Deep Set Operators for XQuery

Bo Luo
Penn State / IST
bluo@ist.psu.edu

Dongwon Lee
Penn State / IST
dongwon@psu.edu

Wang-Chien Lee
Penn State / CSE
wlee@cse.psu.edu

Peng Liu
Penn State / IST
pliu@ist.psu.edu

ABSTRACT

There are three *set operators* defined in XQuery, namely `union`, `intersect` and `except`. They take node sequences as operands, in which each node is identified by its node-ID and treated as an atomic entity. However, according to XML semantics, each node is "a set of set(s)", which have descendants in a tree-structured hierarchy. Unfortunately, the regular set operators as described above ignored this structural feature of XML data. On the other hand, some XML applications can be benefited from set operators with different semantics considering ancestor-descendant relationships between nodes. In this extended semantics, the comparison during query processing are conducted not only on nodes of both operands, but also on their descendants, in a "deep" manner. In this paper, we identify the needs of such "deep" set operators and propose the `deep-union`, `deep-intersect` and `deep-except` operators. We further explore their properties as well as relationships to regular set operators, and present a preliminary experience on implementing them as user-defined functions of XQuery.

1. INTRODUCTION

In recent years, the eXtensible Markup Language (XML) [3] has emerged as the *de facto* standard for storing and exchanging information. As such, the needs arise to query and tailor information in XML documents for various requirements in a more flexible and powerful manner. Towards this goal, XQuery [2] was developed by two W3C working groups to serve as the standard XML query language. In [2] and [5], three *set operators* are defined, namely `union`, `intersect` and `except`. First, in [2], they are defined as:

- *"The union and | operators are equivalent. They take two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands."*
- *"The intersect operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands."*
- *"The except operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand."*

XIME-P '05 Baltimore Maryland, USA

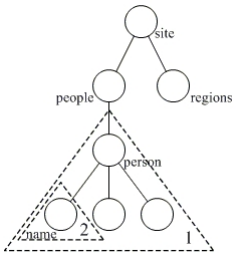

Figure 1: Tree structure of an XML document.

Second, according to [5], the set operators of XQuery are defined using the notion of node-IDs – unique (conceptual) ID per XML node. Therefore, for instance, the "union" of two node sequences are the union of node-IDs from both sequences. On the other hand, XQuery uses XPath [1] to locate nodes. As defined in XPath, once the query is processed and the final node-IDs, say {5, 7}, are found, the answer to be returned to the user is the entire subtree rooted at the node with IDs 5 and 7. For instance, Figure 1, an XPath expression "//person" represents all the subtree rooted at <person> node (subtree 1), and "//person/name" represents the whole subtree rooted at "<name>" node (subtree 2), which is hierarchically nested as a subtree in subtree 1.

In addition, the set operators defined in XQuery only require two operands to be "node sequences" without any further requirements on their comparability. Therefore, two operands may be sequences of nodes at different level, or even nodes from mixed levels. For instance, in the expression "//person `union` //name", two operands contain different nodes, <person> and <name>, and are thus incomparable regarding their semantics. In the relational model, this kind of union is not valid because of incompatible domains. However, XQuery accepts this query and would return a sequence of mixed nodes, <person> and <name>. Consequently, the "regular" set operators defined in XQuery are more flexible than their counterparts in the relational model, but they sometimes generate confusing semantics. In particular, we have observed that some XML applications would have been benefited greatly if there are "novel" set operators with different semantics in XQuery. Consider the following two motivating examples.

