Querying Streaming XML Data Using Hash-Lookup Query Trees

James Cheng Wilfred Ng
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{csjames, wilfred}@cs.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. While a number of efficient systems have been developed to process XPath filters on XML streams, the performance of existing systems that query streaming XML data is inadequate. We propose the QstreamX system for querying streaming XML data using a novel structure, called Hash-Lookup Query Trees, which consists of a Hashtable, a Static Query Tree (SQT) and a Dynamic Query Tree (DQT). The Hashtable is used to filter out irrelevant elements and provide direct access to relevant nodes in the SQT. Based on the SQT, the DQT is built dynamically at runtime to evaluate queries. QstreamX supports all XPath axes (except the sideways axes), multiple and nested predicates, `and`/`or` *expressions, a common set of aggregations, and multiple queries/outputs. 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 and the development of publish-subscribe systems, processing streaming XML data has gained increasing attention in recent years. Two main and also closely related stream processing problems in XML are *filtering* [1, 9, 7, 2, 10, 18, 11] and *querying* [14, 13, 3, 19, 16, 12]. The problem of filtering is to match a set of boolean path expressions (usually in XPath [8] syntax) with a stream of XML documents and return the identifiers of the matching documents or queries. Querying streaming XML data, however, outputs all the elements in the stream that match the input query. In this paper, we focus on the problem of querying. Our goal is to *efficiently* evaluate XPath queries on *unbounded* streaming data at *small memory consumption*.

In querying streaming XML data, in general we cannot determine whether or not an element is in the query result with the data received so far. Due to the read-once-only nature of streaming data, we must buffer the element until its inclusion in or exclusion from the query result is verified (with some element that comes later in the stream). However, buffer handling in querying streaming XML data is non-trivial, as illustrated by the following example.

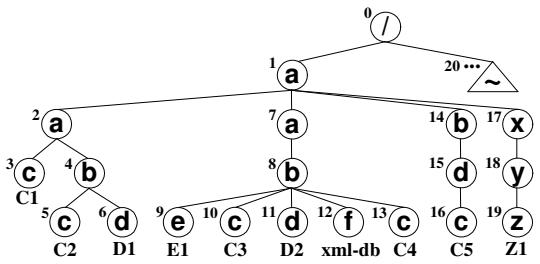


Figure 1. Sample XML Document Tree

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.

When the element $\mathtt{c}_5$ (i.e. the node with label c and *id* $=$ 5) 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. Since 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 $\mathtt{c}_5$ 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 $\texttt{f}_{12}$ arrives, q_1, q_3 and q_4 satisfy Q. Hence, we need to immediately *flush* the $\texttt{c}_5$ buffered for q_1 and the $\texttt{c}_{10}$ buffered for q_3 and q_4. However, we should *flush* $\texttt{c}_{10}$ *only once*, though it is buffered for both q_3 and q_4.

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

□

Example 1 suggests some important issues in the query processing: (1) *buffering* potential query result or *outputting* determined query result; (2) *flushing* or *removing* buffered data as soon as their inclusion in or exclusion from the query result can be decided; 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. In the presence of multiple and nested predicates with the `and`/`or` operators, buffer handling, in conjunction with predicate evaluation, poses significantly greater challenges. 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 addresses all the above-mentioned challenges with the use of a novel data structure, called *Hash-Lookup Query Trees* (*HL-QT*). HL-QT consists of three components: a *Hashtable*, a *Static Query Tree* (*SQT*) and a *Dynamic Query Tree* (*DQT*). The Hashtable 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.

This paper makes the following contributions:

- QstreamX supports all XPath axes except the sideways axes (`preceding(-sibling)` and `following(-sibling)`). It also supports multiple and nested predicates with `and`/`or` operators, a common set of aggregations, and multiple queries/outputs. To the best of our knowledge, our set of queries is the most expressive among those supported by other existing stream filtering and querying systems.

- Our algorithm is the first stream-querying algorithm that achieves $O(|D|)$ 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.

- HL-QT is a very elegant design, which makes the implementation of QstreamX straightforward. The Hashtable 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.

- We evaluate QstreamX on several real datasets and a large set of queries. Our results show that QstreamX achieves throughput at least five times higher than the two most recently proposed stream querying systems, XSQ [19] and XAOS [3], but at significantly lower memory consumption. We also extensively study the factors that affect the performance of QstreamX using a set of complex synthetic datasets and queries. The results prove the robustness of QstreamX in querying streaming XML data.

