper. This step can be performed either before the run of the query, where results are saved somewhere (similar as in Section 3.3), or it can be performed as the very first thing to be done when running the query (similar as in Section 3.4). In this paper, we will deal only with the case where the full materialization is done as the first step of the query run. Details of proposed algorithms for this flavor are presented in Sections 3.3 and 3.4.

2.4 Before a Correlational Predicate

When the word *every* occurs *before* a correlational predicate in a Universal Quantification query, the query sounds like: Find all object1 that have *every* subobject2 satisfying correlational predicate involving object3. This predicate is considered correlational because of the association with another object in the database that cannot be precomputed. It has to be figured out during the run of this query.

We now examine Example 3, where we need to find the advisor of the author, then make sure that their affiliations are equal. Again, this kind of query can be evaluated using traditional plans that contain set difference or grouping and counting. The plans are even more complicated and more time consuming than the ones used to evaluate complex predicate queries, because precomputation is no longer an option. This kind of query has a multicondition value join feel to it. The correlation condition has to be evaluated afresh for every candidate. We discuss the proposed algorithm for this flavor of universal quantification in Section 3.5.

3 Algorithms for Universal Quantification

In this section, we discuss various algorithms for different flavors of Universal Quantification queries for XML data. First, however, we introduce some necessary background on XML query processing. In particular, we introduce how to determine parent-child or ancestor-descendent containment relationships in an XML document.

3.1 Background on Node Labeling

In many XML database systems (for example, Niagara [2] and Timber [3]), each node in the database is labeled with a numeric start and end key (called "label" in Timber). This pair of keys defines an interval, which strictly includes the intervals of all descendants of the node. This enables identification of ancestor-descendant relationship between nodes. If every node is also labeled with its level, which indicates the nested depth of the node in the document, we can also find parent-child relationship. We can decide the relationship between two nodes according to the following formulas [4]:

- Ancestor-descendant relationship. A node (S_1, E_1, L_1) is an ancestor of node (S_2, E_2, L_2) iff $S_1 < S_2 \wedge E_1 > E_2$.
- Parent-child relationship. A node (S_1, E_1, L_1) is the parent of node (S_2, E_2, L_2) iff

$$S_1 < S_2 \wedge E_1 > E_2 \wedge L_1 = L_2 - 1.$$

(S_1 and S_2 are the start keys, E_1 and E_2 are end keys, and L_1 and L_2 are level labels in the above formulas.)

3.2 The Basic Algorithm XML-Univ-Quant-Simple

In this section, we describe the algorithm to handle the flavor where the *every* comes before a simple condition (that is, queries similar to the one in Example 1). We also discuss an improvement to it that involves a specialized index.

3.2.1 Description

To evaluate Universal Quantification, we devised a stack-based algorithm that follows along the steps of the structural joins introduced in [5]. In Fig. 4, we present the pseudocode of this algorithm. It takes as inputs, a list of nodes that are potentially query results, a chain pattern, a condition, and a relationship. We use Example 1 to explain

```
Algorithm XML-Univ-Quant-Simple (RList,P,C,rel)
/* RList contains lists of nodes to be returned in
ascending startKey order from upstream iterators.
RList[0] is the list of nodes that are possible results.
RList[i] (i=1 to m) are lists of nodes from the chain
patten.*/
/* P is a chain pattern. Each node in P is numbered
sequentially starting from root. Root number is 1. Each
edge in P is labeled parent-child or ancs-desc. The root
of P is to satisfy the EVERY part. Number of nodes in P
is m.*/
/* C is the universal quantification condition. This
condition is applied to the leaf node of P.*/
/* rel: an array storing relationship (either parent-child
or ancs-desc) between nodes in RList and root of P. rel[i]
is the relation between the i-th and (i+1)th node in P.*/
/* min: the number of stack with smallest start-key.*/
1. n0 = RList[0].first
2. for i=1 to m
3.     ni = List[i].next
4. while nm ≠ NULL
5.     min = i where ni.start-key is smallest and i=0..m
6.     pop-stacks(min)
7.     if min ≠ m
8.        if stacki is not empty for i=0..min-1 AND
          !(rel[min] is parent-child AND
          nmin.level+1 ≠ level of top node in stackmin+1)
9.           stackmin.push(nmin)
10.    else
11.       if satisfies-condition(min,C) AND
12.       stacki is not empty for i=0..m-1 AND
          !(rel[min] is parent-child AND
          nmin.level+1≠ level of top node in stackmin+1)
13.          mark all nodes in stack0 as potential output
14.       else
15.          pop-stacks(-1)
16.    nmin = RList[min].next
17.    if nmin = NULL AND min ≠ m AND stackmin is empty
18.       exit loop
19. if nm ≠ NULL
20.    for each element e in stack0
21.       if e is potential output
22.          add e to result
```

Fig. 4. Algorithm for simple predicate queries.

the inputs. The list of nodes is the list of books in the database. They could have passed some condition or pattern or they were simply read from an index. The pattern consists of two nodes, author and affiliation, where author is the root, and affiliation is the leaf of the pattern. The condition is that the value of affiliation is equal to "University of Michigan." Finally, the relationship between book and author is ancestor descendant. Note that with the *iterator model* [6], we actually pass iterators to lists of books, authors, and affiliations as arguments, with which we can fetch corresponding items in an ascending order of the start key when necessary (all from index scans). Therefore, essentially, the algorithm takes three lists of nodes as input.

The algorithm goes through each list of nodes and compares the start key of the first node on each list. By comparing the start and end keys, we can effectively discover ancestor-descendant or parent-child relationships between two nodes (as in Section 3.1). The node with the smallest start key is put in the corresponding stack. If the book has the smallest start key, it is placed in the book stack. If the author or affiliation comes first, the algorithm gets the next element from the list of the node with the smallest start key. It checks again for smallest start key. Assume that the author has the smallest start key now (and it does not pop any element), the algorithm checks the book stack. If the book stack has at least one element, the algorithm places the author in the author stack. Otherwise, it gets the next author since the current one is not a descendant of any book in the list. It checks again which node has the smallest start key. Assume that it is affiliation this time. The algorithm checks first if it has the value "University of Michigan." If it does, it marks elements in the book stack as potential results.

```
Procedure pop-stacks(num)
        return void
{
1. if num = -1
2.     for i=0 to m-1
3.        stacki.reset
4.     return
5. else
6.     for i=0 to m-1
7.        while (stacki is not empty AND
          stacki.top.end-key < num.start-key)
8.           popped-node = stacki.pop()
9.           if popped-node is potential output
10.             add popped-node to result
}
```

Fig. 5. Routine `pop-stacks` used in the algorithm in Fig. 4.

Otherwise, it has found an author with an affiliation that is not the "University of Michigan," and the algorithm immediately ignores all nodes in all stacks because they cannot be part of the output. In Fig. 4, the relation between two nodes could be ancestor descendant or parent child, and the relation is checked using the formula in Section 3.1.

