Generic and Updatable XML Value Indices Covering Equality and Range Lookups

Lefteris Sidirourgos
CWI
Amsterdam, The Netherlands
E.Sidirourgos@cwi.nl

Peter Boncz
CWI
Amsterdam, The Netherlands
P.Boncz@cwi.nl

ABSTRACT

We describe a collection of indices for XML text, element, and attribute node values that *(i)* consume little storage, *(ii)* have low maintenance overhead, *(iii)* permit fast equi-lookup on string values, and *(iv)* support range-lookup on any XML typed value (e.g., double, dateTime). The equi-lookup string value index depends on an *elaborate hash function* and on an *associative combination function* to facilitate updates on both mixed-content and element nodes. We also present techniques for creating range-lookup indices supporting any *ordered* XML typed value. These indices rely on a *finite state machine* that accepts the type specific language, and on a *state combination table* for combining states to speed-up updates. We evaluate the stability of the hash function, the storage overhead, and the indices creation and maintenance time in the context of the open-source XML database system MonetDB/XQuery.

1. INTRODUCTION

The semantics of XQuery are designed to facilitate querying both typed and untyped XML data, whose contents in turn may vary from strongly structured data to very loosely structured mixed-content data. For instance,

```
doc("persons.xml")//person[.//age = 42]
```

will return `<person>` nodes, that have at least one `<age>` node with integer value 42. In absence of an XML Schema that would give `<age>` a specific type, the equality predicate will match all `<age>` nodes with a *string value* that is castable to an `xs:double` with value `42.0`, e.g.,

```
<age>42</age>
```

However, other matching instances exist, such as

`<age>42.0</age>` and `<age>     +4.2E1</age>`

That is, any text node containing a decimal number, or a double in various syntactical forms (including leading white space characters) all cast to `42.0`.

To further complicate things, the string value of an element node or a mixed-content node, is the concatenation of the string values of all descendant text nodes [10], such that the following node also matches the predicate.

```
<age><decades>4</decades>2<years/></age>
```

While this flexibility is one of the crucial advantages that XML offers over relational data management technology, it complicates the generic use of value indices for general comparisons. As a result, XQuery systems typically require a system administrator to specify the document sub-paths and the value casts to be indexed. In case of DB2 PureXML [7], for example, one could do

```
create index myindex on items(person) generate key
  using xmlpattern "//person//age" as sql double
```

However, this approach has the following disadvantages: *(i)* only queries that use the specific listed paths can be accelerated, *(ii)* the index is type specific; i.e., an index on double values cannot be used in case of string lookups, and *(iii)* in contrast to current trends in data management systems, it requires explicit (DBA-induced) index configuration.

In our view, the above mixed XML content example of `age` being decomposed in `decades` and `years`, yet accidentally mapping onto 42, is a rather unintended consequence of the XPath and XQuery standards, that may even be called undesired. However, it *is* part of these well-entrenched standards and should thus be supported by XML database systems. This work addresses the above by investigating indexing techniques that enable *self-tuning* index management, by creating generic XML value indices that cover an entire document, not just a single path with one particular value type, and allow equi-comparisons on string values, as well range-lookup on any XML typed value. The index structures presented in this paper are optimized for the cases where values represented by a single node are looked up, but also are able to correctly deal with mixed XML content.

String Equi-Index. The string value index depends on a specialized hash function $H(\texttt{str}) : int$, carefully designed to map arbitrary length string values into integer hash values in such a way that hash collisions are kept low. The hash function can be used to index XML text, element, and attribute node values. A (B-tree) index, constructed on the hash values, can be used for accelerating string value lookups, e.g., for evaluating the XPath expression

```
doc("persons.xml")//person[.//first="Arthur"]
```

that returns all `<person>` nodes with first name `"Arthur"`.

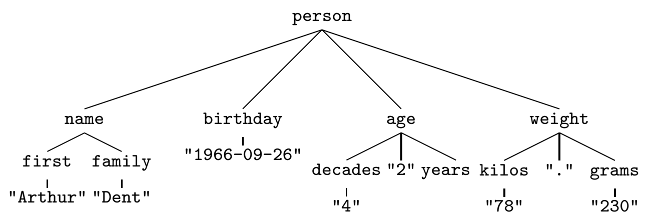

Figure 1: XML Document about persons

In addition, we define an associative combination function $C(int, int) : int$ that can be used to derive the hash value of the concatenation of a list of string values. This is achieved by using the *hash values* computed over the individual string values of the list, thus avoiding accessing the actual string data which can be costly. Figure 1 depicts the tree representation of an XML document about a person. In this example, the string value of node `<name>` is `"ArthurDent"`. Thus, the XQuery query

```
doc("persons.xml")//*[fn:data(name)="ArthurDent"]
```

returns node `<person>`. In effect, the hash value of an element can be computed by combining, through function C, all hash values of its direct children. When updates are also considered, reconstructing the hash value index requires visiting all ancestors of the updated nodes and their immediate children, in order to recompute the hash values with function C. This means that on well-shaped trees, with $\log n$ depth and a low fanout, index maintenance costs remain limited.

Typed Range-Index. Moreover, we construct indices for range-lookup on other XML typed values, and still respect the mixed-content and element node semantics. Of particular interest are the types `xs:dateTime` and `xs:double` [11]. Note that an index on `xs:double` can be used to accelerate predicates on all numerical XQuery types.

