A Development of Hash-Lookup Trees to Support Querying Streaming XML

James Cheng and Wilfred Ng

Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology, Hong Kong
{csjames, wilfred}@cse.ust.hk

Abstract. The rapid growth in the amount of XML data and the development of publish-subscribe systems have led to great interest in processing streaming XML data. We propose the QstreamX system for querying streaming XML data using a novel structure, called Hash-Lookup Query Trees, which consists of a Filtering HashTable (FHT), a Static Query Tree (SQT) and a Dynamic Query Tree (DQT). The FHT is used to filter out irrelevant elements and provide direct access to relevant nodes in the SQT. The SQT is a tree model of the input query. Based on the SQT, the DQT is built dynamically at runtime to evaluate queries. We show, with experimental evidence, that QstreamX achieves throughput five times higher than the two most recently proposed stream querying systems, XSQ and XAOS, at much lower memory consumption.

1 Introduction

With the rapid growth in the amount of XML data, processing streaming XML data has gained increasing attention in recent years. Two main and closely related stream processing problems in XML are *filtering* [1, 6, 5, 2, 7, 8, 13] and *querying* [3, 10, 11, 14]. The problem of filtering is to match a set of boolean path expressions (usually in XPath syntax) with a stream of XML documents and to return the identifiers of the matching documents or queries. In querying streaming XML data, however, we need to output all the elements in the stream that match the input query. Apart from natural streaming data used in publish-subscribe systems such as stock quotes and breaking news, it is sometimes more feasible to query large XML datasets in a streaming form, since we need to parse the document only once and keep only data that are relevant to the query evaluation in the memory.

In this paper, we focus on processing XPath queries with streaming XML data. Unlike filtering, querying outputs an element if it matches the input query. The difficulty is that in the streaming environment, we sometimes cannot determine whether an element is in the query result with the data received so far. However, we cannot simply discard the element as its inclusion in the query result may be verified with some element arriving in the future. Therefore, we need to buffer the potential query results. Proper buffer handling for querying XML streams, however, is rather complex, as illustrated by the following example.

Example 1. Consider evaluating the query Q = "//a[.//f]//b/c" on the XML document tree in Figure 1, assuming its elements come as a stream in ascending order of their (numerical) *id*s marked near the circle.

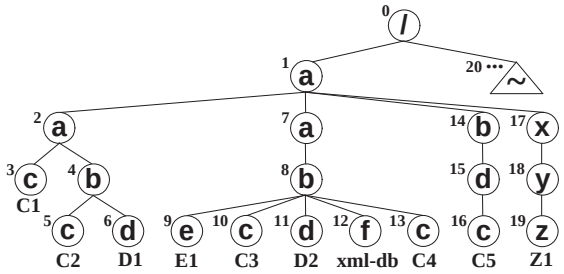

Fig. 1. A Sample XML Document Tree

When the element $\mathtt{c}_5$ (i.e. the node with label "c" and $id = 5$ on the left side of the tree) arrives, we have two node sequences, $q_1 = \langle \mathtt{a}_1, \mathtt{b}_4, \mathtt{c}_5 \rangle$ and $q_2 = \langle \mathtt{a}_2, \mathtt{b}_4, \mathtt{c}_5 \rangle$, matching the main path of Q, i.e. "//a//b/c". However, we cannot output $\mathtt{c}_5$ at this stage, since the predicate, "[.//f]", of both $\mathtt{a}_1$ and $\mathtt{a}_2$ have not been satisfied. As this predicate may be satisfied with an f element that comes later, we must *buffer* $\mathtt{c}_5$ for both q_1 and q_2; but *only one copy* of $\mathtt{c}_5$ should be kept in memory as to avoid *duplicate buffering*.

When the end-tag of the element $\mathtt{a}_2$ arrives, $\mathtt{a}_2$ expires and so does the node sequence q_2. Since $\mathtt{a}_2$'s predicate is not satisfied, we need to *remove* the element $\mathtt{c}_5$ buffered for q_2. But $\mathtt{c}_5$ should not be deleted, since it is still being buffered for q_1, which may satisfy Q if there is an f element, descendant of $\mathtt{a}_1$, coming in the stream. Similarly, we buffer $\mathtt{c}_{10}$ for the node sequences, $q_3 = \langle \mathtt{a}_1, \mathtt{b}_8, \mathtt{c}_{10} \rangle$ and $q_4 = \langle \mathtt{a}_7, \mathtt{b}_8, \mathtt{c}_{10} \rangle$. Then when the start-tag of the element $\mathtt{f}_{12}$ arrives, q_1, q_3 and q_4 satisfy Q. Hence, we need to immediately *flush* the element $\mathtt{c}_5$ buffered for q_1 and the $\mathtt{c}_{10}$ buffered for q_3 and q_4. However, we should *flush* $\mathtt{c}_{10}$ *only once*, though it is buffered for both q_3 and q_4.