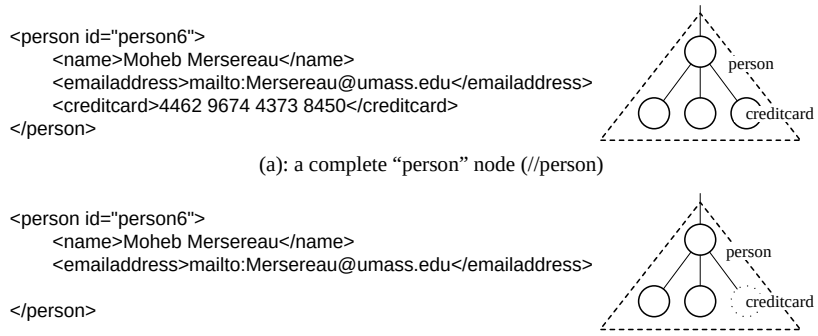

Figure 2: An example of `deep-except` semantics

Example 1 (XML Access Controls). In [8], we proposed an XML access control enforcement method that re-writes users' incoming query Q to a safe query Q' such that all fragments in Q that are asking for illegal data access are pruned out. In this context, conceptually, the safe query is either (1) "intersect" of Q and what users are granted to access (i.e., positive access control rules), or (2) Q "except" what users are prohibited to access (i.e., negative access control rules). That is, if Q is //person, but a positive access control rule grants only the data matching //person/name[age>18], then users must have an access to the data that are the intersect of //person and //person/name[age>18]. However, the expression using the regular `intersect` operator, "//person `intersect` //person/name[age>18]" would return NULL since no node-IDs from //person and //person/name[age>18] match. What is desirable here is, thus, a novel "intersect" operator that compares operands to check their structural overlap in a deep manner, and returns the overlapped region. That is, "//person `deep-intersect` //person/name[age>18]" should return the nodes matching //person/name[age>18] since it is completely nested in //person.

Symmetrically, for negative access control rules, we need novel `deep-except` operator. For instance, if a user issues a query //person but a negative access control rule prevents her from accessing the data matching //person/creditcard, then what she can really access is the data matching the expression "//person `deep-except` //person/creditcard". Again, the regular `except` operator, if used, would have resulted wrong semantics. Figure 2 illustrates the above semantics of `deep-except`: (a) shows two <person> nodes in their original form, while (b) shows the expected output of the "deep" query "//person `deep-except` //person/creditcard".

Example 2 (Database as a Service). In recent proposal to use database as a service model [6], query and data are delivered over to the database site which processes the query and returns answers back to users. Furthermore, XML data may be gathered and stored in a non-replicating fashion. A small company A that has branches in LA and NY may store different but partly overlapping XML data in both branches. When analysis needs to be done, the company ships query and two snapshots of XML data to 3rd party company B that provides database-as-a-service. For instance, the company A wants to gather aggregated statistics over items that were sold in 2004, and may request names of all items in the LA branch to be sent to B, while requesting complete item information of Northern America to be sent to B. That is, what B will receive is the "merged snapshot" of //item/name and //namerica/item. Like Example 1, this cannot be handled by the regular `union` operator, and can only be coped by introducing the novel `deep-union` operator as follows: "//item/name `deep-union` //namerica/item". □

2. RELATED WORK

In [4], "deep union" and "deep update" operators are proposed to process semi-structured data. This operator takes two edge-labeled tree-structural documents as input and merges/updates them based on their structural similarities.

In [9], a deep-equal function is introduced to check the equality of two sequences. It checks if the arguments contain items that are equal in values and positions. Despite the same name, their deep operators (functions) are different from ours in semantics or underlying operation objects.

In [7], TAX algebra is proposed for tree-structured data, in accordance with the relational algebra for relational data. TIMBER [10] is developed based on this algebra.

As we illustrate above, there are needs for novel set operators with enhanced semantics than regular set operators. In this paper, we explore this issue of deep set operators. The remainder of the paper is organized as follows. In section 3, we define deep set operators and illustrate them with examples. Section 4 proposes their properties and comparison with regular set operators. Section 5 describes the algorithm of our implementation of three deep set operators, and in Section 6 we provide our experiment results.

3. FORMAL DEFINITIONS

We first give precise formal definitions of three novel deep set operators, followed by illustrative examples.