This group of range-lookup indices relies on *finite state machines (FSM)* that recognize the language that accepts the specific type. For example, if double values are considered, an *FSM* can be defined such that it recognizes (*potential*) valid lexical representations. This *FSM* will return a *reject* state if an illegal sequence of characters is encountered, or the *state* in which the *FSM* terminated. In case of doubles and for typical XML data, the majority of all text nodes that do not represent a numeric value will be rejected immediately, avoiding waste of indexing resources. A minority of nodes will contain *potential* valid lexical representations, i.e., those that although they cannot be casted to a double value, do not contain an illegal sequence of characters (they are just incomplete). For example, the text node `"78"` rooted at node `<kilos>` in Figure 1 is a valid lexical representation that can be casted to a double. On the other hand, the text node `"."` rooted at node `<weight>`, although not castable to a double value, is a potential valid representation – it misses one or more leading and trailing digits.

In addition to an *FSM*, a *state combination table (SCT)* is defined for each supported XML typed value. The role of the *SCT* is similar to that of function C for the string value index: to efficiently decide if the combination of two nodes is a (potential) valid lexical representation of the XML typed value and to return their combined state. For example, concatenating all descendant text nodes of node `<weight>`, i.e.,

```
<kilos>78</kilos>.<grams>230</grams>
```

produces a valid lexical representation of a double value, namely `78.230`. As in the case of function C, SCT is used during the creation and updates of the range-lookup index. Finally, based on the selection of only those nodes that contain a valid lexical representation of the indexed type provided by the *FSM*, a (B-tree) index is built on the values of those nodes to facilitate fast range lookups.

The main characteristics and novelty of the indices presented in this work can be summarized in the following:

cover the entire document: all element, attribute, and text nodes are indexed, independently from their path

respect the XQuery semantics: the indices works correctly even in the presence of intermediate and mixed-content nodes

self-tuned: in the sense that no explicit index configuration, which defines a type and/or a path, is required

consume little storage: no data replication is needed

updatable and low maintenance overhead: the design of the indices are driven by the need of efficient updatable generic indices.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 and 4 details the indexing techniques. Section 5 describes the implementation details for creating and updating the indices. The experimental justifications of our claims are presented in Section 6. Section 7 concludes our work.

2. RELATED WORK

We focus our study of related work to fully functional XML engines, and their implementation of value indices. To the best of our knowledge there is no other work that address the problem of generic XML value indices that work well, are updatable and are able to deal with the abnormality of the mixed-content node data values. All systems we examined require a user-originated definition of specific paths to be indexed, in contrary to our solution which covers the entire document.

In the context of DB2 native XML engine [4, 7] value indices are supported only for path-specific values and for predefined types. A query is been accelerated by those indices only if the path and the type match. The key of such XML indices is `[pathid, value, nodeid, rid]`. The order in which these tuples are indexed (e.g., on pathid or value) facilitate different kind of query acceleration.

In [8] two different types of XML indices are constructed, namely *primary* and *secondary*. The primary index is a B^+-tree on tuples of the form `[ordpath, tag, nodetype, value, pathid]`, ordered according to the document order. The secondary indices are defined on top of the B^+-tree keys, and on the columns of interest. For example, the XML value index is defined on `[value, pathid, ordpath]`. Such indices allow to speed up evaluation for specific queries, however, as also pointed out in [9], they do not respect the XQuery Data Model.

3. STRING VALUE INDEX

We present a fully updatable index covering equality lookups on XML string values. The index is based on a specialized

```
Hash function H
input: XML string value str as sequence of chars
output: 32-bit hash value h_val

1: h_val = 0; /* initialize h_val */
2: /* populate the c-array */
   for (offset = 0; *str!='\0'; str++)
       c = *(str) & 127;
3:     /* circular_XOR operation*/
       h_val ^= c << offset;
       if (offset > 20)
          h_val ^= c >> (27 - offset);
4:     offset += 5; /* update offset */
       if (offset > 26) offset -= 27;
5: /* set the OFFC bits */
   h_val <<= 5; h_val |= offset;
6: return h_val;
```

Figure 2: Algorithm for Hash Function H

hash function H that maps string values to hash values. Hash values are then indexed for fast lookup during query time. Each hash value is associated with a set of *candidate* XML node ids, which can then be further processed to select those nodes whose value and path structure are relevant to the query.

The hash function H accepts a *sequence of characters* (i.e., an XML string value) and outputs a *32-bit hash value*. The 27 most significant bits of the hash value, called in the sequel *c-array*, are used for hashing characters. Since an XML string value may be of arbitrary length we base the hash function on a *circular XOR operation*. The *circular XOR operator* works by applying the XOR operator between the 7 least significant bits of the value[1] of each character and the c-array, starting from the right most position (i.e., position 0), and gradually incrementing the offset by 5 bits to the left. When the end of the c-array is reached, that is when the offset is set to 25, 2 bits of the next character are XOR-ed to positions 25 and 26, while the remaining 5 bits are XOR-ed back to positions 0-4 of the c-array. The next character is then XOR-ed with the offset set to $(25 + 5) \mod 27 = 3$, thus circling around the c-array. The process is repeated until all characters of the sequence are processed. To produce hash values with such circular XOR operation, the 5 least significant bits of the 32-bit hash value are reserved for storing the offset information of the circle, i.e., the offset where the next character should be XOR-ed. More specific, the format of the hash value produced by the hash function H is

$$\underbrace{\text{C27}\cdots 1}_{\text{27-bits}} | \underbrace{\text{OFFC}}_{\text{5-bits}} \qquad \text{where}$$

C1. . . 27: The 27 most significant bits, forming the c-array, are reserved for hashing all characters of the XML string value. The c-array is built by applying the *circular XOR operator*.

OFFC: The least 5-bits form the OFFC field that encodes the 27 different offset positions of the c-array (i.e., the elements of set $\mathbb{Z}_{27}$).