When $\mathtt{c}_{13}$ arrives, we should not buffer but *output* $\mathtt{c}_{13}$ immediately, since this time the node sequences, $\langle \mathtt{a}_1, \mathtt{b}_8, \mathtt{c}_{13} \rangle$ and $\langle \mathtt{a}_7, \mathtt{b}_8, \mathtt{c}_{13} \rangle$, instantly satisfy Q. Again, we should output $\mathtt{c}_{13}$ *only once* for the two sequences.

Example 1 suggests some important issues in the query processing: (1) *buffering* of potential query results or *outputting* determined query results; (2) the decision of *flushing* or *removing* buffered data; and (3) *duplicate avoidance* in buffering, outputting, flushing and removing. Let us call all these issues collectively as *buffer handling* in our subsequent discussion.

Buffering comes only with the presence of predicates. The query in Example 1 contains only a single atomic predicate but the problem is already very complex. Another important issue is that a substantial amount of elements in a stream are usually irrelevant, however, no existing querying systems have considered filtering out these elements.

We propose the QstreamX system, which attempt to address the above-mentioned challenges with the use of a novel data structure, called *Hash-Lookup*

Query Trees (*HLQT*). HLQT consists of the following three components: a *Filtering HashTable* (*FHT*), a *Static Query Tree* (*SQT*) and *Dynamic Query Tree* (*DQT*). The FHT filters out irrelevant streaming elements and provides direct access to nodes in the SQT that are relevant for the processing of relevant elements. The SQT is a tree model of the input query, based on which the DQT is constructed dynamically at runtime to evaluate queries.

QstreamX has the following desirable features:

Language Expressiveness. QstreamX supports all XPath axes except the sideways axes (i.e. `preceding-sibling` and `following-sibling`). It also supports multiple and nested predicates with `and` and `or` operators, a common set of aggregations, and multiple queries and outputs.

Processing Efficiency. Our algorithm is able to achieve $O(|D|)$ time complexity and $O(|Q|)$ space complexity, where $|D|$ is the size of the streaming data and $|Q|$ is the size of the input query.

Buffering Effectiveness. QstreamX (1) buffers only those data that *must* be buffered for the correct evaluation of the query; (2) flushes or removes buffered data with no delay; and (3) avoids buffering and outputting any duplicate data.

Effective Design. HLQT makes the implementation of QstreamX straightforward. The FHT is realized as a simple array that stores distinct query elements and pointers to the SQT nodes. The SQT is translated directly from the input query by four simple transformation rules, while the DQT is constructed with correspondence to the structure of the SQT.

In the rest of the section, we discuss related work on stream processing. In Section 2, we present the XPath queries supported by QstreamX. We define Hash-Lookup Query Trees and present query evaluation in Sections 3. We evaluate QstreamX in Section 5 and conclude the paper in Section 6.

1.1 Related Work

A number of *filtering* systems [1, 6, 5, 2, 7, 13, 8] have been proposed to process XPath filters on streaming XML documents. XFilter [1] converts queries into separate Deterministic Finite Automata (DFAs), while YFilter [6] eliminates redundant processing on common prefixes in the queries by a single Non-Deterministic Finite Automaton (NFA). XTries [5] also supports shared processing of common subexpressions of the queries by a trie. The throughput of these systems decreases linearly with the number of queries. LazyDFA [2, 7] ensures a constant high throughput by lazily constructing a DFA for the entire workload of queries. However, LazyDFA may require excessive memory for XML data with complex structures. This problem is addressed in [13], which clusters the queries into n DFAs to reduce the number of DFA states and introduces a shared NFA state table to reduce the size of the NFA state table stored in each DFA state. The XPush machine [8] eliminates common predicates by translating the query workload into a deterministic pushdown automaton. Among these systems, only [13] and [8] support almost the same set of queries (except aggregations) as QstreamX. Although we consider the same query language, filtering only outputs the identi-

fier of matching documents or queries and does not require buffering of potential query result.

A closer match to QstreamX is the XAOS algorithm [3], which translates an XPath query into a tree and uses an extra graph to support the `parent` and `ancestor` axes by converting them into forward axes. The graph determines which set of elements (and with what depth) to look for in the incoming stream. The tree is used to maintain a structure to keep track of the matched elements. However, the query results are only determined at the `ROOT` of the structure, i.e., at the end of the stream, while HLQT outputs an element no later than when its inclusion in the query result is decided. Keeping the matched data until the end of a stream also does not scale, especially because streaming data is unbounded. Moreover, features such as aggregations, `or`-expressions and multiple queries are infeasible in XAOS's approach.

The filtering systems [2, 7, 13, 8] guarantee a constant high throughput using a hash algorithm to access directly relevant states for processing each element. However, direct access to relevant states or nodes using hash-lookup is considerably complicated by buffer handling in the querying problem. In fact, all existing querying systems need to search for matching transitions or relevant nodes for each (including irrelevant) streaming element. Our proposed HLQT adopts a hash-lookup strategy, which is natural to filter out irrelevant elements and provide direct access to nodes relevant for processing relevant elements.