In the algorithm in Fig. 4, we use a procedure `pop-stacks` shown in Fig. 5. The procedure can pop all stacks (given argument -1), or all nodes that are not an ancestor of the node with the smallest start key. If popped nodes are potential results (for example, book nodes that satisfy all conditions so far), they are added to the result.

3.2.2 Improved Algorithm

The algorithm proposed can run more efficiently if we have an index jointly built on the element tag and start key. The index enables more efficient processing of input lists in the sense that some nodes are skipped even before checking the start key or putting in the stack. To be more specific, once we decide that a book is not part of the result, we can skip all authors and affiliations (from input lists) that belong to it and start with authors and affiliations that actually belong to the next book by using its start key as an input to the index scan along with the author or affiliation tag. The index has been originally proposed in [7] to speed up structural joins. The use of such an index entails a slight modification to the algorithm. In line 15, after the stack is popped, a new node from `RList` is read and nodes read from the pattern are skipped based on the start key of the new node. Only nodes that are potential descendants of the new node are read. We save here because we are not reading all the input nodes in the pattern anymore. Instead, we are reading only potential matches. Also, if a node is not part of the output, we need not worry about reading its descendants.

3.3 Full Materialization for Complex Predicates

We now turn to the case where we have the *every* before a complex condition. The straightforward solution is to materialize the set in the condition beforehand. To achieve this, a few changes are required to the algorithm in Fig. 4.

1. `C` is no longer a simple condition consisting of an operation (equal, less than, and so forth) and a constant value. Instead, it is a subquery that will return a set of nodes in the XML document. In Example 2, `C` should be a query that returns universities with more than 25,000 students.

```
Procedure satisfies-condition(min, C, T)
        return bool
/* min is the node we want to make sure satisfies
condition C*/
/* C is a sub-query that results in a set of nodes
sorted by value to be compared to min in ascending
order.*/
/* T is a hash table passed from original algorithm.
T is empty the first time this procedure is evoked.*/
{
1. if (T is empty)
2.      curr = get next of C
/* curr is a global variable*/
3. else
4.      m = get first match of min from T
5.      if (m != NULL) return true
6. while (curr != NULL AND min < curr)
7.      insert curr in T
8.      curr = get next of C
9. if (min == curr)
10.     return true
11. else
12.     return false
}
```

Fig. 6. Routine `satisfies-condition` used in the algorithm in Fig. 4. This routine is used in the algorithm to evaluate the complex predicate.

```
Procedure satisfies-condition(min, C, min-ancs)
        return bool
/* min is the node we want to make sure it
satisfies condition C*/
/* C is a sub-query that results in a set of nodes
related to min-ancs*/
/* min-ancs is an ancestor of min that is referenced in
query of C*/
{
1. m = get first match of min-ancs from LT
/* LT is a global hashing table mapping a min-ancs
to a true or false value*/
2. if (m != NULL)
3.      return m
4. pass min-ancs to C
5. curr = get next of C
/* curr is a local variable*/
6. while (curr != NULL AND min != curr)
7.      curr = get next of C
8. if (min == curr)
9.      insert (min-ancs,true) in LT
10.     return true
11. else
12.     insert (min-ancs,false) in LT
13.     return false
}
```

Fig. 7. Routine `satisfies-condition` used in the algorithm in Fig. 4. This routine is used in the algorithm to evaluate correlational predicates.

2. Materialization is the first step, in which the condition `C` is used to establish a set of nodes `s`.
3. The second step should be the hash of the set `s` based on the value to be compared against. For example, if we are checking for affiliation, then the name of the organization is the value we hash on. Note that this step can be merged with the previous so we can hash as we get results. The result of this step is a hash table `T`.

   The first two steps are added before the first line of the algorithm in Fig. 4.
4. In line 11, `satisfies-condition` routine should take in an extra parameter `T`. Now, there are three cases. In the first and usual case, the result of the condition `C` is a set of nodes, and the universally quantified relation is equality. In such a case, hashing can be used to advantage. Instead of checking `min` against a constant value in the condition `C`, the hashing function is applied to `min` to get matching entries in `T`. If a match is found, `satisfies-condition` returns true. Otherwise, it returns false. In the second case, the precomputed condition just generates a single value and one could check equality or inequality relation with this value without using hashing. In the last case, the precomputed condition retrieves multiple values, and the universally quantified relation is inequality. Hashing cannot be used in this case, and we need to compare with values from `C` until we find that the relation is satisfied.

3.4 Progressive Evaluation for Complex Predicates

Sometimes, the size of the precomputed set of results of the subquery may be too large for the precomputation technique described above to be practical. In such a case and also in the case of correlational predicates that we will consider later, it is necessary to evaluate the complex predicate as we go. To implement this, the function `satisfies-condition` in the algorithm in Fig. 4 has to be extended from a simple predicate check to the procedure in Fig. 6.

Here, `C` is an independent subquery returning a set of nodes from the XML document. In Example 2, `C` would be a query that returns universities with more than 25,000 students. This is an independent query. Note that we require the results of this query to be returned in an ascending order of the value to be compared to `min`. Sorting is thus necessary as a preprocessing step if the results do not satisfy this condition. The first time the function `satisfies-condition` is evoked, the first result of `C` is read. With the iterator model, we only produce and read an item when needed. Later runs check for matches of `min` in the hashing table. If a match is found in the table, we are done, and `min` satisfies the complex predicate. If no match is found, we continue to produce and read results of the subquery `C` as long as these results are smaller than `min`. Each of these results is stored in the hash table to be compared against future `min`s. Once a match is found, we return true. If we reach a result that is greater than `min`, we stop and return false. Next time, `min` will be checked against the hash table first, then against `curr`, and so on. Compared to full materialization, the progressive evaluation algorithm not only saves temporary storage but also avoids computing some results for the condition query. In Example 2, instead of precomputing all universities with more than 25,000 students and saving the results, we only fetch on-demand universities with which the author of `min` is affiliated.

3.5 Evaluating Correlational Predicates

This extension is used to handle the cases where the *every* is before a correlational condition. To implement this extension, we redefine in Fig. 7 the function `satisfies-condition`.

Again, `C` is more complex than in the original algorithm. It is still a subquery. However, in this extension, it is NOT an independent query. `C` has a reference to a node in the original query. In Example 3, `C` would be a query returning affiliations of advisors of an author *A*. This author is part of the original query. In the function in Fig. 7, this author is `min-ancs` and his affiliation is `min`. The main idea in this extension is to evaluate the subquery for a node at most once. We do that by saving a hashing table `LT` that maps a node to a bit identifying whether or not it satisfied the subquery. The first step in this method is to check whether we encountered this node before. If we have, then we get the result and exit. If

this is the first time we encounter `min-ancs`, we submit it to the query `C` and get the results of the query one by one. Once we find a match to `min`, we insert `min-ancs` and true in `LT`. If we do not find a match, we insert `min-ancs` and false. Therefore, next time the same `min-ancs` comes, we need not evaluate the potentially expensive query `C`. Instead, we look it up in `LT`. `min-ancs` is passed from the original algorithm by getting the member of the stack-holding nodes that are referenced in the subquery. In our example, `min-ancs` is the node in the stack-holding authors. Our algorithm considers the case where there is no nesting in the node referenced by the subquery. If there is nesting (for example, there is another author nested under an author), query results vary according to the specification of the query. One could require all or some nested authors to satisfy the subquery, and this gives totally different query results. Suppose we have at most n nodes under a node, then we could have $O(n^2)$ different specifications for query results. Hence, we provide a base algorithm here and leave the users to specify what the query should return.