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 and 4, respectively. We analyze the complexity of QstreamX query processing in Section 5 and evaluate QstreamX in Section 6. We conclude the paper in Section 7.

1.1. Related Work

A number of *filtering* systems [1, 9, 7, 2, 10, 18, 11] have been proposed to process XPath filters on streaming XML documents. XFilter [1] converts queries into separate Deterministic Finite Automata (DFAs), while YFilter [9] eliminates redundant processing on common prefixes in the queries by a single Non-Deterministic Finite Automaton (NFA). XTries [7] 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, 10] 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 [18], 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 [11] eliminates common predicates by translating the query workload into a deterministic pushdown automaton.

Among these systems, only [18] and [11] support almost the same set of queries (except aggregations) as QstreamX. Although we consider the same query language, filtering only outputs the identifier of matching documents or queries and does not require buffering of potential query result.

Many *querying* systems are transducer-based. The XSM system [14] translates an XQuery [6] into a network of XML stream transducers that can be reduced to a single transducer. However, XSM supports only a small, non-recursive subset of XQuery, while other features such as closures and aggregations are not supported. A similar transducer-based system, SPEX [16], supports regular path expressions with qualifiers on well-formed XML streams.

The XSQ system [19] translates an XPath query into a hierarchy of pushdown transducers augmented with buffers. However, XSQ supports only five primitive forms of predicates, since buffer handling significantly complicates the encoding of the logic of the predicates into the automata.

Two tree-based approaches are XAOS [3] and TurboXPath [12]. Both methods translate an input query into a parse tree and support `parent` and `ancestor` axes by converting them into forward axes in a graph. The tree and graph are used to construct structures that keep matched data. XAOS outputs query result only at the end of a stream, while TurboXPath processes predicates and query result at the end of each closure fragment, which can be the entire stream. During the process, all matching structures need to be kept in memory. This may incur prohibitively huge memory consumption for unbounded streaming data and is therefore not practical. Conversely, QstreamX does not delay processing predicates and query result, nor does it keep any extra data in memory.

TurboXPath supports a subset of the for-let-where constructs of XQuery and produces query result as tuples, with the use of multiple output nodes. This subset of XQuery can be easily supported by QstreamX with only a slight modification. However, XAOS and TurboXPath do not support aggregations, `or`-expression and multiple queries.

Multiple queries are discussed in [13], which processes several regular path expressions by a global template that is based on a finite state machine model. However, the regular expressions are evaluated in turn and their commonalities are not exploited. In QstreamX, redundancies in common prefixes is easily eliminated by transforming the multiple queries into a single SQT.

The filtering systems [2, 10, 18, 11] guarantee a constant high throughput using a hash algorithm to access directly relevant states for processing each element. However, direct access to relevant states/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 HL-QT adopts a hash-lookup strategy, which is natural to filter out irrelevant elements and provide direct access to nodes relevant for processing relevant elements.

The studies [4] on lower bounds on the memory requirements of XPath evaluation on XML streams show that most current stream processing algorithms require memory far greater than the lower bound. However, the memory requirement of our algorithm lies closely to the lower bound.