Figure 2 details the algorithm for computing the hash value. The input is an XML string value `str`. The algorithm populates the c-array by iterating over all characters of `str` (line 2). The circular XOR operator is implemented by first shifting `offset` times to the left the 7 least bits

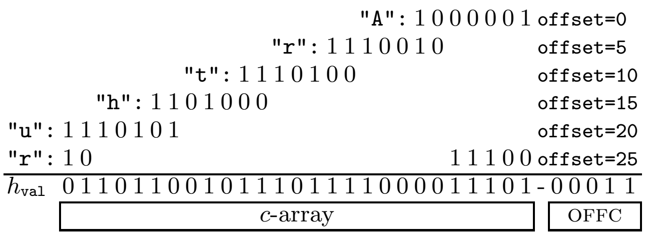
"A": 1 0 0 0 0 0 1 offset=0
"r": 1 1 1 0 0 1 0 offset=5
"t": 1 1 1 0 1 0 0 offset=10
"h": 1 1 0 1 0 0 0 offset=15
"u": 1 1 1 0 1 0 1 offset=20
"r": 1 0 1 1 1 0 0 offset=25
$h_{\texttt{val}}$ 0 1 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 1 1 0 1 - 0 0 0 1 1
c-array
OFFC

Figure 3: Example of computing H("Arthur")

of the current character, and then XOR-ing with $h_{\texttt{val}}$ (line 3). If the offset is larger than 20, then the remaining bits, computed as $(\texttt{offset} + 7) \mod 27$, are XOR-ed back at the beginning of the c-array. Next, the offset is incremented by 5 for the next iteration (line 4). When all characters of `str` are consumed, the OFFC bits are set (line 5) and the $h_{\texttt{val}}$ is returned (line 6).

Figure 3 depicts the iteration steps of function H for computing the hash value of `"Arthur"`, the text node of element `<first>` in the XML document of Figure 1. The procedure starts with `offset=0`. When the `offset` is 25, character `"r"` has to be processed. The 7 least significant bits of `"r"` are 1110010. From those, the two least significant bits (10) are XOR-ed with $h_{\texttt{val}}$ at positions 25 and 26, while the 5 remaining bits (11100) are XOR-ed to positions 0 to 4. The algorithm ends and the $h_{\texttt{val}}$ is returned. The OFFC bits are set to 3 (00011), indicating the next value of `offset`.

We define an *associative* function $C(int, int) : int$ to combine hash values during index creation and updates. Function C is designed such that given two string values $\texttt{str}_{\texttt{left}}$ and $\texttt{str}_{\texttt{right}}$ it holds that

$$H(concat(\texttt{str}_{\texttt{left}}, \texttt{str}_{\texttt{right}})) = C(H(\texttt{str}_{\texttt{left}}), H(\texttt{str}_{\texttt{right}}))$$

where *concat*() is the *string concatenation* function.

The combination function C is used during the creation of the index, as well in the event of updates. Suppose that the string value of node `<family>` in Figure 1 is updated from `"Dent"` to `"Prefect"`. After computing the new hash value of node `<family>` with H(`"Prefect"`), the hash value of node `<name>` must be updated too, and consequently node `<person>`. Without the combination function C that would call for evaluating once more

$$h_{\texttt{<name>}} = H(\texttt{"ArthurPrefect"}), \text{ and}$$
$$h_{\texttt{<person>}} = H(\texttt{"ArthurPrefect1966-09-264278.230"}).$$

Obviously, for large documents this is very inefficient since the string values of all nodes in the document have to be visited in order to reconstruct the hash values. However, with function C we only need to invoke function H once for the updated text node. The hash values of all ancestors of the updated node are reconstructed by visiting only the siblings of the ancestors, and reading their hash values, as opposed to reconstructing their string values. For example the new hash value of node `<name>` will be computed by

$$h_{\texttt{<name>}} = C(h_{\texttt{<first>}}, h_{\texttt{<family>}})$$

where $h_{\texttt{<first>}}$ and $h_{\texttt{<family>}}$ are the already computed hash values of nodes `<first>` and `<family>`. Similarly, the new hash value of node `<person>` will be

$$h_{\texttt{<person>}} = C(h_{\texttt{<name>}}, C(h_{\texttt{<birthday>}}, C(h_{\texttt{<age>}}, h_{\texttt{<weight>}}))).$$

[1]ASCII or UTF value depending on the implementation

Combination function C
input: hash values $\mathtt{h_{left}}$ and $\mathtt{h_{right}}$
output: combined hash value $\mathtt{h_{comb}}$

```
1: h_comb = 0; /* initialize $h_{comb}$ */
2: /* copy the $c$-array of the left operand to $h_{comb}$ */
   h_comb |= mask27(h_left)
3: /* circular left shift */
   h_comb ^= (mask27(h_right) << mask5(h_left)) |
        mask27(mask27(h_right) >> (27-mask5(h_left)));
4: /* add the OFFC bits of the left and right operands */
   h_comb |= (mask5(h_left) + mask5(h_right)) % 27;
5: return h_comb;
mask5(h) := h & 31
mask27(h) := h & (~31)
```

Figure 4: Algorithm for Combination Function C

Figure 4 details the algorithm to combine two hash values, namely $h_{\mathtt{left}}$ (the left operand) and $h_{\mathtt{right}}$ (the right operand). The algorithm outputs the combined hash value $h_{\mathtt{comb}}$ of the input hash values. First, the c-array of the left operand is copied to $h_{\mathtt{comb}}$ (line 2). In order to combine the c-array of the right operand with $h_{\mathtt{comb}}$, we apply the standard *circular left shift* operation to the c-array of $h_{\mathtt{right}}$. The c-array of $h_{\mathtt{right}}$ is shifted to the left by as many positions as indicated by the OFFC bits of $h_{\mathtt{left}}$. The result is then XOR-ed back to $h_{\mathtt{comb}}$ (line 3). Then, the OFFC bits of $h_{\mathtt{left}}$ and $h_{\mathtt{right}}$ are added, to update the offset information of the result hash value $h_{\mathtt{comb}}$ (line 4). Finally, the combined hash value $h_{\mathtt{comb}}$ is returned (line 5). The algorithm uses functions *mark5*() and *mark27*(), which apply bit operations to separate the c-array and the OFFC bits from the input 32-bit hash values.