3.6 Multiple Quantifiers

Up until now, we have been discussing techniques to deal with *one* universal quantifier per query. In this section, we consider what to do when there are multiple universal quantifiers within a single query expression. There are two cases that are handled in quite distinct ways: the first is when one quantifier occurs "below" another, the second is when this is not the case. We call the former a *chain pattern* and the latter a *twig pattern*.

3.6.1 Chain Pattern

We consider the following example.

Example 5. Find books that have *every* chapter have *every* section have *every* paragraph contain the word "XML."

Now, we need to make sure that for a section, all paragraphs have the word "XML," and then, for each chapter, we need to make sure that all sections pass the paragraph condition. Then, for each book, all chapters should pass the section condition. This is easily accomplished by having three Universal Quantification access methods placed right after each other in the query evaluation plan.

Indeed, this simple chaining of access methods is sufficient for such chain patterns of universal quantifiers.

However, there is scope for optimization, if the schema is known. In the example here, suppose that the only way in which a paragraph node can be a descendant node of a book element is through chapter and section. In that case, the chain can be collapsed to the last node. A single universal quantification over paragraph descendants of book is equivalent to the chain of quantifications specified. Thus, we could write the example query as

Find books that have every paragraph contain the word "XML."

This is a simple query with a single universal quantification and can be evaluated using techniques previously described.

3.6.2 Twig Pattern

The universal quantifiers may not occur in a nested chain pattern. Instead, they could occur in "sibling" locations (possibly at different levels), creating a *twig pattern.* Here, is an example:

Example 6. Find books with every author having first name "John" and every chapter having five sections.

A way of dealing with this is to treat the two conditions as two separate Universal Quantification operators and then intersect the results. In other words, independently find books satisfying the condition on author and books satisfying the condition on chapter and then the return books in common between these two lists. This sort of computation strategy is reasonable for index-based selection conditions that are inexpensive to evaluate. However, for expensive quantifier-based conditions, a great deal of work in creating these two lists can be avoided if both conditions were evaluated jointly.

This suggests modifying our algorithm in Fig. 4 to deal with multiple *every* clauses breadthwise. Our algorithm assumes that we do not have nested nodes (for example, book/book) in the document and wild card (for example, book/*/) in the query at the same time. The following is a step-by-step explanation of how the modified algorithm works.

1. Instead of having a chain of stacks, the algorithm will have a chain of stacks followed by a branching and multiple chains of stacks. This is needed to accommodate the branching in the actual pattern. We will use Example 6 to explain this point. The main chain of stacks consists of one stack that holds books. Then, a branching occurs and two parallel chains of stacks are needed. One holds authors and first names and the other holds chapters.
2. We keep reading in nodes from the lists and keep them in their prospective stacks. Note that popping a node from a stack in a branch does not affect the stack in the other branch, but it may affect the stack in the main chain. Popping a node from a main chain stack does not make any branch empty its stacks. To continue with our example, as we read in nodes, if an author pops another author, the chapter's stack is not affected. If the author pops nodes from the book stack, the chapter stack stays intact. Nodes from a stack is popped through the procedure `pop-stacks`.
3. If a leaf node is read, then it is checked against the condition. If it returns true, the nodes in the root stack are marked as potential output for this particular branch. If it returns false, the root stack is popped and nodes in it are not output. This behavior is exactly like the behavior of the algorithm in Fig. 4 except that in the original algorithm, we had one flag. In the modified version, we have a flag for each branch. Only if all the flags were true, then the root node is considered output. In our example, if a first name (it is a leaf) is encountered, it is checked to see if it is equal to "John." If it is not, then all the books in the root stack are immediately discarded. If it turns out that the first name is actually equal to

```
Algorithm XML-Univ-Quant-Twig (NList,P,C,rel)
/* NList contains lists of nodes to be returned in
ascending order of start-key from upstream iterators.
NList[i][j] (i=0 to n, j=0 to Li) is the list of
nodes from the j-th node in chain patten i, where Li is
the length of chain pattern i. NList[n][0] is the list
of nodes that are possible results. */
/* P[i]: Chain pattern. Each node in P[i] is numbered
sequentially starting from root. Each edge in P[i] is
labeled parent-child or ancs-desc. The root of P[i] is
to satisfy the EVERY part. Number of nodes in P[i] is
Li. */
/* C: Array of universal quantification conditions.
Condition C[i] is applied to the leaf node of P[i]. */
/* rel: Array storing relationship (either parent-child
or ancs-desc) between nodes in NList and root of P[i].
rel[i][j] is the relation between the j-th and (j+1)th
node in P[i]. */
/* X: index of chain pattern with smallest start-key. */
/* Y: index of stack with smallest start-key in chain
pattern X. */
/* stack: stack[i][j] is the stack for node j in chain
pattern i. stack[n][0] is the root stack.*/

1. for i = 0 to n,
2.     for j = 0 to Li
3.        nk = NList[i][j].next; k++
4. while nmin ≠ NULL
5.     nmin is the node with smallest start-key in all
       lists. Denote the list as NList[X][Y]
6.     pop-stacks[X, Y]
7.     if Y ≠ L[X] - 1
8.        if stack[X][j] is not empty for j=0..Y-1 AND
          !(rel[X][Y] is parent-child AND
          nmin.level+1≠level of top node in stack[X][Y+1])
9.           stack[X][Y].push(nmin)
10.    else
11.       if satisfies-condition(X, Y,C[X]) AND
          stack[X][i] is not empty for i=0..Y-1 AND
          !(rel[X][Y] is parent-child AND
          nmin.level+1≠level of top node in stack[X][Y+1])
12.          turn on flag of stack[X][0]
13.       else
14.          pop-stacks(-1, 0)
15.    nmin = next node with smallest start-key,
       corresonding to NList[X][Y]
16.    if nmin = NULL AND Y ≠ L[X]-1
       AND stack[X][Y] is empty
17.       exit loop
18. if nmin ≠ NULL
19.    for each element e in stack[n][0]
20.       if all flags are on for e
21.          add e to result
```

Fig. 8. Algorithm for twig pattern queries.

"John," then the author flag in all the books in the root stack is turned on.

4. Just like in the original algorithm, when it is time to pop the root stack, the flags of each node are checked. If all are true, this node is part of the output. Otherwise, the node is discarded. In our example, when it is time to pop a specific book, the author flag and the chapter flag are checked. If both are true, this book is in the output. Otherwise, it is not in the output.