3.1 Definitions

First, we denote node sequences as $P = \{p_1, ...p_n\}$ and $Q = \{q_1, ...q_n\}$, where p_i and q_i are XML nodes, identified by node-IDs according to XQuery semantics [5]. And the enumeration of the nodes and all their descendant nodes as:

$$P_d = P/\texttt{descendant-or-self}::*$$
$$Q_d = Q/\texttt{descendant-or-self}::*$$

Definition 1 (deep-union) *`deep-union` operator ($\overset{D}{\cup}$) takes two node sequences P and Q as operands, and returns a se-*

quence of nodes (1) who exist as a node or as a descendant of the nodes in "either" operand sequences, and (2) whose parent does not satisfy (1). Formally,

$$P \overset{D}{\cup} Q = \{r|(r \in P_d \vee r \in Q_d) \wedge (r :: parent() \notin P_d \wedge r :: parent() \notin Q_d)\} \square$$

In the above definition, condition (1) represents the foundamental semantics of deep union operator: compare not only nodes in operand node sequences but also their descendants; (2) serves as a supplement: when a node satisfies condition (1), all its descendants also satisfy condition (1), thus we wanted to eliminate the descendants and keep the "greatest common node" only [1]. Condition (2) is directly expressed as:

$$! (r :: parent() \in P_d \vee r :: parent() \in Q_d)$$

According to De Morgan's Law, it is equal to:

$$(r :: parent() \notin P_d \wedge r :: parent() \notin Q_d)$$

Similarly, we have

Definition 2 (deep-intersect) *`deep-intersect` operator ($\overset{D}{\cap}$) takes two node sequences P and Q as operands, returns a sequence of nodes (1) who exist as a node or as a descendant of the nodes in "both" operand sequences, and (2) whose parent does not satisfy (1). Formally,*

$$P \overset{D}{\cap} Q = \{r|(r \in P_d \wedge r \in Q_d) \wedge (r :: parent() \notin P_d \vee r :: parent() \notin Q_d)\} \square$$

To formally define `deep-except`, we first need to define the `deep-except-node` operator. W3C XPath [1] standard limits the connection of any two nodes be in one (or more) of the twelve axis. For any two given nodes, we can further categorize their relationship into three classes: (1) they are identical, (2) they are ancestor-descendant, or (3) there is no overlap between them (including sibling, etc.). In other words, two nodes cannot be "partly overlapped". Then we define the `deep-except-node` operator as:

Remark 1 *`deep-except-node` operator takes two nodes as operands, processes them according to the following conditions: (1)when the first node is equal to the second node, or is a descendant of the second node, return* ***null****; (2) when the second node is a descendant of the first node, remove it from the subtree of the first node and return the remaining; (3) otherwise, when there is no overlap between the first and second nodes, return the first node.*

In addition to Remark 1, we extend the second operand to a "node sequence" to define `deep-except-nodeseq`:

Remark 2 *`deep-except-nodeseq` operator takes one node as the first operand and one node sequence as the second operand, process them according to the following conditions: (1)when the first operand is equal to any node in the second operand, or is a descendant of any node in the second operand, return* ***null****; (2) when any node(s) of the second operand is descendant(s) of the first operand, remove it(them) from the first operand and return the remaining; (3) otherwise, when there is no overlap between the first and second operands, return the first operand.* $\square$

[1]According to XML standards, when this node is projected to the document, the whole subtree rooted at this node is returned.

	$\cup$	$\overset{D}{\cup}$	$\cap$	$\overset{D}{\cap}$	-	$\overset{D}{-}$
A=B	A	A	A	A	$\emptyset$	$\emptyset$
//A//B	{A, B}	A	$\emptyset$	B	A	partial content
no overlap	{A, B}	{A, B}	$\emptyset$	$\emptyset$	A	A

Table 1: Comparison between set operators and deep set operators

Finally, `deep-except` operator is defined as follows.