The correctness of the output of function C is based on the observation that the XOR operator has the associative property, i.e., if i, j, k are integers, then $(i\hat{}j)\hat{}k = i\hat{}(j\hat{}k)$. By shifting the right operand to the left, we permute the position 0 of the c-array to the position indicated by the offset of the left operand. Recall that the OFFC bits of the hash value indicate the offset where the next character should be XOR-ed. Effectively, function C continues the circular XOR operation of function H but in a different order of applying the XOR operation. However, because the XOR operation has the associative property the result is guaranteed to be correct.

Next, we prove by induction the associative property of function C, that is:

$$\begin{aligned} &H(a_1 \cdots a_n) = && \text{(eq. 1)} \\ &\quad = C(C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1})), H(a_n)) \\ &\quad = C(H(a_1), C(H(a_2), \ldots C(H(a_{n-1}), H(a_n)) \ldots)) \end{aligned}$$

proof. The proof is by induction on n, the number of string values $a_1, \ldots, a_n$ that form the concatenated hash value $H(a_1 \cdots a_n)$. The base case is $n = 2$, that is $H(a_1 a_2) = C(H(a_1), H(a_2))$, which holds from the definition of function C. For n greater than 2, assume that (eq. 1) holds for all $k \geq 2$ such that $k < n$.

$$H(a_1 \cdots a_{n-1}) = C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1}))$$
(Ind. Hyp. with $k = n - 1$)

$$H(a_1 \cdots a_{n-1} a_n) = C(H(a_1 \cdots a_{n-1}), H(a_n))$$
(by base case)

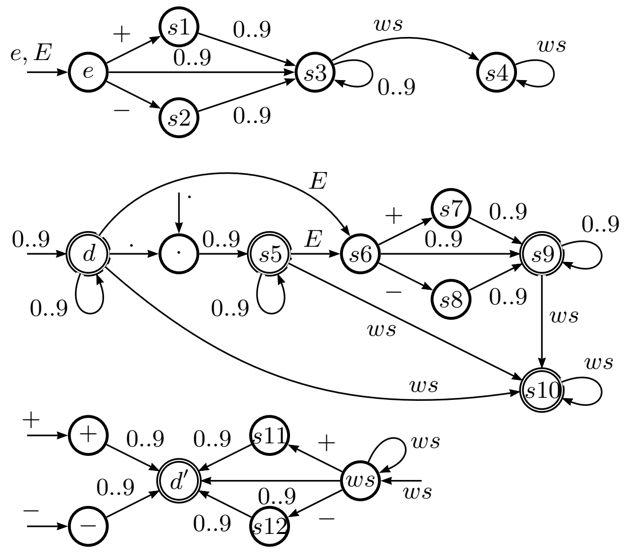

Figure 5: Finite State Machine for double values

$$\begin{aligned} &H(a_1 \cdots a_{n-1} a_n) = \\ &\quad = C(C(\ldots C(H(a_1), H(a_2)) \ldots, H(a_{n-1})), H(a_n)) \end{aligned}$$
(by base case and Ind. Hyp.)

Similarly, we can prove the second equality of equation (eq. 1). Thus, changing the order of *operations* (i.e., the order of applying function C) does not produce a different hash value H.□

In Section 5 we present an efficient and simple algorithm for visiting the relevant nodes and (re)compute the hash values, during index creation and updates.

4. TYPED RANGE-LOOKUP INDEX

We describe an updatable index covering range lookups which can be defined over any XML typed value. We detail the index over *double values*, however any other XML built-in type can be supported by applying the same ideas. The index is exact: it does not return false positives, neither it misses any nodes with value that matches the query.

This family of indices is based on *finite state machines* (FSM) that recognize the language that accepts the syntax (i.e., lexical representation) of the indexed XML type. Figure 5 illustrates the corresponding FSM for double values. Each distinct state of the FSM is depicted with a circle and each transition from one state to an other by an arrow. The arrows are labeled with the symbol of the language that permits the specific transition (E stands for the exponent character of doubles, and ws for whitespace characters). Double lined circles signify final states and states with incoming edges without a source signify initial states. Each state is marked with a unique label. In this content, all finite state machines that recognize an XML type are deterministic.

Notice that although the FSM of Figure 5 appears to have 5 initial states and 3 independent state transition graphs, each initial state is marked with a different symbol. Thus, depending the first character of the node string value, a different initial state is considered, implying the existence of a virtual empty initial state that redirects to the correct

	rej	**e**	**.**	**d**	**s1**	**s2**	...
rej	rej	rej	rej	rej	rej	rej	...
e	rej	rej	rej	s3	rej	rej	...
.	rej	rej	rej	s5	rej	rej	...
d	rej	s6	s5	d	s7	s8	...
s1	rej	rej	rej	s3	rej	rej	...
s2	rej	rej	rej	s3	rej	rej	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Figure 6: State Combination Table (SCT)

initial state.