The pseudocode of this algorithms is shown in Fig. 8. Procedure `pop-stacks` is updated and shown in Fig. 9.

3.6.3 Complex Patterns

In general, in a large XQuery expression, there could be several universal quantifiers, bearing various relationships to one another. Any pair of quantifiers must either form a chain pattern or twig pattern. Therefore, we can solve the entire complex expression through this simple algorithm:

1. Form a pattern tree from the quantifiers of the query. This pattern tree should have at least one quantifier in each of its paths.
2. Label each node as root, every, sec-every, or none. Only the root of the pattern is labeled *root*. *every* is the label given to the nodes in the pattern that are preceded in the query by *every*. If we have two *every*s in a row, the deeper one is labeled *sec-every*. The rest of the nodes in the pattern are not labeled.
3. Apply the recursive algorithm in Fig. 10 to the labeled pattern tree starting from its root. The algorithm will build a universal quantification evaluation tree that is a part of the evaluation tree of the whole query. It starts from the root and handles the pattern tree node by node in a left-to-right depth-first manner. For each node, it checks to see if it is a leaf, a node in a path, or a branching. If it is a leaf (lines 2 and 3), the current node is returned. If it is a node a path (lines 4-6), then it is one of four cases; if it is the root of the whole tree, it is labeled *sec-every*, it is labeled *every*, or it is not labeled. If it is one of the two first cases (lines 5-6), then the algorithm handles the child of this node and feeds it to a chain quantifier (our proposed algorithm in Section 3.2 or any of its extensions) that is added to the plan. If it is one of the two last cases (lines 7 and 8), the algorithm handles the child (passes the child to the algorithm) and returns the result. If the node is a branching (lines 9-12), then the algorithm handles each of the children (by passing them to the same algorithm) and feeds the results to a twig quantifier (proposed in the previous section).
4. Feed the output universal quantification evaluation plan tree to the appropriate access method in the

```
Procedure pop-stacks(X, Y)
       return void
{
1. if X = -1
2.     reset all stacks
3. else
4.     for all stacks stack[i][j] (i=0..n-1, j=0..L[i]-1)
5.        while stack[i][j] is not empty AND
          stack[i][j].top.end-key <
          stack[X][Y].top.start-key
6.              popped-node = stack[i][j].pop()
7.     while stack[n][0] is not empty AND
       stack[n][0].top.end-key <
       stack[X][Y].top.start-key
8.        popped-node = stack[n][0].pop()
9.        if all flags are on for stack[k][0], where k =
          0..n-1
10.          add popped-node to result
}
```

Fig. 9. Routine `pop-stacks` used in the algorithm in Fig. 8.

```
function build-UQ-eval-plan(curr-node)
       returns pointer to evaluation tree node
{
1.     let n be number of children of curr-node
2.     if (n == 0)
3.        return curr-node
4.     else if (n == 1)
5.        if (curr-node.label == sec-every OR root)
6.           return chain-univ-quant-node(curr-node,
             build-UQ-eval-plan(child))
7.        else
8.           return build-UQ-eval-plan(child)
9.     else
10.       for each child 0..n-1
11.          children[i] = build-UQ-eval-plan(child)
12.       return twig-univ-quant(curr-node,children)
}
```

Fig. 10. Algorithm that builds a universal quantification evaluation tree based on the multiple quantifiers in the query.

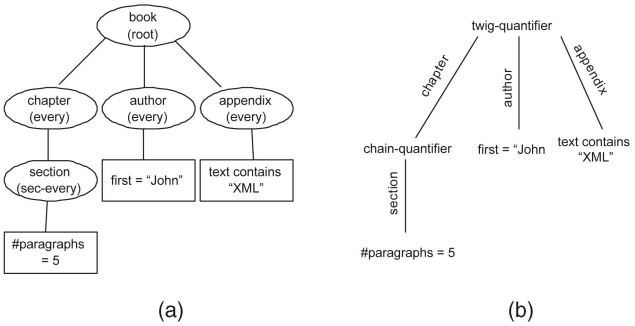


Fig. 11. (a) Labeled pattern tree of Example 7 and (b) the corresponding evaluation plan tree.

original evaluation plan (that contains both quantifiers and nonquantifiers).

To explain how the algorithm in Fig. 10 works, consider the following query:

Example 7. Find books where every chapter that has every section with at least five paragraphs and every author has the first name "John." These books should also have every appendix's text containing the word "XML."

We first convert the quantifier part of our query into the labeled pattern tree in Fig. 11a. Now, we start from `book` node and feed it to the algorithm. It has three children. This means that we pass each of them to the algorithm. After we get their results, we pass them to a twig quantifier node in our evaluation plan. We first pass, the first child, `chapter`. It has one child, which is also labeled *every*. This means its child, `section`, is passed to the algorithm. `section` is labeled *sec-every*, and it has one child. This means that we pass its child to the algorithm, and the result is fed into a chain quantifier. The child is a leaf node and thus we return. Now, we handle the second child of `book` and so on. The resulting evaluation plan is in Fig. 11b. In this figure, intermediate nodes of the evaluation tree are universal quantifiers, tree edges are entities, and leaf nodes are conditions.

4 Analysis of the Proposed Algorithms

In this section, we analytically study the performance of the proposed algorithms. We begin with the analysis of the XML-Univ-Quant-Simple.

4.1 Basic Algorithm

4.1.1 Time Complexity

Given as input lists of nodes sorted by their start key, the algorithm examines each node from a list exactly once. Denote cardinality of input and output as $|InputLists|$ and $|OutputList|$, respectively. Also denote the length of the chain pattern as $|P|$. In the pseudocode in Fig. 4, each node is put in the stack at most once and gets popped once and, therefore, the cost of all nodes push and pop is $O(|InputLists| + |OutputList|)$. We now examine cost of other operations. Finding `min` from multiple input lists (line 5) can be done in constant time after the first time if we keep the order of top nodes from each list. Checking whether the stacks are empty (lines 8 and 12) needs at most $|P|$ units of time, and it is performed for $O|InputLists|$ times in the worst case. Each line of the rest of the code takes constant time, except line 13. If the nodes to be returned (for example, books in Example 1) has no nesting in the document, this line has constant complexity since there is at most one node in $\texttt{stack}_0$. If there is nesting, it appears that the time complexity of line 13 depends on the nesting and the selectivity of lines 11-12. A clever implementation of line 13 operates $\texttt{stack}_0$ as an entity, meaning that we mark the status of the whole stack instead of individual nodes, which gives constant time complexity. It is apparent from the pseudocode that this implementation does not change the algorithm. Thus, the worst-case time complexity is $O(|P| \times |InputLists| + |OutputList|)$.

4.1.2 I/O Complexity

Each input list is read once, and the result is the output after being computed. Similar as that in [5], it is reasonable to assume that stacks fits in the memory at all times. The expected I/O cost is $O(\frac{|InputLists|}{B} + \frac{|OutputList|}{B})$, where B is the blocking factor.