Definition 3 (deep-except) *`deep-except` operator ($\overset{D}{-}$) takes two node sequences as inputs, and conducts `deep-except-nodeseq` operation between each node in the first operands vs. the second operand, and combine the outputs. Formally,*

$$P \overset{D}{-} Q = \{r|r \in (p_i \texttt{ deep-except-nodeseq } Q)\} \square$$

Here we can see that the definition of `deep-except` operator appears to be different from the other two deep set operators. The differences are further discussed and explained in Section 4.

3.2 Examples

Consider the following XML fragment:

```
<a> <b> <c/> </b> <d/> </a>
```

Query "/a `union` //b" yields both <a> and <b> nodes. When projected on the document, the answer would be:

```
<a> <b> <c/> </b> <d/> </a>, <b> <c/> </b>
```

On the other hand, query "/a `deep-union` //b" yields <a> nodes only. When projected on the document, the answer would be:

```
<a> <b> <c/> </b> <d/> </a>
```

Query "/a `deep-intersect` //b" yields <b> nodes:

```
<b> <c/> </b>
```

Finally, query "/a `deep-except` //b" yields newly constructed <a> nodes, which is different from original <a> nodes:

```
<a> <d/> </a>
```

As another example of deep set operators, we compare deep set operators and regular set operators at micro level: given two nodes (i.e. only one item in each node sequence as operand), what could be the productions of deep set operations, as well as regular set operations?

As we pointed out, the relationship between two nodes A and B can only be one of the following: (1)they are the same (A=B); (2) A is an ancestor of B (//A//B)[2]; or (3) they are not related (no overlap between them). Table 1 summarizes the results of conducting regular set and deep set operators on two nodes of each category.

As an example, if we take a node (e.g. //person[@id='1']) and one of its grandchild (e.g //person[@id='1']/address/zip) as operands to conduct three set operations defined in X-Query, they will generate the results as shown in Figure 3. As we can see, regular set operators compares the IDs of two nodes and found them different. Thus union operation returns both nodes, intersect operation returns NULL,

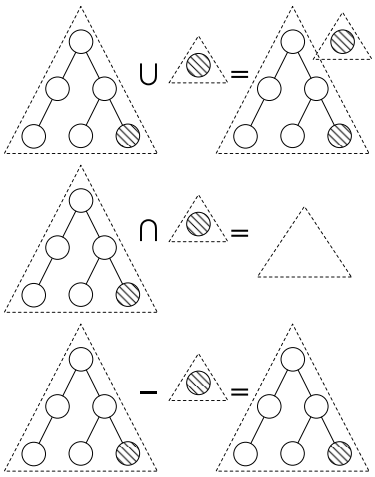

Figure 3: set operators defined in XQuery

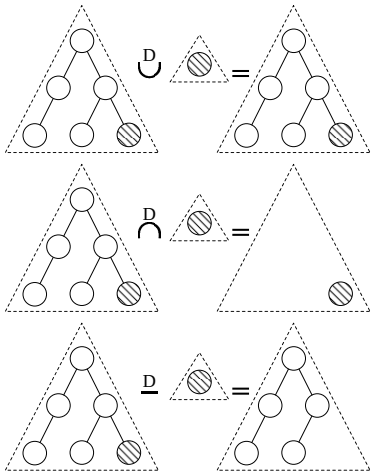

Figure 4: Deep set operators

and except operator returns the first operand (grandparent node).

On the other hand, if we take the above nodes and conduct deep-set operations, different results are generated, as shown in Figure 4. Deep set operators detects that the node of second operand is a descendant of the first operand, thus: (1) deep-union operation returns the ancestor (e.g //person[@id='1']) only; (2) deep-intersect operation returns the descendant only (e.g //person[@id='1']/address/zip); and (3) deep-intersect operation returns a newly constructed node, which contains partial content of the ancestor node, with one of the descendant node (the second operand) been removed.

The above example is case 2 in Table 1. Comparing Figure 3 with Figure 4, we can observe the essential difference between regular set operators and deep set operators: the regular set operators only compare and process node(s) in operands, while deep set operators compare and process node(s) as well as their descendants.