2 QstreamX Query Expressions

We support a practical subset of XPath 2.0 queries with extended aggregations, whose Extended Backus-Naur Form (EBNF) is shown in Figure 2.

```
Q  ::= /LP (/OE)?
LP ::= LS | LS/LP
LS ::= AX::(tag | ∗) P? | (@attribute | @∗) CP?
AX ::= self | child | descendant | descendant-or-self | parent |
          ancestor | ancestor-or-self
P  ::= [P (and | or) P] | [LP CP?]
CP ::= OP literal | [[. OP literal] (and | or) [. OP literal]]
OP ::= > | < | >= | <= | = | != | contains | starts-with
OE ::= text() | count()  | sum()  | avg() |  max() | min()
```

Fig. 2. EBNF Grammar of QstreamX Queries

3 Hash-Lookup Query Trees

We now define the three components of *Hash-Lookup Query Trees* (*HLQT*): the *Static Query Tree*, the *Dynamic Query Tree* and the *Filtering HashTable*.

The Static Query Tree The *Static Query Tree* (*SQT*) is a tree model of the input query constructed by four transformation rules, as depicted in Figure 3, where elements in dotted line are optional components. The transformation rules are derived directly from the EBNF of the language presented in Figure 2.

We now explain the four transformations that are used to construct the SQT.

(a) LocationStep Transformation. A location step is transformed into an *SQT node*, or a *snode* for short, which is a triplet, (*axis*, *predicate*, *dlist*), where *axis* is the axis of the location step; *predicate*, if any, is handled by Predicate Transformation; and *dlist* is a list of DQT node pointers that provide direct access to the DQT nodes. A *dlist* is initially empty, since node pointers are added to the *dlist* at runtime during query evaluation.

(b) LocationPath Transformation. A location path is a sequence of one or more location steps. Therefore, LocationPath Transformation is just a sequence of one or more LocationStep Transformations, where a *snode* is connected to its parent by its *axis*.

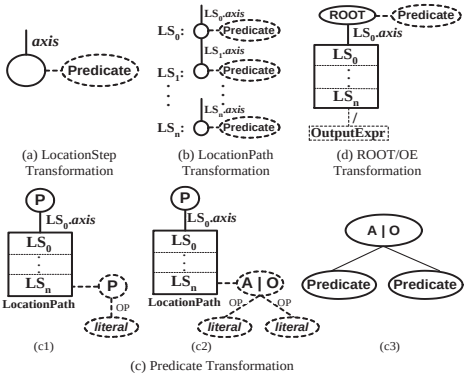

Fig. 3. SQT Transformation Rules

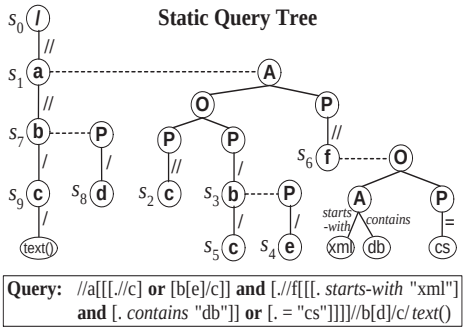

Fig. 4. The SQT of an Example Query

(c) Predicate Transformation. To facilitate efficient predicate processing, we require predicates be fully parenthesized when they are joined by the logical operators. We then model the predicates as a binary tree, called a *Predicate Binary Tree* (*PBT*). A node in the PBT is called an *SQT predicate node*, or a *spnode*, which is one of the following three types: `O` (for `or`-expression), `A` (for `and`-expression) and `P` (an encapsulation of other predicate). Value comparison, if any, is also modelled by a `P`-*spnode*, by an `A`-*spnode* or by an `O`-*spnode* for multiple value matches.

We classify predicate transformation into the following three categories: (1) an atomic predicate is transformed by applying LocationPath Transformation on the location path in the predicate, as shown in Figure 3(c1) and 3(c2); (2) a nested predicate is transformed by applying Predicate Transformation recursively; and (3) an `and`/`or` expression is transformed by applying Predicate Transformation on both sides of the logical operator, as shown in Figure 3(c3).

(d) ROOT/OE Transformation. This transformation is carried out in two steps. The first step is at the beginning of the SQT construction, we create the root of the SQT. The second step is at the completion of the SQT construction, we create a node, called the *output node*, in order to model the output expression of the query.

Let s be a *snode*. If s has an ancestor that is a *spnode*, then we say s is *under* a PBT. Note that s is not part of the PBT, since a PBT consists of only *spnode*s. If the root of a PBT is connected to s, then the PBT is the PBT of s. We say that s is the *parent* of another *snode*, s', if s and s' are connected by the *axis* of s', and that s is the *indirect-parent* of s', if s and s' are connected by a path of *spnode*s in the PBT of s.