2. Queries Supported by QstreamX

We support a practical subset of XPath [8] queries with extended aggregations, whose 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()
```

Figure 2. Grammar of QstreamX Queries

An *XPath query*, `Q`, is a *location path*, `LP`, followed by an optional *output expression*, `OE`. The `LP` selects nodes in the input XML document by a sequence of one or more *location steps*, `LS`. Each `LS` consists of an *axis*, `AX`, a *node test* and an optional *predicate*, `P`. We support all XPath 2.0 [5] axes except `preceding(-sibling)` and `following(-sibling)`. The node test refers to the matching of the element/attribute label. The predicate of each `LS` can in turn be an `LP` containing more predicates and so on recursively, to refine the set of nodes selected by the `LS`. To write more expressive and useful queries, the `and` and `or` operators are used to join the predicates. The `OE` specifies the format of the query result. QstreamX supports the following `OE`s: (1) *not specified*: the query returns the set of nodes selected by its `LP`; (2) `text()`: only the text contents of the elements in the result set are returned; (3) one of the five aggregation operations.

3. Hash-Lookup Query Trees

In this section, we define the three components of *Hash-Lookup Query Trees* (*HL-QT*): the *Static Query Tree* (*SQT*), the *Dynamic Query Tree* (*DQT*) and the *Hashtable*.

3.1. 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 query language presented in Figure 2.

(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*.

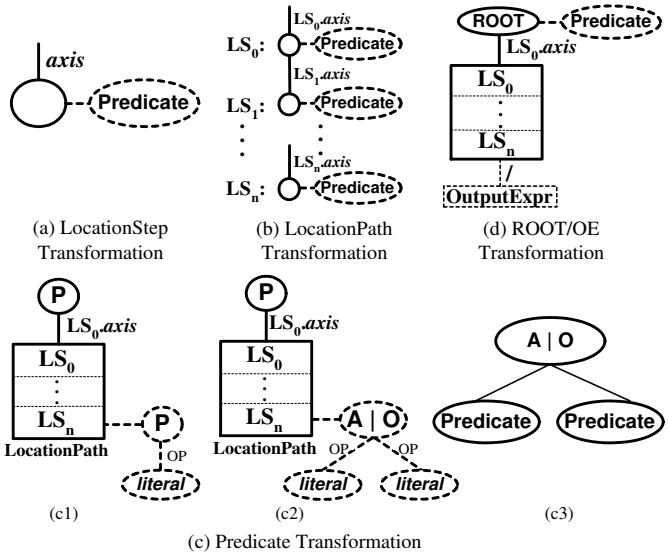


Figure 3. SQT Transformation Rules

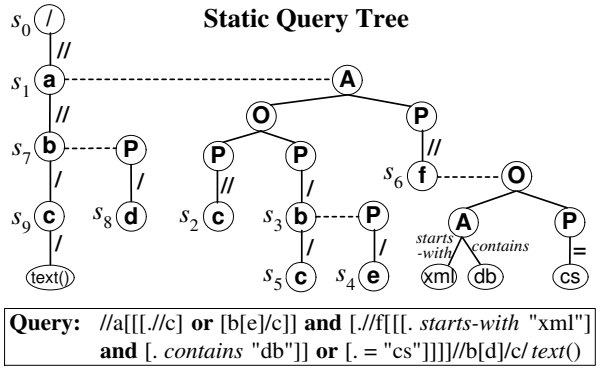
(c) Predicate Transformation. To facilitate efficient predicate processing, we require predicates be fully parenthesized when they are joined by the logical operators. We then model the predicates as a binary tree, called a *Predicate Binary Tree* (*PBT*). A node in the PBT is called an *SQT predicate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

We classify predicate transformation into 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; or (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. At the beginning of the SQT construction, we create the root of the SQT. At the completion of the SQT construction, we create a node, called the *output node*, to model the output expression of the query.

We remark that in case that any reverse axes, i.e. `parent` and `ancestor(-or-self)`, are present in the input query, they are converted into equivalent forward axes (c.f. [17]). For example, "`/descendant::b/ancestor::a`" is converted into "`/descendant-or-self::a[descendant::b]`".

Figure 4 shows the SQT of an XPath query. Note that the *dlist*s of all *snode*s are initially empty. The node identifier, s_i, to the side of each *snode* and the element labels are only included for referencing the *snode*s in later discussion.



Query: //a[[[.//c] **or** [b[e]/c]] **and** [.//f[[[. *starts-with* "xml"] **and** [. *contains* "db"]] **or** [. = "cs"]]]]//b[d]/c/*text*()

Figure 4. The SQT of an Example Query

Notations. We define a few notations used throughout the paper. These notations also apply to the corresponding components in the DQT.

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'; while 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 remains when all PBTs and all *snode*s under the PBTs are removed. 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, as we will show later.

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.

The dot notation $a.b$ means that b is the component of a. For example, s.*dlist* refers to the *dlist* of s.

3.2. 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. The construction of the DQT is central to the query evaluation mechanism of QstreamX and will be discussed in Section 4. 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* $\neq \rho$):
  - d.*flag* = `T`: The predicates of all d's ancestors and d are satisfied.