4.2 Algorithms for Complex Predicates

The difference between algorithms for complex predicates and simple predicates is the cost of computing and preprocessing the results of the subquery. We assume the time cost of computing query `C` to be T_C and the cardinality of its result to be $|C|$. If the results are already sorted in the required order (as in Section 3.3), the total time cost is $O(|P| \times |InputLists| + |OutputList| + T_C)$. Otherwise, the cost is

$$O(|P| \times |InputLists| + |OutputList| + T_C + |C|log(|C|)),$$

where the last item is the time for sorting (we assume in-memory sorting for the sake of simplicity). Assume the I/O cost of the subquery to be IO_C. The total I/O now is $O(\frac{|InputLists|}{B} + \frac{|OutputList|}{B} + IO_C)$, where B is the blocking factor.

4.3 Algorithms for Correlational Predicates

The only difference with the basic algorithm is in the function `satisfies-condition`, which no longer takes constant time. Note that this function is called only for the last node in the pattern chain (for example, "affiliation" in Example 3). We continue to use the notations in Fig. 7 here. In the worst case, we compute query `C` for every `min-ancs` once. Denote the average time and I/O cost for this computation as T_C and P_C, respectively. If the cardinality of node `min-ancs` is $|\texttt{min} - \texttt{ancs}|$, the additional time and I/O cost compared to the basic algorithm is $O(T_C \times |\texttt{min} - \texttt{ancs}|)$ and $O(P_C \times |\texttt{min} - \texttt{ancs}|)$, respectively.

4.4 Algorithms for Multiple Quantifiers

4.4.1 Chain Pattern

It is easy to analyze the straightforward algorithm of applying consecutive universal quantification operators. For each operator, the cost is the same as the basic algorithm (Section 4.1), except that we do not need to output intermediate results.

4.4.2 Twig Pattern

The difference between twig pattern and the simple predicate is that now we have one main chain pattern and multiple branch patterns to manage. The cost of managing twig pattern can be analyzed in a similar way as in the case of simple predicate. We denote the length of the main chain pattern to be L_{main} and the cardinality of its input lists as $|InputLists_{main}|$. We assume there are n branches. For branch $i\ (i = 1, 2, \ldots, n)$, we denote its pattern length to be L_i and cardinality of its input lists to be $|InputLists_i|$. The total time complexity is thus the sum of time cost for each individual chain and output. As shown in Section 4.1, the worst-case time complexity for the simple case is $O(|P| \times |InputLists| + |OutputList|)$. Adding the cost for all branches, we have

$$O\left(L_{main} \times |InputLists_{main}| + \sum_{i=1}^{n}(L_i \times |InputLists_i|)\right) + |OutputList|.$$

The I/O complexity is the same as the basic algorithm, which is $O(\frac{|InputLists|}{B} + \frac{|OutputList|}{B})$, where B is the blocking factor.

5 Experimental Evaluation

We experimentally evaluated the techniques that we developed. We split the experiments into three groups: simple predicate queries, complex predicate queries, and correlational predicate queries. In each group, we ran multiple queries and evaluated the new techniques versus the two traditional ways of evaluating the query, namely, set difference and counting (recall Section 2.2.1, where these techniques were outlined).

5.1 Experimental Testbed and Workload

We ran the experiments using TIMBER [3]. Experiments were run on a 1.6-GHz Centrino machine with 1 Gbyte of RAM running WindowsXP. Coding was done using Microsoft Visual C++ .NET. Each experiment was run five times. The lowest and highest readings were ignored, and the remaining three were averaged.

Our primary workload consisted of three data sets:

- **Sigmod Record** [8]. This XML file consists of a group of issues. Each issue contains a bunch of articles where each article has a title, page numbers, and a list of authors. The size of the replicated XML file is approximately 102 Mbytes. Loaded in Timber, the data size was approximately 470 Mbytes.
- **XMark** [9]. This XML file consists, among other things, of a group of items in an auction. Each item has a list of categories and a list of descriptions. The size of this XML file was approximately 113 Mbytes. Loaded in Timber, the data size was approximately 360 Mbytes.
- **Book**. This is a synthetic data set, especially generated for chain pattern queries. It contains nine levels: collection, bookSet, book, chapter, section, subSection, subSubSection, paragraph, and sentence. Each sentence contains 15 words randomly drawn from a dictionary. From the dictionary, we picked 10 words (for example, "ACM") that are much more likely to be chosen and, thus, our queries can return some results. This data set is 131 Mbytes on disk and 245 Mbytes loaded in Timber.

TABLE 1
Simple Predicate Queries

query	*Data Set*	*Plan 1*	*Plan 2*	*New Alg.*	*Tag-ID Ext.*
S1	SIGMOD	153.21	155.88	70.46	55.64
S2	SIGMOD	150.97	153.80	65.98	58.50
S3	SIGMOD	150.76	151.70	62.40	50.75
S4	XMark_S	17.21	19.03	8.35	6.45
S4	XMark	36.15	38.34	16.08	14.00
S4	XMark_L	55.28	58.11	25.16	22.31
S4	XMark_E	70.14	81.33	34.46	29.53
S5	XMark	44.81	47.64	23.89	19.09
S6	XMark	43.41	48.19	0.39	0.36
S7	XMark	46.91	48.95	0.46	0.45

In order to test the scalability of our proposed algorithms, we used three additional XMark data sets: $\mathbf{XMark}_S$ (the data file is 58 Mbytes; 170 Mbytes loaded in Timber), $\mathbf{XMark}_L$ (174 Mbytes file size; 523 Mbytes loaded in Timber), and $\mathbf{XMark}_E$ (233 Mbytes file size; 708 Mbytes loaded in Timber). We selectively ran some queries on these two data sets to show that our algorithms scale gracefully with file size.

The groups of queries we ran are presented in the Appendix.

5.2 Simple Predicate Queries

We compared the two traditional approaches to the new algorithm we developed, XML-Univ-Quant-Simple. Plan 1 represents the set difference approach, whereas Plan 2 represents the grouping by and counting approach. We also compared the new algorithm to the extension that uses a tag-ID index. We present the results in Table 1, where we show time (in seconds) measuring the performance of the new algorithm (XML-Univ-Quant-Simple) versus using set difference (Plan 1) versus using group by and counting (Plan 2). Also, the Tag-ID extension time is measured.

We ran seven simple predicate queries on the two primary data sets mentioned above. Query S4 is also run on the two additional XMark data sets for scalability test. The queries were written to cover a range of (intermediate and final result) selectivities.