The *primary path* of the SQT is the path that still remains when all PBTs and all *snode*s under the PBTs are removed. For example, in Figure 4, the nodes s_1, s_3, s_6 and s_7 have a PBT, while the nodes s_2, s_3, s_4, s_5, s_6 and s_8 are under a PBT; s_1 is the parent of s_7 but the indirect-parent of s_2, s_3 and s_6; $\langle s_0, s_1, s_7, s_9 \rangle$ is the primary path. Moreover, if a *snode* is not on the primary path, then it is under a PBT. Note that there may be more than one primary path in the DQT, if the streaming data is recursive with respect to an *axis* on the primary path of the SQT. The dot notation $a.b$ means that b is the component of a. For example, $s.dlist$ refers to the *dlist* of s.

The Dynamic Query Tree The *Dynamic Query Tree* (*DQT*) is constructed dynamically at runtime to simulate the execution of query evaluation. We use the SQT to guide the construction of the DQT and to provide direct access (using the *dlist*s) to nodes in the DQT that are relevant for the processing of a streaming element. We now detail the structure of the DQT, with reference to the SQT.

Like the SQT, there are two types of nodes in the DQT: *DQT node* (*dnode*) and *DQT predicate node* (*dpnode*). Each *dnode* (*dpnode*) corresponds to a unique *snode* (*spnode*) and the relationship between the *dnode*s (*dpnode*s) is the same as that between the corresponding *snode*s (*spnode*s).

A *dnode*, d, is a triplet, (*depth*, *blist*, *flag*), where *depth* is the depth of the corresponding XML element in the streaming document, and the *blist* and the *flag* are used to aid buffer handling and predicate evaluation. The content of $d.blist$ is described as follows:

- If d is on the primary path, then $d.blist$ is either $\emptyset$ or a list of pointers to where query results are buffered.
- If d is under a PBT, then d is used to evaluate a predicate and hence no data need be buffered for d. However, we assign a special value, ρ, to $d.blist$ so that we can immediately identify whether a *dnode* is under a PBT or on the primary path during query processing.

The *flag* is either `T` or `F`, which has different meanings:

- If d is on the primary path (i.e. $d.blist \neq \rho$), :
  Case of $d.flag =$ `T`. The predicates of all d's ancestors and d are satisfied.
  Case of $d.flag =$ `F`. The predicate of some of d's ancestors has not been satisfied.
- If d is under a PBT (i.e. $d.blist = \rho$):
  Case of $d.flag =$ `T`. All d's descendants are satisfied.
  Case of $d.flag =$ `F`. d has some descendant not satisfied.

When we say that a *dnode*, d, is satisfied, we mean that the predicates of all d's descendants and d are satisfied. When we say that d's predicate is satisfied, we mean that d's PBT is evaluated to be true (and deleted), but it does not imply that the predicates of d's descendants are all satisfied. A *dpnode* is one of the following types: `P`, `A` (i.e. `and`), `O` (i.e. `or`), `L` (i.e. left) and `R` (i.e. right), where `L` (or `R`) indicates that the left (or right) side of the `and`-predicate has been satisfied and only the right (or left) side needs to be processed.

The Filtering Hashtable The Filtering HashTable (FHT) filters out all streaming elements that do not match any element in the query. A hash value is generated for each distinct element or attribute label in the query. The labels are then stored in the corresponding hash slot. Collision is handled by chaining. In practice, collisions are very rare in QstreamX, since we use a hashtable of default size 1024 (only a few KB memory size), while most XML datasets have less than 200 distinct elements.

To provide direct access to *snode*s that match a streaming element, a list, called the *slist*, is kept in each hash slot. An element of the *slist* is a triplet, (*sparent*, *schild*, *dp*), where *sparent* and *schild* are two *snode* pointers, and *sparent* is either the parent or indirect-parent of *schild*; and *dp* is a list of `L` or `R` symbols to represent the left or right direction, respectively, from *sparent* to *schild*, if *sparent* is the indirect parent of *schild* and *sparent*'s PBT has more than one *spnode*s; *dp* is denoted by $\emptyset$ otherwise.

Figure 5 shows the *slist* of the six elements, `a`, `b`, `c`, `d`, `e` and `f`, of the query in Figure 4. For example, `b`'s *slist* has two elements since there are two `b`s in the query. In both *slist*-elements, the *schild*s, s_7 and s_3, model `b`; while the *sparent*, s_1, is the parent of s_7 but the indirect-parent of s_3. The first *dp* is $\emptyset$ since we can reach s_7 from s_1 directly, while the second *dp*, `LR`, shows that from the root of s_1's PBT, we reach s_3's parent by going left and then right.