Moreover, if there is more than one path leading to the same state, we expand the FSM in such a way that these paths lead to different copies of the same state, which in turn, the copied states are marked with a new unique label. This *normalization* of the FSM allows to uniquely identify the initial state and the path that leads to each state. For the proposed index, only through this normalization of the FSM, we can properly determine the consequences of concatenating arbitrary node string values by examining the states and not their string values. For presentation reasons, we have omitted the detailed expansion of the FSM in Figure 5: state d' redirects the FSM to state d, and for each distinct incoming edge of state d' (e.g., +, -, $s12$, etc.) different new labels are applied to all states following d. As a result, there are 60 different states including the *reject* state. The reject state (not visible in Figure 5) is reached always when the next character is not part of the language or it does not infer a valid state transition.

According to the semantics of the XQuery Data Model, each value of an XML node, if it does not evaluate to the reject state, is a potential accepted value. That is, because of the mixed content semantics: a non-final state may be evaluated to a final state if its siblings are considered. Therefore, the first step of creating the typed range-lookup index is to associate each XML node with a state. More specific, during the creation of the index, the lexical value of each text node is fed to the FSM. The FSM returns the state at which the recognition process has stopped. For example, if the lexical value of the given text node is `"E+93 "`, state $s4$ is returned. Similarly, if the lexical value is `" +32.3"` then the state labeled $s11 + s5$ is returned (state is not shown in Figure 5, but implied by the jump from d' to d). Finally, if the lexical value is not a (potential) valid representation of the typed value, for example `"42 text"`, then the reject state is returned. The result of this first step of the index creation is that each text node is assigned a state accordingly to the returned state of the FSM. Notice, that since the total number of states is small – in the case of doubles only 60 – the state can be saved with only one byte for each node. Moreover, the nodes that are evaluated to the reject state, which will be in most cases the majority, do not need to store any state – the absence of a state signifies the reject state.

The next step for creating (or updating) the index is to determine the state of the intermediate nodes. The desire is to be able to efficiently compute the state of an intermediate node without reconstructing the lexical representation of that node. Moreover, it should be possible to early reject intermediate nodes that do not have a valid lexical representation and thus avoid unnecessary traversal of the XML tree. This desire is fulfilled by defining a *state combination table* (SCT). The SCT is a succinct representation of all possible valid combinations of the states. Valid combinations are the ones that do not result in a reject state. The SCT for the double type is depicted in Figure 6. The size of the complete table is 60×60. However, most of the combinations result to a reject state. The succinct representation of the SCT omits all pairs that result in a reject state. Out of the 3600 different combinations, only 389 are non-reject. Moreover, since each state can be represented by a byte, the SCT consumes even less memory space. Finally, the normalization of the FSM described in the beginning of this section ensures that such a SCT always exists.

To facilitate fast range lookups a clustered (B-tree) index is built on top of the typed values which are associated with the id of the node they belongs to. In addition, a second index is built on the node ids. This index is used during the creation and updates of the typed value index for retrieving the state of a node id. The keys of the (B-tree) index are tuples of the form `[value, state, node id]`[2]. Moreover, some nodes may have a (potential) valid state but for storage efficiency do not need to store a value. For example if the state is $s2$, then there is no valid double value, and the state label itself is sufficient to deduce the lexical representation, namely `"E-"` or `"e-"` (which of the two equivalent representations is irrelevant for constructing the combined double value). In addition, the (B-tree) indices are used during creation or update of the typed XML indices to reconstruct the lexical representation of a specific node, without accessing the document data. For example if the value of a node is `"26"` while the state is $s7$, then the lexical representation of that node is `"26E+"`.

5. IMPLEMENTATION DETAILS

In this section we present the implementation details for creating and updating the indices by employing the functionality and data structures introduced in the previous sections. We have implemented both indices in the context of the open-source XML database system MonetDB/XQuery [6]. Although some details are system specific and are tightly coupled with the internal data structures of the XQuery engine at hand, we present the skeleton of the algorithms as generic as possible. In general, only small adjustments would be necessary to successfully apply these ideas to other systems.

5.1 Index Creation and Updates

Internally, MonetDB/XQuery stores an XML document in such a way that permits efficient depth-first traversal. This is achieved by employing a range encoding on the documents nodes, similar to the pre/post encoding. For more details we refer the reader to [1]. It is realistically assumed that every XQuery engine provides a similar interface for efficient traversal over the XML document tree.

All indices are created and updated with the same skeleton algorithm, only the functions called in each case are changed. The algorithm is based on a depth-first traversal of the XML document and since all indices are independent of each other,

[2]depending the design of the XQuery engine – focusing on space or computational efficiency – the second (B-tree) can be clustered also, thus having tuples of the form `[value, node id]` and `[node id, state]`

```
Index Creation
input: a sequence of all XML text nodes of a document as ctx
output: hash value or state for all XML nodes of the document

01: /* initialize variables */
02: init all nodes.field to 0 or undef
03: cur_node = DFS.getRoot();
04: /* loop while all text nodes are consumed */
05: while (ctx.hasNext())
06:       if (ctx.current == cur_node)
07:          cur_node.field = $H$(cur_node)|FSM(cur_node);
08:          ctx.next();
09:       else if (ctx.current descendantOf cur_node)
10:           stack.push(cur_node);
11:           cur_node = DFS.nextChildNode();
12:       else if (ctx.current siblingOf cur_node)
13:           father_node = DFS.getFatherNode();
14:           father_node.field =
               = C(father_node.field, cur_node.field)/
                 SCT[father_node.field][cur_node.field];
15:           cur_node = DFS.nextSiblingNode();
16:       else
17:           pop_node = stack.pop();
18:           pop_node.field =
               = C(pop_node.field, cur_node.field)|
                 SCT[pop_node.field][cur_node.field];
19:           cur_node = pop_node;
20: /* empty stack with visited nodes */
21: while (stack.notEmpty())
22:      pop_node = stack.pop();
23:      pop_node.field =
          = C(pop_node.field, cur_node.field)|
            SCT[pop_node.field][cur_node.field];
24:      cur_node = pop_node;
```

Figure 7: Algorithm for Index Creation

creating and updating multiple defined indices can be done simultaneously with only one pass.