  - 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* $= \rho$):
  - d.*flag* = `T`: All d's descendants are satisfied.
  - 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` (`and`), `O` (`or`), `L` (left) and `R` (right), where `L` (`R`) indicates that the left (right) side of the `and`-predicate has been satisfied and only the right (left) side needs to be processed.

3.3. The Hashtable

The Hashtable filters out all streaming elements that do not match any element in the query. A hash value is generated for each distinct element/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 *spnodes*; 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`)}.

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

4. Query Processing in QstreamX

In this section, we first discuss the mechanism of querying streaming XML data by the dynamic construction of the DQT. We then discuss the processing of aggregations, wildcards and multiple queries/outputs. Due to space constraints, we limit our discussion to the processing of the `child` and `descendant` axes only, since the reverse axes can be converted into the forward ones and axes containing the `self` component can be handled by a test on the context node.

4.1. DQT Construction

The execution of query evaluation in QstreamX is simulated by the dynamic construction of the DQT. We now discuss how the DQT is constructed at runtime with correspondence to the SQT and the use of hash-lookup.

Hash-Lookup. The streaming XML data is parsed as a sequence SAX-events, which are classified into four types: S (*start-tag*), E (*end-tag*), A (*attribute*) and T (`text()`). The events, S, A and E, carry the label of an element/attribute. We also use a stack to keep the labels so that each T event is given the stack-top label. Thus, we can apply hashing on the label of each SAX-event, x, to perform hash-lookup:

- If x is hashed into an empty slot in the Hashtable, it is filtered out immediately.
- If x is hashed into an non-empty slot, we then process x if its label matches with the label stored in the hash slot, or else we filter out x if x's label does not match.

When x passes the hash-lookup filtering, we access the relevant *snode*s via their pointers in each *slist*-element, (*sparent*, *schild*, dp), of the *slist* kept in the hash slot.

Let s_p and s_c be the *snode* pointed at by *sparent* and *schild* respectively. If x is S/A (T/E) and s_p.*dlist* (s_c.*dlist*) is empty, i.e. no *dnode* has been created to evaluate s_p (s_c), then the root-to-s_p (-s_c) path has not been matched. In this case, we do not process the current *slist*-element but continue processing x with next *slist*-element, if any. Thus, we only process x if the *dlist* of the relevant *snode* is not empty.

If s_p.*dlist* (s_c.*dlist*) is not empty, let d be the *dnode* pointed at by a pointer in the *dlist*. If s_c.*axis* is `child` and d.*depth* is not one less than (not equal to) the depth of x in the streaming document, we do not process x for d. If the depths match, we then process x for d as follows. (In the following discussion, when we say that *we quit*, we mean that we do not process x for d, but process x for the next *dnode* pointer, if any, in the *dlist*.)

Processing of Start-Tag. If x is an S or A event, then d is pointed at by a pointer in s_p.*dlist*. We process x, depending on the position of s_c, as follows:

(Case 1:) s_p is the parent of s_c and d.*blist* is not ρ. Then s_c is on the primary path, since d.*blist* $\neq \rho$ implies that d is on the primary path and so is d's corresponding *snode*, s_p, and so is s_p's child, s_c. We create a new *dnode* to process query result or handle buffering. The creation of a new *dnode* is discussed later in this subsection.

(Case 2a:) s_p is s_c's parent and d.*blist* is ρ. Then s_c is under a PBT, since d.*blist* $= \rho$ implies that d is under a PBT and so is s_p, and so is s_p's child, s_c. If d.*flag* is `T`, then we quit, since d's descendants have been satisfied and need not be reprocessed. Otherwise, if s_c has no PBT and no child, then x immediately satisfies s_c and we update d.*flag* to `T` to indicate that all d's descendants have been satisfied.