Generally, the new algorithm savings are between 47 percent and 59 percent over the two traditional plans (the two traditional plans had very similar performance, with the counting-based plan consistently worse than the set difference plan, but by a very small amount). The first three queries have three different selectivities. S1 returns around 5 percent of the total articles in the database, whereas S2 and S3 return 2 percent and 0.1 percent, respectively. Given that the universal quantification queries are selective by nature and the structure of the SIGMOD data set, S1 is considered of low selectivity, S2 is of medium selectivity, and S3 is considered of high selectivity. As the

TABLE 2
Complex Predicate Queries

Query	*Data Set*	*Plan 1*	*Plan 2*	*Full Mater.*	*Prog. Eval.*
C1	XMark	403.62	457.93	83.16	77.57
C2	XMark	578.81	599.25	85.50	78.19
C3	XMark	256.14	373.59	40.00	29.70
C4	XMark	204.68	249.96	53.67	51.82
C5	XMark	164.49	199.35	39.13	29.23

query becomes more selective (returns less results), the savings of universal quantification plan steadily increase.

Even though query S4 has a low selectivity at 30 percent and S5 has a high selectivity at 0.6 percent, we notice that unlike the behavior of the first three queries, S5 performs slightly worse than S4. The reason for this is the long chain in query S5 (look for the exact query in the Appendix). S5 specifies a long path of three nodes that needs to be evaluated, whereas S4 has a single node path. For queries S6 and S7, the new algorithm performs around 100 times better than the two traditional approaches. The reason for this is the small size of the output and large size of the input. S6 and S7 are extremely selective at 0.03 percent and 0.3 percent, respectively. Plans 1 and 2 perform all the work and produce the small result, whereas the new algorithm skips lots of intermediate results once they are known to be not part of the final result.

The last column in the table shows the performance of the Universal Quantification using the tag-ID index. This index will allow the algorithm to even skip scanning intermediate results once we know that they are not part of the result (the original XML-Univ-Quant-Simple needs to scan and discard the intermediate results to reach the following set of results, whereas the tag-ID index allows direct access to the following set of results). The savings of the tag-ID extension over the new algorithm (XML-Univ-Quant-Simple) range between 11 percent and 21 percent (in the last two queries, S6 and S7, the runtime is too small (less than half a second total) for these savings to be apparent—constant time overheads probably dominate).

The performance on query S4 over data sets of different sizes shows that our algorithm scales up and down gracefully.

5.3 Complex Predicate Queries

To evaluate the two extensions developed to handle complex predicate queries, we ran a set of five queries on the XMark data set, as listed in the Appendix. The SIGMOD data set has too simple a schema to admit believable complex predicate queries and, hence, was not used in these experiments. In Table 2, we show the time (in seconds) measuring the performance of the two approaches to handling complex predicate queries (Full Materialization and Progressive Evaluation) versus using set difference (Plan 1) versus using group by and counting (Plan 2). In general, the full materialization plan performs better than Plans 1 and 2 by 73 percent to 89 percent. The reason for these big savings is that the new technique not only speeds up the structural join part of the query (by skipping unnecessary intermediate results) but also speeds up the value join part (where a join is performed between the Universal Quantification part and the complex predicate part of the query). The progressive evaluation plan speeds up things even more. The reason for this is that the later results of the right query side may be skipped by this plan and, therefore, it performs slightly better than the full materialization (it evaluates the whole right side). The savings of the progressive evaluation approach range between 3 percent and 26 percent over the full materialization. Although the savings of the same approach over Plans 1 and 2 range between 75 percent to 92 percent. In these five queries, we covered a wide range of selectivities. The first two queries, C1 and C2, asked for items, whereas the last three queries asked for open auctions. C1 has a selectivity of around 9 percent, whereas C2 has a selectivity of 1 percent. The reason for C2 to be performing worse than C1 even though it produces less results is that C2 searches for the word "condition" under all text elements under category elements. C1, on the other hand, searches for the word "good" under a single text element under description of a category. C3, C4, and C5 have selectivities of 60 percent, 35 percent, and 11 percent, respectively. In general, C3, C4, and C5 perform better than C1 and C2 because the total number of open auctions is about half the total number of items.

TABLE 3
Correlational Predicate Queries

Query	*Data Set*	*Plan 1*	*Plan 2*	*Co-Rel Ext.*
R1	XMark	625.89	641.28	375.05
R2	XMark	591.52	604.20	388.81
R3	XMark	413.15	423.81	291.18

5.4 Correlational Predicate Queries

To evaluate the extension developed to handle correlational predicate queries, we ran three queries on the XMark data set. We could not run more because the data set would not support more Universal Quantification correlational predicate queries. Moreover, again, the SIGMOD data set does not provide enough structure to run such queries. We compared the performance of the extension against the two traditional techniques, set difference (Plan 1) and grouping by and counting (Plan 2), and show the results in Table 3. The correlational extension plan performs better than Plans 1 and 2 by 30 percent to 42 percent. The reasons for these savings are the skipping of the nodes that have one false and the avoidance of redundant evaluation of the right-side query by keeping track of whether or not a node has previously passed the condition. Selectivities of R1, R2, and R3 are 2.6 percent, 0.5 percent, and 0.03 percent. Because of the nature of the correlational predicate queries, these selectivities are considered low, medium, and high, respectively.

5.5 Twig Pattern Queries

We run three queries (T1, T2, and T3) on the XMark data set for Twig pattern queries. T2 is run on XMark_S and XMark_L data sets for scalability test. The results are shown in

TABLE 4
Twig Pattern Queries

Query	*Data Set*	*Plan 1*	*Plan 2*	*Twig*
T1	XMark	68.23	70.52	22.21
T2	XMark	62.43	66.12	20.33
T2	XMark_S	33.25	35.44	11.49
T2	XMark_L	94.38	98.42	32.37
T3	XMark	829.32	842.51	412.30
T4	XMark	801.16	822.45	395.40

Table 4. Query T2 has a lower selectivity than T1. T3 contains a twig pattern and a correlational predicate. Our proposed algorithms achieve much better time performance than the other two plans. Table 4 shows our algorithm is both efficient and scalable.

5.6 Chain Pattern Queries

We run four chain pattern queries (P1, P2, P3, and P4) on the Book data set with increasing pattern length (see Table 5). All queries try to find books with every paragraphs having the word "ACM." All four queries in fact return the same set of results. P1 is the most efficient way to query the database if one knows the schema. P2, P3, and P4 each added universal quantification to upper levels of the document tree. As suggested in Section 4.4, we could directly push down the quantification to the lowest level ("paragraph" in this case) if we are aware of the schema. We consider this optimization future work, since there is much room to develop. Here, we use the simple approach of cascading the quantifiers and show the performance. We could see that time cost increases with the length of the chain, and indeed, there is much room to improve. Nonetheless, our algorithm still performs better than the baselines.

6 Related Work

In relational databases, the problem of evaluating Universal Quantification has been addressed. The earliest work done in this field is probably proposed in [10]. In this paper, the problem is addressed indirectly through optimizing general relational expressions. A naive algorithm that uses Cartesian product is proposed. Then, Carlis [11] proposes a relational algebra extension to evaluate Universal Quantification but does not discuss how the query engine would handle it. Rantzau et al. [12] surveys related algorithms and introduced methods for classifying input data and identifying the most efficient algorithm for such data. SQL extensions to express Universal Quantification are proposed in [13], [14].