Figure 7 depicts the algorithm for creating both the string equality index and the XML typed range index. Intuitively, the algorithm works as follows. The depth-first traversal starts at the root of the document until all text nodes are visited (lines 3-5 of Figure 7). The traversal is guided downwards until the first text node is found, while each visited intermediate node is pushed into a stack (lines 9-11). When a text node is located, function H or the FSM is called (lines 6-8). Next, and to locate the next text node, either the node on the head of the stack is popped (line 16), or the sibling nodes of the current node are considered (line 12). These cases traverse the XML document tree either upwards or rightwards to locate the next text node. However, in both cases the parent node of the current node has to be updated with the combined hash value or state. This is achieved by invoking function C or probing the SCT (lines 13-15 and 17-19). Finally, and after all text nodes are visited, the stack is emptied and all nodes popped from the stack are updated accordingly (lines 21-24). Notice that the functions calls on the DFS module appearing in Figure 7 (e.g., `DFS.nextSiblingNode()`) are always evaluated against the current node.

The update algorithm is outlined in Figure 8. It works similar to the create algorithm in Figure 7. The first difference between the two algorithms is that when a new node is added in the stack, its field is reset since it is not valid any more (line 9). This is, because if the depth-first traversal reached that node, it means that some of its descendants have been updated. The second difference is that when a node is popped from the stack, its new hash value or state is the combination of all of its children. In other words, its

```
Index Update
input: a sequence of updated XML text nodes as ctx
output: updated hash value or state for all affected XML nodes

01: /* initialize variables */
02: cur_node = DFS.getRoot();
03: /* loop while all text nodes are consumed */
04: while (ctx.hasNext())
05:       if (ctx.current == cur_node)
06:          cur_node.field = $H$(cur_node)|FSM(cur_node);
07:          ctx.next();
08:       else if (ctx.current descendantOf cur_node)
09:           cur_node.field = 0|undef;
              stack.push(cur_node);
10:           cur_node = DFS.nextChildNode();
11:       else if (ctx.current siblingOf cur_node)
12:           cur_node = DFS.nextSiblingNode();
13:       else
14:           pop_node = stack.pop();
15:           while (DFS.hasSiblingNode())
                   cur_node = DFS.nextSiblingNode();
                   pop_node.field =
                   = C(pop_node.field, cur_node.field)|
                     SCT[pop_node.field][cur_node.field];
16:           cur_node = pop_node;
17: /* empty stack with visited nodes */
18: while (stack.notEmpty())
19:      pop_node = stack.pop();
20:           cur_node = DFS.leftMostSibling();
              while (DFS.hasSiblingNode())
                   cur_node = DFS.nextSiblingNode();
                   pop_node.field =
                   = C(pop_node.field, cur_node.field)|
                     SCT[pop_node.field][cur_node.field];
21:      cur_node = pop_node;
```

Figure 8: Algorithm for Index Updates

combined hash value or state must be recomputed across all its immediate children (lines 14-16 and 19-21). A side effect of this recomputation is that during the traversal of siblings nodes (line 11) – contrary to the create algorithm – there is no need to update the father node, this will happen eventually when that node is popped from the stack in a later step.

The algorithm presented in Figure 8 expects only updates on the value of a text node. However, in the case of a node or subtree deletion, a slight change in the algorithm solves the problem. More precisely, the algorithm gets as input the node that served as the root of the subtree that was deleted. In that case, the text value of that node –after the deletion of the subtree– is either the empty string or a new value. In either case, the update algorithm is invoked with the new value of the node, oblivious of the deleted subtree.

5.2 Transaction Management

A final consideration for the update implementation is transaction management overhead, in particular its locking requirements. A general challenge in XML value indexing is that the value of a node is (potentially) influenced by all its descendants. This implies that each update may impact the root node, and locking the root for each transaction can easily become a bottleneck.

A first remark on the proposed typed XML range indices, is that in typical XML documents, only leaf nodes and not intermediate nodes (let alone the root) have a type such as *xs:double* (see also Table 1). The proposed indexing scheme only stores nodes with a potential valid typed lexical value, which in general means that it contains only leaf nodes. Therefore, an adaptive locking scheme that supports fine-

Data	Size (MB)	#Nodes	#Text Nodes		#Double Values		#non-leaf
XMark1	112	4,690,640	3,024,328	(64%)	377,123	(8%)	0
XMark2	224	9,394,467	6,056,817	(64%)	754,936	(8%)	0
XMark4	448	18,827,157	12,138,505	(64%)	1,514,227	(8%)	0
XMark8	896	37,642,301	24,269,192	(64%)	3,026,029	(8%)	0
EPAGeo	170	6,558,707	4,372,404	(66%)	517,862	(7%)	0
DBLP	474	34,799,707	23,198,402	(66%)	3,748,565	(10%)	21
PSD	685	58,445,809	37,139,989	(63%)	2,441,791	(4%)	902
Wiki	2024	94,672,619	53,564,889	(56%)	104,059	(0.1%)	0

Table 1: Statistical information for the data sets

grained locks and is able to gradually enlarge the lock granularity [3] should work rather efficiently with this index.