If d's predicate has been satisfied, we also invoke BubbleUp (d, $\emptyset$), which will be discussed in Section 4.3, to bubble up d's satisfaction to its ancestors. If s_c has a PBT or a child, then we create a new *dnode* for the (future) processing of the PBT or the child.

(Case 2b:) s_p is the indirect-parent of s_c. Then by definition s_c is under a PBT. If d has no PBT, i.e. the PBT is deleted at its satisfaction, we quit, since a satisfied PBT (i.e. predicate) need not be reprocessed. Otherwise, we start from the root of d's PBT and follow the path indicated by dp to the leaf *dpnode*, p, which corresponds to the parent of s_c. While we are following dp, if we find that a dp component (`L` or `R`) matches the type of a *dpnode* on the path, we quit, since the left or the right side of the `and`-predicate must be satisfied. But if we can reach p and if s_c has no PBT and no child, then x immediately satisfies s_c and we invoke BubbleUp (d, p) to bubble up the satisfaction from p to its ancestors. If s_c has a PBT or a child, then we create a new *dnode* for the (future) processing of the PBT or the child.

Creation of *dnode***.** Let d' be the new *dnode* to be created.

(Case 1:) s_c is on the primary path. Then, if d.*flag* is `T` and s_c has no PBT, d'.*flag* is set to `T`; otherwise, d'.*flag* is set to `F`. However, d'.*blist* is always initialized to be $\emptyset$.

(Case 2:) s_c is under a PBT. Then, if s_c has no child, d'.*flag* is set to `T`; otherwise, d'.*flag* is set to `F`. However, d'.*blist* is always set to ρ.

We then assign the depth of x to d'.*depth*. If s_c has a PBT, we also construct a PBT for d' with correspondence to the PBT of s_c. Finally, we connect d' to its parent and insert it as the head of s_c.*dlist* to provide direct access to d'.

Processing of text(). If x is a T event, then d is pointed at by a pointer in s_c.*dlist*. We have the following two cases:

(Case 1:) d is on the primary path. Then if the child of s_c is the output node and d has no PBT, we buffer or output x; otherwise, x is not a query result and hence we quit.

(Case 2:) d is under a PBT. Then if d has a PBT and the PBT is evaluated to be true with x, we delete d's PBT and invoke BubbleUp (d, $\emptyset$) to bubble up d's satisfaction to its ancestors; otherwise, we quit.

Processing of End-Tag. If x is E, then d is pointed at by a pointer in s_c.*dlist*. We delete d and remove its pointer from s_c.*dlist*. If d is on the primary path, we also upload or remove d.*blist*.

4.2. Buffer Handling

Buffer handling in QstreamX includes: (a) buffering or outputting; (b) flushing; and (c) uploading or removing.

Since streaming XML data can be recursive with respect to a query, we must avoid a query result being duplicately processed. We define our buffer data structure by *Buffer* = (*store*, *counter*). The use of Buffer to handle duplicate avoidance is discussed as follows.

(a) Buffering or Outputting. Given a potential query result, r, with the context *dnode*, d, we buffer r if d.*flag* is `F` or output r if d.*flag* is `T`.

To buffer r, we assign a Buffer, b, and store r in b.*store*. We keep b's pointer in a *Register* until r expires, i.e. before the next SAX-event is parsed. In this way, even if r needs to be buffered for many *dnode*s, we can simply insert b's pointer into the *blist* of these *dnode*s, and increment b.*counter* for each such *dnode*.

If we output r, we set a flag, which is unset when r expires, to prevent r being buffered or outputted subsequently. If r has already been buffered (i.e. the Register keeps b's pointer), we delete b.*store* and set it to be "`flushed`".