4. PROPERTIES

4.1 Arithmetic Properties

[2]For case "A is a descendant of B" (//B//A), we can simply swap token A and B, thus it is still categorized as case 2

The following properties of deep set operators are most fundamental and can be easily proved by their definitions (here we omit the proof due to space limit). They are similar to the properties of regular set operators.
Commutativity

$$P \overset{D}{\cup} Q = Q \overset{D}{\cup} P$$
$$P \overset{D}{\cap} Q = Q \overset{D}{\cap} P$$
$$P \overset{D}{-} Q \neq Q \overset{D}{-} P, \quad \text{unless} \quad P = Q$$

Associativity

$$(P \overset{D}{\cup} Q) \overset{D}{\cup} R \;=\; P \overset{D}{\cup} (Q \overset{D}{\cup} R)$$
$$(P \overset{D}{\cap} Q) \overset{D}{\cap} R \;=\; P \overset{D}{\cap} (Q \overset{D}{\cap} R)$$

Distributivity:

$$(P \overset{D}{\cup} Q) \overset{D}{\cap} R \;=\; (P \overset{D}{\cap} R) \overset{D}{\cup} (Q \overset{D}{\cap} R)$$

In addition, we would like to point out one essential difference between `deep-except` operator and the other two. As we can see, both `deep-union` and `deep-intersect` operators return a sequence of nodes that are originated from the given XML tree, i.e. they do not create any new nodes. In this way, the production of these two operators are available for any other operations defined in XQuery that accepts nodes as operands, e.g. union, intersect, etc. On the other hand, whenever `deep-except-nodeseq` (case 2 of Remark 2) operation is conducted for `deep-except` operator, new nodes are constructed. Therefore, the production of `deep-except` operator may not be existing node in the XML tree. In this way, we should identify that although three operators are named "deep set operators" together, they are actually operators of different properties. `deep-except` operator is constructing new nodes while the other two operators return nodes of the original XML tree, which is similar to the regular union and intersect operators.

As an example, "//person `deep-except` //person/name" returns "person" nodes. However, the returned "person" nodes are not the same as "person" nodes that reside in original XML document: the "person" nodes produced by `deep-except` operation do not have "name" child. As a result, the newly constructed "person" nodes have new nodeIDs, which are different from original "person" nodes. Therefore, with the production of `deep-except` operator, we have to be careful when conducting further operations with existing "person" nodes in the XML document, such as union, intersect etc. Moreover, we cannot use

$$//person \overset{D}{-} //person/name \overset{D}{-} //person/age$$

although it looks fine in semantics. Instead, we have to use:

$$//person \overset{D}{-} (//person/name \cup //person/age)$$

4.2 Comparison with Set Operators

As we described above, `deep-union` and `deep-intersect` operators return nodes of the original document (probably node-IDs in actual applications). Here we provide two theorems to further describe the output of these two operators, especially their relationships with regular set operators.

Lemma 1. *`deep-union` of two node sequences is subset of their union production:*

$$(P \overset{D}{\cup} Q) \subseteq (P \cup Q) \qquad \blacksquare$$

PROOF. Lemma 1 is equivalent to:

$$\text{if} \quad r \in P \overset{D}{\cup} Q, \quad \text{then} \quad r \in P \cup Q. \tag{1}$$

Suppose we have an r that $r \in P \overset{D}{\cup} Q$, according to Definition 1:

$$r \in P_d \quad \text{or} \quad r \in Q_d.$$

Consider the equivalency of P and Q, we can assume

$$r \in P/descendant - or - self :: * \tag{2}$$

According to Definition 1, we also have

$$r :: parent() \notin P/descendant - or - self :: *$$

which means

$$r \notin P/descendant :: * \tag{3}$$

Comparing (2) and (3), we have

$$r \in P, \quad \text{thus} \quad r \in P \cup Q,$$

which proves Equation 1. (q.e.d)

Lemma 2. *`deep-intersect` of two node sequences is subset of their union production:*

$$(P \overset{D}{\cap} Q) \subseteq (P \cup Q),$$

On the other hand, it is not possibly subset of their intersect production:

$$\exists P, Q \quad that \quad (P \overset{D}{\cap} Q) \not\subseteq (P \cap Q) \qquad \blacksquare$$

This theorem is also described as : if $r \in P \overset{D}{\cap} Q$, then $r \in P \cup Q$, but not always $r \in P \cap Q$.