For our other proposal concerning the equality index on strings, we do have the situation that all XML nodes have a string value and must be indexed, the root node inclusive. Each single update changes the hash value of all its ancestors, thus always affects the root node! However, in the context of structural updates for the MonetDB/XQuery system [2] it was shown that locking ancestors can be avoided if updates are *commutative*, and the combination function C has indeed been developed specifically with this property in mind. It is in fact possible to avoid locking any ancestors of updated nodes during transaction processing until the commit point. A committing transaction should re-read the latest value of all ancestor nodes of an update (and their direct children, per the update algorithm) to recompute their new hash values. Even if siblings of the updated node where changed in the meantime and thus affected these ancestors, the commutativity of the C function ensures that we will compute the correct hash values in the end.

6. EXPERIMENTAL EVALUATION

In this section we present our experiments for evaluating the index creation time, the disk storage overhead, and the maintenance overhead (i.e., update time), for both the equality string and the range typed index. We also studied the collisions introduced by the hash function H to asses the *stability* of the string index and identify which cases of abnormal input text values affect this stability.

The experiments were conducted on an Intel^{R} Core^{TM}2 Quad CPU Q6600 machine, running at 2.40GHz with 4MB cache memory. The machine had 8GB of RAM and 2 hard disks with active raid level 0, capable of reading(writing) from(to) disk with approximately 100MB/sec.

We used 8 different documents of varying size and structure. The first 4 documents were synthetically created with XMark[3] with scale factors 1, 2, 4, and 8, respectively. The remaining 4 documents reflect *"real life"* data, since they were download from online databases. The data set EPAGeo[4] contains geospatial data, the PSD[5] dataset contains protein sequence data, while DBLP[6] and Wiki[7] contain text data about publications and abstracts of articles, respectively. Table 1 details the size (in MBs) of each data set before shredding in the database, the number of nodes in the document, the number of text nodes, and the number of text nodes that have a (potential) valid double lexical representation. Next to the number of text and double nodes, the percentage compared to the total nodes of the data set is given to ease the comparison. The last column depicts the number of non-leaf nodes that have a (potential) valid double value. The datasets generated with XMark do not have any such nodes, while only the DBLP and PSD datasets have a few number of them. This observation strengths our claim that intermediate nodes that cast to a specific XML type is a rare phenomenon, nevertheless an XML value index should respect the semantics of the XQuery Data Model. Our indices are semantically correct, and, as we will illustrate in this section, this is achieved without introducing any significant overhead.

The first set of experiments study the time and space overhead for creating the indices during shredding, that is when the document is processed and stored in the database. All runs were done in cold memory – none of the XML data resided in memory – thus the total shredding time includes the time needed to read the data from disk. We repeated the same experiments three times and report the average times. The deviation of each run from the average time was minor (less than 10ms). The two upper graphs of Figure 9 list the time in milliseconds needed by MonetDB/XQuery to shred the document, and the time needed by our create algorithm to construct the string and double indices. The bars in Figure 9 give a visual of the overhead percentage introduced in the shredding process from the index creation. For the string index, the overhead never exceeds 10% in the worst case, while for half of the cases is less than 5%. For the double index, the creation time overhead is less than 2% in all cases. This is expected, since the combination step is cheaper than that of the string index – probing an array vs. invoking a function.

The lower part of Figure 9 depicts the storage consumed by the string and double index compared to the storage demands of the database for the specific documents. The storage needs of the string index is at most 20% (e.g., EPAGeo, PSD) over the total document and 10% in the best cases (e.g., XMark, Wiki). The difference is explained by the distribution in each document of the string data and the number of nodes. Small number of text nodes with large size of string data resort to less storage consumption than large number of text nodes with few data in each node. On the other hand, the storage demands of the double index is limited. It never exceeds 2-3% of the total size of the database. This is mainly because a) each state is only 1 byte and b) there are few text nodes that have a valid double lexical representation compared to the total number of nodes.

We next evaluate the update performance of the indices. The update queries were created by first defining the number

[3]http://www.xml-benchmark.org/
[4]http://www.epa.gov/enviro/geo_data.html
[5]http://pir.georgetown.edu/
[6]http://dblp.uni-trier.de/xml/
[7]http://download.wikimedia.org/