`a`:{(s_0,s_1,$\emptyset$)}; `d`:{(s_7,s_8,$\emptyset$)}; `e`:{(s_3,s_4,$\emptyset$)}; `f`:{(s_1,s_6,`R`)};
`b`:{(s_1,s_7,$\emptyset$),(s_1,s_3,`LR`)}; `c`:{(s_7,s_9,$\emptyset$),(s_3,s_5,$\emptyset$),(s_1,s_2,`LL`)}.

Fig. 5. The *slist* of the Query in Figure 4

4 QstreamX Query Processing

Consider the query shown in Figure 4 on the XML document presented in Figure 1. For brevity, we use $\mathtt{l_i}.S$ to denote the S event (same for A, T and E) of the element, whose label is `l` and id is `i`, in Figure 1. For example, $\mathtt{a_1}.S$ refers to the S event of $\mathtt{a_1}$. Throughout, we use s_i to denote a *snode* in the SQT (see Figure 4) and d_i to denote a *dnode* in the DQTs (see Figures 6(a)-6(f)).

(a) Basic DQT Construction. We first create the root of the DQT, d_0 = (0,$\emptyset$,`T`), and add d_0's pointer to the *dlist* of the corresponding *snode*, s_0. On the arrival of $\mathtt{a_1}.S$, we apply hashing on the label, `a`, and access `a`'s *slist* (c.f. Figure 5), {(s_0,s_1,$\emptyset$)}, that is stored in `a`'s hash slot. We use s_0's pointer in `a`'s *slist* to access s_0 and then use d_0's pointer in $s_0.dlist$ to access d_0. From d_0 we create its child, d_1 = (1,$\emptyset$,`F`), to correspond to s_0's child, s_1. We set $d_1.blist$ to $\emptyset$, since s_1 is on the primary path, and $d_1.flag$ to `F`, since s_1 has a PBT. We then construct the PBT for d_1 according to the PBT of s_1 and insert the pointer to d_1 into

$s_1.dlist$. In the same way, for the next (recursive) event $\mathtt{a}_2.S$, we create another child, d_2, for d_0. In the following discussion, when we create a *dnode*, we also construct its PBT, if any; and after the *dnode* is created, its pointer is inserted into the *dlist* of its corresponding *snode* to provide direct access. We show the DQT constructed so far in Figure 6(a), in which we also show all the non-empty *dlist*s of the *snode*s.

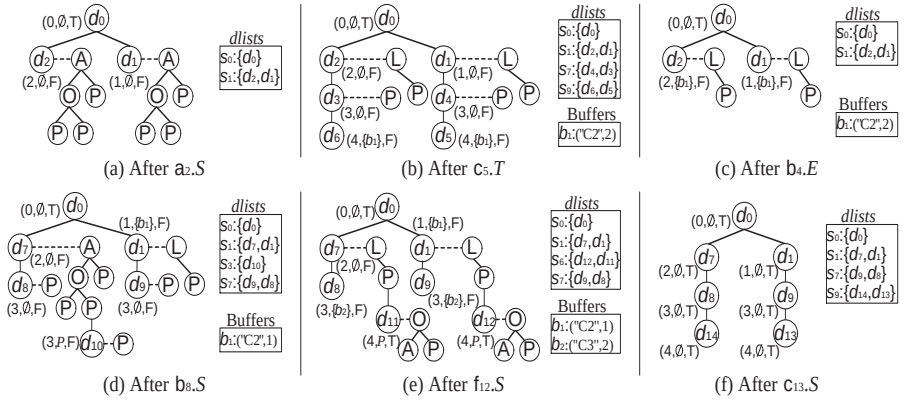

Fig. 6. The DQTs for Processing the Query in Figure 4 on the XML Doc in Figure 1

(b) Predicate Processing (Bubble-Up). The next streaming event is $\mathtt{c}_3.S$ and we have three elements in c's *slist*: $\{(s_7,s_9,\emptyset),(s_3,s_5,\emptyset),(s_1,s_2,\mathtt{LL})\}$. However, the *dlist*s of the parent *snode*s, s_7 and s_3, are empty, which implies that s_7 and s_3 have not been matched. Hence, we only process $(s_1,s_2,\mathtt{LL})$ and access d_2 and d_1 via their pointers in $s_1.dlist$. We then use dp, i.e. LL, to start from the root of d_2's PBT, p_r, to reach the leftmost leaf *dpnode*, p_l. Since s_2 has no PBT and child, $\mathtt{c}_3.S$ satisfies s_2. Thus, no *dnode* is created but we bubble the satisfaction from p_l up the PBT. The bubble-up immediately satisfies p_l's parent since it models an `or`-predicate. Hence, we continue bubbling up to p_r, which is an `and`-predicate. We change $p_r.type$ to L to indicate that the left child of p_r is evaluated to be true. In the same way, we evaluate d_1's PBT with $\mathtt{c}_3.S$. We update the DQT in Figure 6(b). (We ignore d_3-d_6 for the time being.)