Lemma 3. *Unless both operands contain ancestor-descendant nodes, `deep-intersect` of two node sequences is superset of their intersect production:*

$$(P \cap Q) \subseteq (P \overset{D}{\cap} Q) \qquad \blacksquare$$

The proofs of the above two theorems are similar to that of Lemma 1, and thus omitted.

5. PRELIMINARY IMPLEMENTATIONS

We have implemented the deep set operators through user-defined functions of XQuery. With this implementation, these operators are executable in any XML engine that supports XQuery. On the other hand, as a drawback, this engine-independent implementation may not be as efficient as implementations at lower level (say, engine level).

As XQuery's user-defined functions, our implementations take node sequences as inputs, including XPath expressions and other forms of node sequences (e.g. products of set operators). In addition, as described above, the results of both `deep-union` and `deep-intersect` operations are XML node sequences and are available to further XQuery operations. Our implementation also supports this property.

5.1 `deep-union` operator

According to theorem 1, product of `deep-union` operator is a subset of regular union operation, i.e. each output node must originally resides in at least one operand. Following this theorem, `deep-union` operator, as shown in Algorithm 1, enumerates the nodes in each operands, referring to requirements of `deep-union` and return the satisfied ones.

Algorithm 1: `deep-union`

Input: input node sequences P and Q
foreach *node* P_i *of* P **do**
  if *empty*$(P_i \cap Q//*)$ *&* *empty*$(P_i \cap P//*)$ **then**
    P_i
foreach *node* Q_i *of* Q **do**
  if *empty*$(Q_i \cap (P \cup P//*))$ *&* *empty*$(Q_i \cap Q//*)$ **then**
    Q_i

5.2 `deep-intersect` operator

`deep-intersect` is implemented in a similar way as `deep-union`. According to theorem 2, each output node of `deep-intersect` operator must originally resides in at least one operand. To enhance the readability of the algorithm, we divide it into three steps: first extract the regular "intersect" of two operands, remove the possible ancestor-descendant relationship that may exists, and output the remaining. Then, for each of the operand, enumerate the node items, output it if it is a descendant of the other operand (some exceptions are eliminated). Algorithm 2 shows how `deep-intersect` works.

Algorithm 2: `deep-intersect`

Input: input node sequences P and Q
foreach *node* r *in* $(P \cap Q)$ **do**
  if *empty*$(r \cap (P \cap Q)//*)$ **then**
    r
foreach *node* P_i *of* P **do**
  if *empty*$(P_i \cap P//*)$ *&* *!empty*$(P_i \cap Q//*)$ *&* *empty*$(P_i \cap (P \cap Q))$ **then**
    P_i
foreach *node* Q_i *of* Q **do**
  if *empty*$(Q_i \cap Q//*)$ *&* *!empty*$(Q_i \cap P//*)$ *&* *empty*$(Q_i \cap (P \cap Q))$ **then**
    Q_i

5.3 `deep-except` operator

For `deep-except` operator, as we have described in Section 3.1, it is different from the other two deep set operators. We implement it in a recursive manner: for each node in the first operand, (1) if it has no overlap with any node in the second operand, output it; (2) if it is ancestor of any node(s) in the second operand, construct a new node with the same name, attributes and text, and enumerate all the children to conduct `deep-except` with the second operand; (3) otherwise, the node is "covered" by nodes in the second operand, eliminate it. The general algorithm is shown in Algorithm 3.