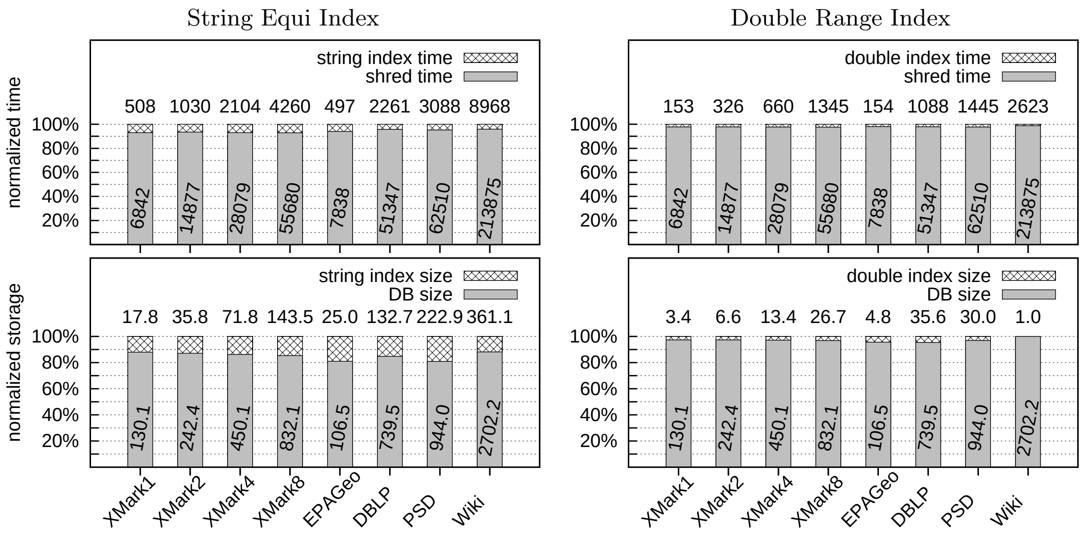

Figure 9: String and Double Index Creation Time and Storage Overhead

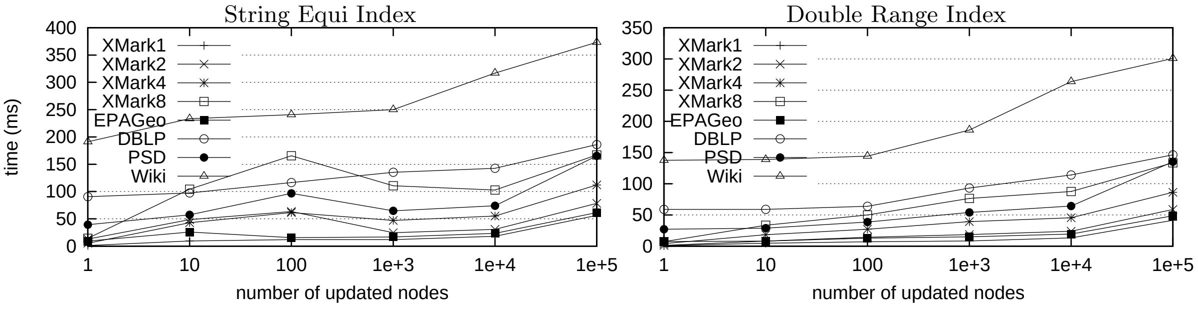

Figure 10: Update Time for String and Double Index

of text nodes whose values should be updated, and then randomly picking the specified number of the text nodes for each document in the database. The number of updated nodes varied from 1 to 1 million. Each update query was run 20 times for each document and the average time is reported. Figure 10 depicts the update times observed for both the string and the double indices. As expected, because of the faster combination step, the double index performs slightly better than the string index. More importantly, for both indices, even in the case of 1 million updated nodes and for a document of size bigger than 2 Gigabytes (i.e., the Wiki dataset), the update time is less than 400 ms. On the other end, the time for updating a small number of nodes is kept less than 50 ms for the smaller documents. The experimental results show that the indices presented in this work are particularly suited for both cases of a) large number of updated nodes in large datasets, and b) small (transactional) updates with few updated nodes.

Finally, we study the stability of the hash function H, used in the equi-lookup index of strings. Many hash functions produce a fixed size output from an arbitrarily long input. In such a design, there will always be collisions, because any given hash has to correspond to a very large number of possible inputs [5]. Figure 11 depicts for all 8 documents the distribution of the number of distinct strings that are associated with the same hash value. Almost all of the strings produce a different hash value. Less than 1% of the total string values collide with another one for most

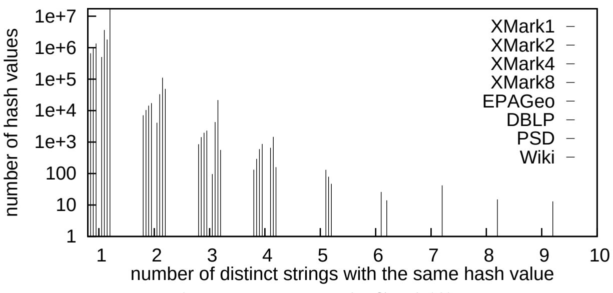

Figure 11: Hash Stability

of the documents, except the last larger ones, namely PSD and Wiki. But even for those the collisions are kept to less than 10%. Especially for the Wiki document there are cases where 9 distinct strings all hash to the same value. This is observed in the case of data representing URLs, were the different characters between two distinct URLs are repeated every 27 positions, while the rest data remain the same to all strings, such as `http://www..` In this case the hash function fails to produce distinct values because each different character is been eliminated by appearing twice exactly after 27 positions.

In conclusion, the experiments presented in this section show that the time and space overhead introduced by our indices is acceptable, while the update time is kept to the minimum and the XQuery Data Model is respected. Our indices are able to support large documents and update millions of nodes while keeping the false positives – due to hash collisions – during query time to a minimum.

7. CONCLUSIONS

We presented a family of generic updatable XML value indices capable of answering equality lookups on string values and range lookups on any XML typed value. These indices are novel compared to prior solutions because they cover the entire document, permit fast updates, and conform with the semantics of the XQuery Data Model. The case were a mixed-content/intermediate node cast to a specific XML type is rare, nevertheless, an XML value index should support it.

We evaluated our design and algorithms in a widely used open-source XML database system, MonetDB/XQuery. The indices presented in this work will be part of a future stable release of MonetDB/XQuery, thus putting our ideas to an every-day test. We intend to expand our work by designing indices capable of answering queries that involve substring matching and regular expressions.

8. REFERENCES

[1] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the ACM SIGMOD*, 2006.

[2] P. A. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proc. of the 2nd XIME-P*, 2005.

[3] M. P. Haustein, T. Harder, and K. Luttenberger. Contest of XML Lock Protocols. In *Proc. of the 32nd VLDB*, 2006.

[4] Kevin Beyer et al. System RX: One Part Relational, One Part XML. In *Proc. of the ACM SIGMOD*, 2005.

[5] D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Publishing Company, 1997.

[6] MonetDB. http://monetdb.cwi.nl.

[7] M. Nicola and B. van der Linden. Native XML support in DB2 universal database. In *Proc. of the 31st VLDB*, 2005.

[8] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In *Proc. of the 30th VLDB*, 2004.

[9] A. Vyas, M. F. Fernandez, and J. Simeon. The Simplest XML Storage Manager Ever. In *Proc. of the 1st XIME-P*, 2004.

[10] XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

[11] XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2/.