(c) Elimination of Redundant Processing. We do not process $\mathtt{c}_3.T$ and $\mathtt{c}_3.E$, since the *dlist*s of s_9, s_5 and s_2 are empty, implying that no *dnode* exists to process $\mathtt{c}_3.T$ and $\mathtt{c}_3.E$. Note that $\mathtt{c}_3.T$ and $\mathtt{c}_3.E$ are indeed redundant for processing the query .

Then it comes $\mathtt{b}_4.S$. Using $(s_1,s_7,\emptyset)$ in b's *slist* we access s_1 and then d_2 and d_1. From d_2 and d_1 we create their respective child, d_3 and d_4, corresponding to s_1's child s_7. However, for the other element, $(s_1,s_3,\mathtt{LR})$, in b's *slist*, when we use dp to process s_3, we find that s_3 belongs to the satisfied part of a PBT, since the first component of dp, i.e. L, matches the type of the root of both d_2's PBT and d_1's PBT. This is also a part of QstreamX's mechanism to eliminate redundant processing. In the same way, we also skip the processing of last two *slist*-elements in c's *slist* for the next streaming element, $\mathtt{c}_5$.

(d) Buffering. We only need to process the *slist*-element, (s_7,s_9,$\emptyset$), for $\mathtt{c_5}$. For $\mathtt{c_5}.S$, we access d_4 and d_3 via $s_7.dlist$, and create their respective child, d_5 and d_6, corresponding to s_9. For $\mathtt{c_5}.T$, we apply hashing on the label, `c`, obtained from the stack top. We then access d_6 and d_5 via $s_9.dlist$. Since s_9's child is the output node and both d_6 and d_5 have no PBT, $\mathtt{c_5}.T$ is a potential query result. We create Buffer b_1 to buffer $\mathtt{c_5}.T$, i.e. "`C2`". Then we insert the pointer to b_1 into both $d_6.blist$ and $d_5.blist$, and increment $b_1.counter$ twice. We show the updated DQT and the Buffer in Figure 6(b).

(e) Uploading. To process $\mathtt{c_5}.E$, we use (s_7,s_9,$\emptyset$) to access s_9 and then access d_6 and d_5, via $s_9.dlist$. We upload $d_6.blist$ and $d_5.blist$ to their parents d_3 and d_4 respectively. Then, we delete d_6 and d_5, and remove their pointers from $s_9.dlist$.

With $\mathtt{d_6}.S$ and `d`'s *slist*, $\{(s_7,s_8,\emptyset)\}$, we then further delete the PBT of d_4 and d_3, since $\mathtt{d_6}.S$ satisfies s_8. Again, s_8's empty *dlist* avoids the redundant processing of $\mathtt{d_6}.T$ and $\mathtt{d_6}.E$.

To process $\mathtt{b_4}.E$, we upload $d_4.blist$ and $d_3.blist$ to their parents d_1 and d_2 respectively. We then delete d_4 and d_3, and remove their pointers from $s_7.dlist$. We update the DQT and the *dlist*s in Figure 6(c). Note that both $d_1.blist$ and $d_2.blist$ now contain the pointer to Buffer b_1.

(f) Buffer Removing. Then for $\mathtt{a_2}.E$, we access d_2 and d_1 via $s_1.dlist$. We do not upload $d_2.blist$ since d_2 has a PBT, i.e. the predicate is not satisfied, and hence the data buffered is not a query result with respect to d_2. We access Buffer b_1 via b_1's pointer in $d_2.blist$ to decrement $b_1.counter$. Then we delete d_2 and its PBT. We do not process d_1, since $d_1.depth$ does not match the depth of $\mathtt{a_2}.E$.

We then create another child, d_7, for d_0 with $\mathtt{a_7}.S$. Then corresponding to s_7, $\mathtt{b_8}.S$ creates d_8 and d_9 as child of d_7 and d_1 respectively. Although $\mathtt{b_8}.S$ is not processed for d_1's PBT, we create d_{10} to evaluate s_3, as shown in Figure 6(d).

Then $\mathtt{e_9}.S$ satisfies s_4 and we delete d_{10}'s PBT, while s_4's empty *dlist* avoids $\mathtt{e_9}.T$ and $\mathtt{e_9}.E$ being redundantly processed. Next $\mathtt{c_{10}}.S$ creates a child for d_9 and d_8 respectively, corresponding to s_9. This $\mathtt{c_{10}}.S$ also satisfies s_5, and the satisfaction triggers d_{10}'s satisfaction, which is bubbled up until it updates the type of the root of d_7's PBT to `L`. The last element in `c`'s *slist* is thus not processed, since s_2 belongs to a satisfied part of the PBT.