Algorithm 3: `deep-except`

Input: input node sequences P and Q
foreach *node* P_i *in* P **do**
  if *empty*(P_i//* $\cap$ Q) *and empty*($P_i \cap (Q \cup Q$//*)) **then**
    P_i
  if *!empty*(P_i//* $\cap Q$) **then**
    construct element{
    element_name=name(P_i);
    element_attributes=P_i/@*;
    element_text()=P_i/text();
    deep-except(P_i/*,Q);
    }
  else

5.4 Complexity

Although the above preliminary implementations may not be fully optimized, we can still estimate the computation of deep set operators. The computation of `deep-union` and `deep-intersect` operators are both $O(n_c * n_s)$, where n_c denotes the total number of nodes in the sequences of operands, and n_s denotes the total size of the subtrees rooted at these nodes. On the other hand, the computation of `deep-except` operator (P `deep-except` Q) is denoted as $O(n_{cq} * n_{sp} * i)$, where n_{cq} denotes number of nodes in P, n_{sp} denotes the size of subtrees rooted at nodes in P, i denotes the maximum depth of these subtrees. This appears to be more expensive than the other two, since recursive function call is employed in the implementation. It could be greatly optimized if implemented at XML engine level.

6. PRELIMINARY EXPERIMENTS

6.1 Experimental Results

In the experiments, we use the well-known XMark schema and its XML document generator [11] to generate the test document. We generate a 570KB XML document and test the performance of deep set operators vs. regular set operators on it. For the underlying XML engine, as we have described, any engine that implements XQuery is usable since our implementation relies on XQuery only; here we pick Galax 0.5.0 (and its Java API) [12].

For each operator, we generate pairs of XPath expressions (P and Q) as operands, and test all three deep set operators. For comparison, we also test regular set operators with the same operands. However, please note that the straight comparison of query evaluation time is unfair since regular set operators are supported within the core of XML engine, but our deep set operators are only implemented through user defined functions (UDF) of XQuery. We present the comparison only for reference, not for competition.

Experiment results are shown in figure 5, note we only measure the pairs where P and Q overlap (i.e. $P \overset{D}{\cap} Q \neq \varphi$), since only in these cases the products of regular set operators and deep set operators differ.

6.2 Discussions

From the figure, we can see that deep set operators are slower than regular set operators, but query processing time is still within 0.5 ms on this document. Deep set operators

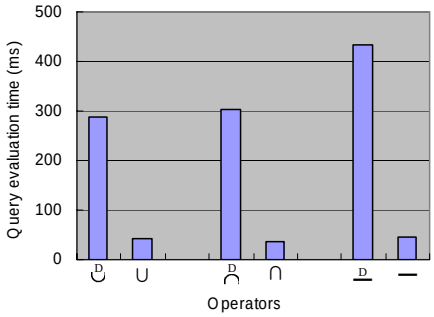

Figure 5: Experiment results of `deep-except`

are slower because of two main reasons:

1. Deep set operators need extra overhead to support more intensive semantics. It is not fair to directly compare these two groups of set operators, since they are in fact different things. In this way, the preliminary experiments are just used as a reference of the performance of deep set operators.

2. In our experiments, deep set operators are implemented through user-defined functions (UDF) of XQuery. Thus they could not compete with regular set operators, which are well optimized in the engine (core). For instance, the main part of deep intersect operator is comparing two node sequences and picking the common elements, for which algorithms of $O(\log n)$ exists. However, we cannot employ these optimized algorithms in XQuery UDF.

In addition, `deep-except` is slower than other two because of the recursive implementation of deep except semantics. However, this could not be avoided as long as we are using UDF of XQuery.

In the future, we plan to improve the performance of deep set operators by implementing them in the XML database engine. In this way, we can design and implement more efficient (and more complicated) algorithms. Together with the advantage of engine level implementation, we are sure to gain performance improvement.