For evaluating universally quantified queries in object-oriented and object-relational databases, Claussen et al. [15] adopt various evaluation plans from relational databases, including division, set difference, grouping with count aggregation, and antisemijoin. Antisemijoin, which is not discussed in this paper, is similar to set difference, but it requires that the attribute(s) in the universally quantified condition to be a (super) key of the object to be returned. In an object-oriented database (OODB) context, this is satisfied when the attribute(s) constitutes the object identifier. In general, this is often not true and, hence, we do not consider this plan. For relational databases, Bry [16] proposes rewriting rules for relational algebra to better evaluate quantified queries. Graefe and Cole [1] gives an overview of all techniques proposed to evaluate Universal Quantification and compares them to each other. The techniques used in [1] range from a naive sort-based algorithm to the use of aggregation (both hash-based and sorting-based) and counting to a more efficient hash-division algorithm. Our work differs from previous papers since we need different techniques and algorithms to efficiently handle the tree structure of XML documents.

TABLE 5
Chain Pattern Queries

Query	*Data Set*	*Plan 1*	*Plan 2*	*Chain - Cascade*
P1	Book	36.38	38.12	19.41
P2	Book	60.21	64.28	35.33
P3	Book	92.55	98.32	46.43
P4	Book	111.19	117.52	55.17

7 Conclusion

In this paper, we proposed a family of algorithms to handle time-consuming Universal Quantification queries. The algorithm XML-Univ-Quant-Simple is tailored to evaluate Universal Quantification queries with simple predicates imposed on XML documents. It uses stacks to keep track of structure. The proposed algorithm saves time by skipping intermediate results that would be otherwise computed if traditional access methods were used. We developed two extensions to the XML-Univ-Quant-Simple to handle Universal Quantification queries with complex predicates. One of these extensions materializes the complex condition result, and the other evaluates it as it goes. Both use hash tables to keep right side results. We developed a third extension to deal with Universal Quantification queries with correlational predicate. In this extension, we kept track of previous results to avoid redundant evaluations.

We also discussed different combinations of multiple *every* clauses in the query. We developed an algorithm to deal with multiple quantifiers breadthwise in a query. Also, to deal with multiple quantifiers in one query, we presented an algorithm to produce a partial evaluation plan that incorporates different chain and twig quantifiers.

We compared the performance of the proposed algorithms against that of previous proposals and found that the new algorithms out perform the old techniques by up to 100 times.

Appendix A

In this Appendix, we list the wording and the XQuery expressions of all the queries we ran in Section 5.

A.1 Simple Predicate Queries

- **S1.** Find articles with every author's name starting with "M."

```
FOR $a1 IN document ("sigmod.xml")//
article
WHERE EVERY $a2 IN $a1//author SATISFIES
```

```
starts-with ($a2//name,"M")
RETURN $a1
```

- **S2.** Find articles with every author's name starting with "C."

```
FOR $a1 IN document ("sigmod.xml")//
article
WHERE EVERY $a2 IN $a1//author SATISFIES
starts-with ($a2//name,"C")
RETURN $a1
```

- **S3.** Find articles with every author's name starting with "Q."

```
FOR $a1 IN document ("sigmod.xml")//
article
WHERE EVERY $a2 IN $a1//author SATISFIES
starts-with ($a2//name,"Q")
RETURN $a1
```

- **S4.** Find items with every incategory's id larger than "category400."

```
FOR $i IN document ("auction.xml")//item
WHERE EVERY $c IN $i//incategory
SATISFIES
$c/@category> "category400"
RETURN $i
```

- **S5.** Find items with every parlist//listitem's text containing the word "sold."

```
FOR $i IN document ("auction.xml")//item
WHERE EVERY $p IN $i//parlist SATISFIES
contains ($p//listitem/text,"sold")
RETURN $i
```

- **S6.** Find open auctions with every bidder's id larger than "person25000."

```
FOR $o IN document ("auction.xml")//
open_auction
WHERE EVERY $b IN $o//bidder SATISFIES
$b/@id> "person11000"
RETURN $o
```

- **S7.** Find categories where the description has every keyword starting with "l."

```
FOR $c IN document ("auction.xml")//
category
WHERE EVERY $d IN $c//description//
keyword SATISFIES
starts-with ($d,"l")
RETURN $c
```

A.2 Complex Predicate Queries

- **C1.** Find items with every in category having the word "good" in its description/text.

```
FOR $i IN document ("auction.xml")//item
FOR $c IN document ("auction.xml")//
category
WHERE EVERY $ic IN $i//incategory
SATISFIES
$ic/@category = $c/@id AND
contains ($c//description/text,"good")
RETURN $i
```

- **C2.** Find items with every in category having the word "condition" in their text.

```
FOR $i IN document ("auction.xml")//item
FOR $c IN document ("auction.xml")//
category
WHERE EVERY $ic IN $i//incategory
SATISFIES
$ic/@category = $c/@id AND
contains ($c//text,"condition")
RETURN $i
```

- **C3.** Find open auctions with every bidder's name starting with "S."

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p IN document ("auction.xml")//person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p/@id AND
starts-with ($p/name,"S")
RETURN $o
```

- **C4.** Find open auctions with every bidder's gender being "female."

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p IN document ("auction.xml")//person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p/@id AND
$p/gender = "female"
RETURN $o
```

- **C5.** Find open auctions with every bidder's city starting with "L."

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p IN document("auction.xml")//person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p/@id AND
starts-with ($p/address/city,"L")
RETURN $o
```

A.3 Correlational Predicate Queries

- **R1.** Find open auctions with every bidder having the same gender as the seller of the auction.

```
FOR $o IN document ("auction.xml")/
open_auction
FOR $p1 IN document ("auction.xml")//
person
FOR $p2 IN document ("auction.xml")//
person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p1/@id AND
$o//seller/@person = $p2/@id AND
$p1/profile/gender = $p2/profile/gender
RETURN $o
```

- **R2.** Find open auctions with every bidder living in the same city as the seller of the auction.

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p1 IN document ("auction.xml")//
person
FOR $p2 IN document ("auction.xml")//
person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p1/@id AND
$o//seller/@person = $p2/@id AND
$p1/address/city = $p2/address/city
```

```
RETURN $o
```

- **R3.** Find open auctions with every bidder at the same age as the seller of the auction.

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p1 IN document ("auction.xml")//
person
FOR $p2 IN document ("auction.xml")//
person
WHERE EVERY $b IN $o//bidder SATISFIES
$b/personref/@person = $p1/@id AND
$o//seller/@person = $p2/@id AND
$p1/profile/age = $p2/profile/age
RETURN $o
```

A.4 Twig Pattern Queries

- **T1.** Find items with every parlist//listitem's text containing the word "sold" and every in-category's ID larger than "category400."

```
FOR $i IN document ("auction.xml")//item
WHERE EVERY $c IN $i//incategory, $p IN
$i//parlist SATISFIES
$c/@category> "category400" AND
contains ($p//listitem/text,"sold")
RETURN $i
```

- **T2.** Find items with every parlist//listitem's text containing the word "sold" and every in-category's ID larger than "category900."