For $\mathtt{c_{10}}.T$, i.e. "`C3`", we buffer "`C3`" in Buffer b_2. On the arrival of $\mathtt{c_{10}}.E$, the *blist*s are uploaded to d_9 and d_8. Then $\mathtt{d_{11}}.S$ satisfies s_8 and we delete the PBT of both d_9 and d_8. Next, $\mathtt{f_{12}}.S$ creates d_{11} and d_{12} to evaluate s_6, as shown in the updated DQT in Figure 6(e).

(g) Predicate Processing (Trickle-Down) and Flushing. Then $\mathtt{f_{12}}.T$, i.e. "`xml-db`", matches the `and`-predicate in d_{12}'s and d_{11}'s PBT. We bubble up the satisfaction to the `or`-predicate, i.e. the root of d_{12}'s and d_{11}'s PBT. Thus, both d_{12} and d_{11} are satisfied; and the satisfaction is bubbled up and triggers the satisfaction of both d_1's PBT and d_7's PBT. Since d_1 and d_7 are on the primary path, we trickle down the satisfaction of their PBT to their descendants.

The trickle-down starts at d_1, since d_{12}, which is under d_1's PBT, is processed before d_{11}. We first update $d_1.flag$ to `T` and access b_1 via $d_1.blist$ to flush b_1. We then decrement $b_1.counter$ to zero and hence we delete b_1. We also set $d_1.blist$

to $\emptyset$. Then we trickle down to d_1's child d_9, we set $d_9.flag$ to `T` and access b_2 via $d_9.blist$ to flush b_2. We then set $b_2.store$ to "`flushed`" and decrement $b_2.counter$. Then we set $d_9.blist$ to $\emptyset$. When the trickle-down reaches d_8, we access b_2 again via $d_8.blist$. Since $b_2.store$ is "`flushed`", we only decrement $b_2.counter$. We delete b_2 since $b_2.counter$ now becomes zero.

(h) Outputting. Then for $\mathtt{c}_{13}.S$ we create d_{13} and d_{14} as child of d_9 and d_8 respectively, as updated in Figure 6(f). Since $d_9.flag$ and $d_8.flag$ are `T`, $d_{13}.flag$ and $d_{14}.flag$ are also set to `T`. Therefore, when we process $\mathtt{c}_{13}.T$ for d_{14}, we immediately output $\mathtt{c}_{13}.T$ as a query result. We also set a flag to indicate that $\mathtt{c}_{13}.T$ is outputted, so that we do not output it again when we process d_{13} next. The flag is then unset.

Then for $\mathtt{c}_{13}.E$, we delete d_{14} and d_{13}; for $\mathtt{b}_8.E$, we delete d_9 and d_8; for $\mathtt{a}_7.E$, we delete d_7.

(i) Depth Mismatch and Hash-Lookup Filtering. Although s_7 is satisfied again with $\mathtt{b}_{14}$ and $\mathtt{d}_{15}$, $\mathtt{c}_{16}$ does not match the *depth* of the child of s_7 and is hence filtered out. The elements, $\mathtt{x}_{17}$, $\mathtt{y}_{18}$ and $\mathtt{z}_{19}$, have no corresponding hash slots and are hence filtered out. Finally, we delete d_1 when $\mathtt{a}_1.E$ comes, while we delete d_0, i.e. the root of the DQT, to terminate the query processing at the end of the stream.

5 Experimental Evaluation

We evaluate QstreamX on two important metrics for XML stream processing: the *throughput* and the *memory consumption*. We compare its performance with two most recently proposed querying systems, the XSQ system V1.0 [14] and the XAOS system[3]. We use the following four real datasets [12]: the Shakespeare play collection (Shake), NASA ADC XML repository (NASA), DBLP, and the Protein Sequence Database (PSD). We ran all the experiments on a Windows XP machine with a Pentium 4, 2.53 GHz processor and 1 GB main memory.

Throughput Throughput measures the amount of data processed per second when running a query on a dataset. For each of the four real datasets, we use 10 queries, which have a roughly equal distribution of the four types: Q_1 consists of only `child` axis, Q_2 consists of only `descendant-or-self`; Q_3 and Q_4 mix the two axes, but Q_3 consists of a single atomic predicate, while Q_4 allows multiple (atomic) predicates. An example of each type is shown below:

(Q_1): "`/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`"
(Q_2): "`//dataset//author//lastname/text()`"
(Q_3): "`//inproceedings[year > 2000]/title/text()`"
(Q_4): "`//ProteinEntry[summary]/reference[accinfo]`
`/refinfo[@refid =“A70500”]//author/text()`"

The throughput[1] of each system on processing a single query is measured as the average of the throughput of processing each of the 10 queries for each dataset. We also measure the throughput of processing multiple queries (5 and

[1] Since outputting the query results to the screen dominates the processing time, we write the results to a disk file for all systems.