7. CONCLUSION

In this paper, we propose *deep set operators* for XQuery. Regular set operators defined in [2, 5] only rely on node-IDs to identify and compare nodes. Therefore they ignore the comparability issue of operands and neglect the tree-structured hierarchy of XML data. The newly defined deep set operators take the ancestor-descendant relationships within input XML nodes into consideration. Items in input node sequences are not regarded as atomic entities, instead, they are treated as hierarchically nested elements, which accords with the designed semantics of XML. Deep set operators traverse into descendants of nodes to conduct comparison and processing, in a "deep" manner. We propose the formal definition of deep set operators, then explore their computational properties as well as relationships with regular set operators. Finally we provide an implementation of deep set operators, which purely rely on XQuery and thus independent from XML engine. Experiment show that some reasonable overhead is required to support deep set operations.

8. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J. Simeon. "XML Path Language (XPath) 2.0". W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xpath20.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. "XQuery 1.0: An XML Query Language". W3C Working Draft, Feb. 2005.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). "Extensible Markup Language (XML) 1.0 (2nd Ed.)". W3C Recommendation, Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[4] P. Buneman, A. Deutsch, and W.-C. Tan. "A Deterministic Model for Semistructured Data". In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1998.

[5] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Ross, M. rys, J. Siméon, and P. Wadler (Eds). "XQuery 1.0 and XPath 2.0 Formal Semantics". W3C Working Draft, Apr. 2004. http://www.w3.org/TR/xquery-semantics/.

[6] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. "Executing SQL over encrypted data in the database-service-provider model". In *ACM SIGMOD*, 2002.

[7] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. "TAX: A Tree Algebra for XML". In *Int'l Workshop on Data Bases and Programming Languages (DBPL)*, Frascati, Rome, Sep. 2001.

[8] B. Luo, D. Lee, W.-C. Lee, and P. Liu. "QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting". In *ACM CIKM' 2004*, Washington D.C., USA, Nov. 2004.

[9] Ashok Malhotra, Jim Melton, and Norman Walsh. "XQuery 1.0 and XPath 2.0 Functions and Operators". W3C Working Draft, Feb. 2005. http://www.w3.org/TR/2005/WD-xpath-functions-20050211/.

[10] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: a native system for querying xml. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 672–672, New York, NY, USA, 2003. ACM Press.

[11] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. "The XML Benchmark Project". Technical Report INS-R0103, CWI, April 2001.

[12] J. Simeon and M. Fernandez. "Galax V 0.3.5", Jan. 2004. http://db.bell-labs.com/galax/.

APPENDIX

A. SOURCE CODE

A.1 Deep Union

```
declare  function local:d_u($p,$q){
  let $pqr:=$q//* union $p//*
  for $pn in $p
  return
    if (empty($pn intersect $pqr)) then
      $pn
    else(),
  let $pqr:=$p//* union $q//*
  for $qn in $q
  return
    if (empty($qn intersect $p) and
        empty($qn intersect $pqr)) then
          $qn
    else()
};
```

A.2 Deep Intersect

```
declare  function local:d_i($p,$q){
  let $pr:=$p//*
  let $qr:=$q//*
  for $pn in $p
  return
    if (empty($pn intersect $pr) and
        (not(empty($pn intersect $qr or
         not(empty($pn intersect $q)))))) then
          $pn
    else(),
  let $pr:=$p//*
  let $qr:=$q//*
  for $qn in $q
    return
      if (empty($qn intersect $qr) and
          not(empty($qn intersect $pr))) then
            $qn
  else()
};
```

A.3 Deep Except

```
declare  function local:d_e($p,$q){
  let $qr:=$q//*
  for $pn in $p
  return
    if (empty($pn//* intersect $q) and
        empty($pn intersect $q) and
        empty($pn intersect $qr)) then
            $pn
    else if (not(empty($pn//* intersect $q))) then
      element{name($pn)}{
        $pn/@*,
        $pn/text(),
        local:d_e($pn/*,$q)
      }
    else()
};
```