```
FOR $i IN document ("auction.xml")//item
WHERE EVERY $c IN $i//incategory, $p IN
$i//parlist SATISFIES
$c/@category> "category900" AND
contains ($p//listitem/text,"sold")
RETURN $i
```

- **T3.** Find open auctions with every bidder having the same gender as the seller of the auction and every keyword in annotation description starting with "l."

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p1 IN document ("auction.xml")//
person
FOR $p2 IN document ("auction.xml")//
person
WHERE EVERY $b IN $o//bidder, $d IN $o//
parlist//keyword SATISFIES
$b/personref/@person = $p1/@id AND
$o//seller/@person = $p2/@id AND
$p1/profile/gender = $p2/profile/gender
AND
starts-with ($d,"l")
RETURN $o
```

- **T4.** Find open auctions with every bidder living in the same city as the seller of the auction and every keyword in annotation description starting with "l."

```
FOR $o IN document ("auction.xml")//
open_auction
FOR $p1 IN document ("auction.xml")//
person
FOR $p2 IN document ("auction.xml")//
person
WHERE EVERY $b IN $o//bidder, $d IN $o//
parlist//keyword SATISFIES
$b/personref/@person = $p1/@id AND
$o//seller/@person = $p2/@id AND
$p1/address/city = $p2/address/city AND
starts-with ($d,"l")
RETURN $o
```

A.5 Chain Pattern Queries

- **P1.** Find books that have every paragraph contain the word "ACM."

```
FOR $b IN document ("book.xml")//book
WHERE EVERY $p IN $b//paragraph SATISFIES
contains ($p, "ACM")
RETURN $b
```

- **P2.** Find books that have every subSubsection and every paragraph contain the word "ACM."

```
FOR $b IN document ("book.xml")//book
WHERE EVERY $sSS IN $b//subSubSection, $p
IN $b//paragraph SATISFIES
contains ($sSS, "ACM") AND
contains ($p, "ACM")
RETURN $b
```

- **P3.** Find books that have every subsection, every subSubsection, and every paragraph containing the word "ACM."

```
FOR $b IN document ("book.xml")//book
WHERE EVERY $sS IN $b//subsection, $sSS IN
$b//subSubSection,
$p IN $b//paragraph SATISFIES
contains ($sS, "ACM") AND
contains ($sSS, "ACM") AND
contains ($p, "ACM")
RETURN $b
```

- **P4.** Find books that have every section, every subsection, every subSubsection, and every paragraph containing the word "ACM."

```
FOR $b IN document ("book.xml")//book
WHERE EVERY $s IN
$b section, $sS in $b//subsection,
$sSS IN $b//subSubSection, $p IN
$b//paragraph SATISFIES
contains ($s, "ACM") AND
contains ($sS, "ACM") AND
contains ($sSS, "ACM") AND
contains ($p, "ACM")
RETURN $b
```

ACKNOWLEDGMENTS

This study was supported in part by US National Science Foundation Grant IIS-0219513.

REFERENCES

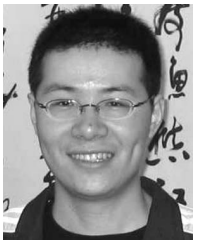
[1] G. Graefe and R.L. Cole, "Fast Algorithms for Universal Quantification in Large Databases," *ACM Trans. Database Systems,* vol. 20, no. 2, pp. 187-236, 1995.

[2] Univ. of Wisconsin, The Niagara System, http://www.cs.wisc.edu/niagara/, 2006.

[3] Univ. of Michigan, The Timber System, http://www.eecs.umich.edu/db/timber/, 2006.

[4] 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 J.,* vol. 11, no. 4, pp. 274-291, 2002.
[5] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, and D. Srivastava, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. Int'l Conf. Data Eng.,* p. 141, 2002.
[6] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, no. 2, pp. 73-169, 1993.
[7] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," *Proc. Int'l Conf. Very Large Data Bases,* vol. 2, pp. 263-274, 2002.
[8] Sigmod Record—XML Version, http://www.dia.uniroma3.it/Araneus/Sigmod/Record/HomePage/index.xml, 2005.
[9] The XML Benchmark Project, http://www.xml-benchmark.org, 2006.
[10] J.M. Smith and P.Y.-T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Comm. ACM,* vol. 18, no. 10, pp. 568-579, 1975.
[11] J.V. Carlis, "HAS, a Relational Algebra Operator or Divide Is Not Enough to Conquer," *Proc. Int'l Conf. Data Eng.,* pp. 254-261, 1986.
[12] R. Rantzau, L. Shapiro, B. Mitschang, and Q. Wang, "Universal Quantification in Relational Databases: A Classification of Data and Algorithms," *Proc. Int'l Conf. Extending Database Technology,* pp. 445-463, 2002.
[13] K.-Y. Whang, A. Malhotra, G.H. Sockut, and L.M. Burns, "Supporting Universal Quantification in a Two-Dimensional Database Query Language," *Proc. Int'l Conf. Data Eng.,* pp. 68-75, 1990.
[14] P.-Y. Hsu and J. Douglas Stott Parker, "Improving SQL with Generalized Quantifiers" *Proc. Int'l Conf. Data Eng.,* pp. 298-305, 1995.
[15] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner, "Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases," *Proc. Int'l Conf. Very Large Data Bases,* pp. 286-295, 1997.
[16] F. Bry, "Logical Rewritings for Improving the Evaluation of Quantified Queries," *Proc. Symp. Math. Fundamentals of Database Systems,* pp. 100-116, 1989.

Shurug Al-Khalifa received the PhD degree in computer science from the University of Michigan, Ann Arbor, in 2005, where she was a founding member of the Timber native XML database project. She is an assistant professor at King Saud University, Saudi Arabia. Her research interests include database systems and XML query processing.



Ben B. Liu received the BEng (first class honors) degree in computer engineering and the MPhil degree in computer science from the Hong Kong University of Science and Technology in 2002 and 2005, respectively. He is currently working toward the PhD degree in the Computer Science and Engineering Division, University of Michigan, Ann Arbor, which is affiliated with the Database Group. His research interests include data integration and database usability.

H.V. Jagadish received the PhD degree from Stanford University in 1985. He is a professor of computer science and engineering at the University of Michigan, Ann Arbor. He spent more than a decade at AT&T Bell Laboratories, Murray Hill, New Jersey, eventually becoming the head of the AT&T Labs Database Research Department, Shannon Laboratory. He is well known for his broad-ranging research on databases and has more than 80 major papers and 20 patents. He is a fellow of the ACM, a trustee of Very Large Databases (VLDB). Among many professional positions that he has held, he has previously been an associate editor of the *Transactions on Database Systems* from 1992 to 1995, the program chair of the ACM SIGMOD Annual Conference in 1996, and the program chair of the International Conference on Intelligent Systems for Molecular Biology (ISMB) in 2005.