10 queries) by QstreamX, where the input queries are simply each half of the 10 queries and the 10 queries as a whole respectively. However, the Xerces 1.0 Java parser used in XSQ is on average two times slower than the C++ parser used in QstreamX and XAOS. Therefore, we use the *relative throughput* [14], which is calculated as the ratio of the throughput of each system to that of the corresponding SAX parser, to give a comparison only on the efficiency of the underlying querying algorithm.

As shown in Figure 7, QstreamX achieves very impressive throughput, which is about 80% of that of the SAX parser (the throughput for Shake is 78% when the dataset is scaled up by three time); in another word, 80% of the upper bound. Compared with XSQ and XAOS, QstreamX on average achieves relative throughput of 2.7 and 4.5 times higher, respectively. The tremendous improvement made by our algorithm over the XSQ and XAOS algorithms is mainly due to the effective filtering of irrelevant elements by hash-lookup and the direct access to relevant nodes through *slist* and *dlist*. Finally, we remark that the raw throughput of QstreamX is on average 5.4 and 9 times higher than that of XSQ and XAOS, respectively.

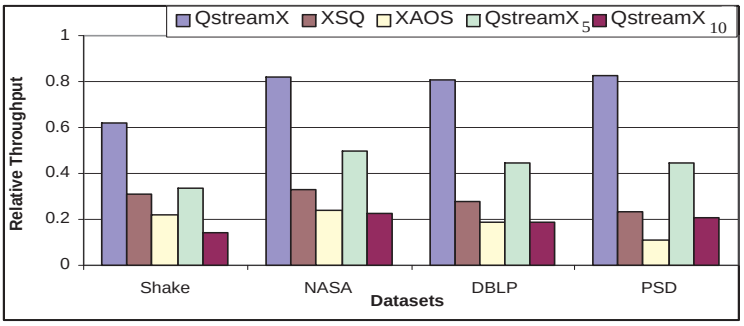

Fig. 7. Relative Throughput

The average relative throughputs of QstreamX on processing 5 queries and 10 queries are 43% and 19%, as denoted by QstreamX$_5$ and QstreamX$_{10}$ respectively in Figure 7. The great drop in the throughput is mainly because 5 and 10 times more potential query results need to be processed and duplicate avoidance has to be performed for 5 and 10 more times. However, this overhead is inevitable for processing multiple queries on XML streams, since we must buffer the potential query results at any given time. Despite of this, we remark that the throughput of QstreamX on 5 queries is still 1.5 times higher than that of XSQ (i.e. a raw throughput of 3 times higher), while that on 10 queries is only slightly lower (but a slightly higher raw throughput).

Memory Consumption We measured roughly constant memory consumption of no more than 1 MB for QstreamX on all datasets and queries (including the two cases of multiple query processing). In fact, a large portion of the memory is used in buffering and in the input buffer of the parser, while the memory used for building the trees is almost negligible. The constant memory consumption

proves the effectiveness of buffer handling, while the lower memory consumption verifies that the size of the DQT is extremely small. The memory consumption of XSQ is also constant (as a result of its effective buffering) but several times higher than that of QstreamX (as a result of a less efficient data structure). The memory consumption of the XAOS system increases linearly, since the algorithm stores both the data and the structure of all matched elements and outputs the results at the end of a stream.

6 Conclusions

We have presented the main ideas in QstreamX, an efficient system for processing XPath queries of streaming XML data, by utilizing a novel data structure, Hash-Lookup Query Trees (HLQTs), which consists of a simple hash table (the FHT) and two elegant tree structures of the SQT and the DQT. We have devised a set of well-defined transformation rules to transform a query into its SQT and discussed in detail how the dynamic construction of the DQT evaluates queries. A unique feature of QstreamX is that it processes only relevant XML elements in the stream by hash-lookup and accesses directly nodes that are relevant for their processing. We have demonstrated that QstreamX achieves significantly higher throughput and consumes substantially lower memory than the state-of-the art systems, XSQ and XAOS.

References

1. M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of VLDB*, 2000.
2. I. Avila-Campillo and et al. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Proc. of PLANX*, 2002.
3. C. Barton and et al. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of ICDE*, 2003.
4. Z. Bar-Yossef, M. F. Fontoura, and V. Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of PODS*, 2004.
5. C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of ICDE*, 2002.
6. Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *ICDE*, 2002.
7. T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of ICDT*, 2003.
8. A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, 2003.
9. V. Josifovski, M. F. Fontoura, and A. Barta. Querying XML Steams. In *VLDB Journal*, 2004.
10. M. L. Lee, B. C. Chua, W. Hsu, and K. L. Tan. Efficient Evaluation of Multiple Queries on Streaming XML Data. In *Proceedings of CIKM*, 2002.
11. B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, 2002.
12. G. Miklau and D. Suciu. XML Data Repository. `http://www.cs.washington.edu/research/xmldatasets.`
13. M. Onizuka. Light-weight XPath Processing of XML Stream with Deterministic Automata. In *Proceedings of CIKM*, 2003.
14. F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of SIGMOD*, 2003.