(b) Flushing. Given a *dnode*, d, and a list of Buffers, $\{b_1, \ldots, b_i, \ldots, b_n\}$, whose pointers are kept in d.*blist*. As soon as d.*flag* is set to `T`, we flush all b_i, i.e. b_i.*store* is outputted and deleted. The flushing also decrements b_i.*counter*. To avoid duplicate flushing, we set b_i.*store* to be "`flushed`", so that subsequent flushing will only decrement b_i.*counter*. We delete b_i if b_i.*counter* becomes zero. We also remove b_i's pointer from d.*blist* and when all Buffer pointers are removed, we set d.*blist* to $\emptyset$.

Due to the presence of multiple predicates, we need to maintain the consistency on the *flag* of *dnode*s on the primary path to ensure immediate flushing: on the satisfaction

of the predicate of a *dnode*, d, on the primary path, if the *flag* of d's parent is T, we *trickle down* from d to its descendants on the primary path, and for each *dnode* visited, we update its *flag* to T and perform flushing, until we stop at the first descendant whose predicate has not been satisfied.

(c) Uploading or Removing. Upon the deletion of a *dnode*, d, if d.*blist* is a list of Buffer pointers and d has no PBT, we upload d.*blist* to d's parent, that is, concatenating d.*blist* to the *blist* of its parent; but if d has a PBT, we access the Buffers via their pointers in d.*blist* to decrement their *counter*. A Buffer is deleted if its *counter* becomes zero.

4.3. Predicate Evaluation

We now discuss the actions performed upon the satisfaction of a *dnode*, d, under a PBT. When d's PBT is evaluated to be true, we first delete d's PBT, as to prevent the PBT being reprocessed and to release the memory. Then we invoke the *BubbleUp* (d, $\emptyset$) procedure to "bubble up" from d, as described in Procedure 4.1.

The basic idea of the *BubbleUp* procedure is as follows: on the satisfaction of a *dnode*, we bubble up its satisfaction to its parent; if the parent is thus satisfied, the bubble-up continues and may trigger the satisfaction of the whole PBT, and then in turn trigger the satisfaction of the PBT in the outer nest and so on recursively. If the satisfaction is bubbled up to the primary path, we apply trickle-down as discussed in Section 4.2(b).

Procedure 4.1 *BubbleUp* (*dnode* d, *dpnode* p)
begin
1. **if** ($p = \emptyset$) /* The bubble-up is from the *dnode*, d */
2. **if** (d.*flag* = T)
3. **if** (d's parent, d', is a *dnode*)
4. Delete all d''s descendants;
5. d'.*flag* := T;
6. **if** (d' has no PBT)
7. BubbleUp (d', $\emptyset$); /* To bubble up from d' */
8. **else** /* d' has not been satisfied */
9. Terminate BubbleUp;
10. **else** (d's parent is a *dpnode*, p')
11. Delete all descendants of p'; /* Since p' is satisfied */
12. BubbleUp(d's indirect-parent,p'); /* To bubble up from p'*/
13. **else** /* Not all d's descendants are satisfied */
14. Terminate BubbleUp;
15. **else** /* $p \neq \emptyset$, the bubble-up is from the *dpnode*, p, in d's PBT */
16. **if** (p is the root of d's PBT)
17. Delete p; /* Since p is satisfied */
18. BubbleUp (d, $\emptyset$); /* To bubble up from d */
19. **else if** (p's parent is of type A)
20. Set p.*type* to L (or R), if p is the left (or right) child;
21. **else** /* The type of p's parent must be O or L or R
Hence, p's parent is also satisfied */
22. Delete p;
23. BubbleUp (d, p's parent); /* To bubble up from p's parent */
end

4.4. A Detailed Example of Query Processing

Consider evaluating 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 (Figure 4) and d_i to denote a *dnode* in the DQTs (Figures 6(a)-6(f)).

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.

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,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) (ignore d_3-d_6 for the time being.).

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,LR), in b's *slist*, when we use dp to process s_3, we find that s_3 belongs to

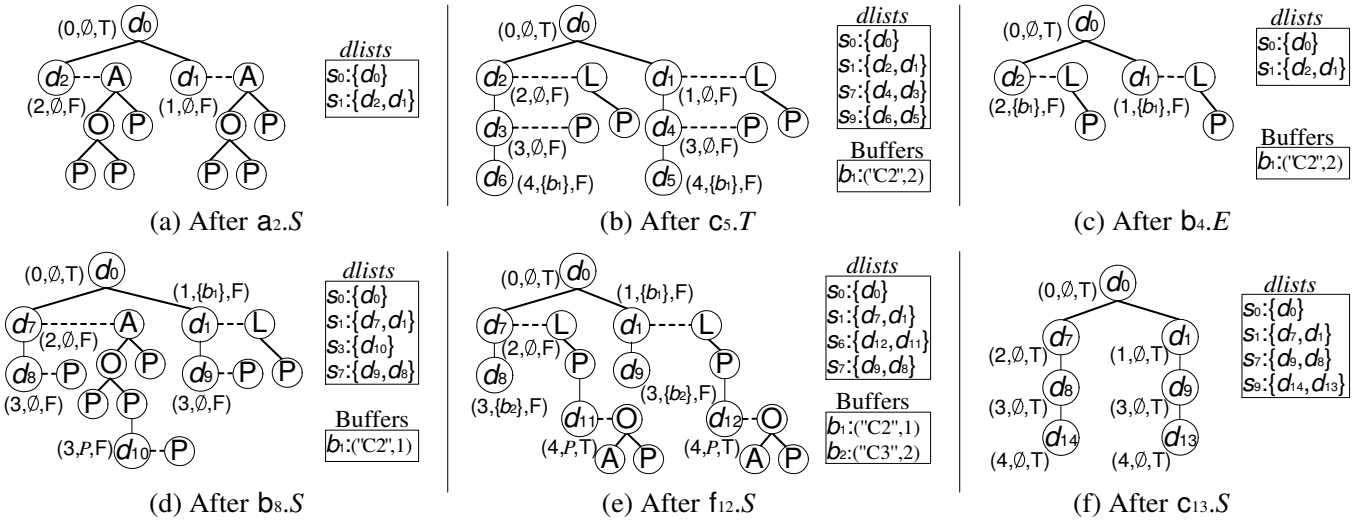


Figure 6. The DQTs for Processing the Query in Figure 4 on the XML Document in Figure 1

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, $\texttt{c}_5$.

Buffering. We only need to process the *slist*-element, $(s_7,s_9,\emptyset)$, for $\texttt{c}_5$. For $\texttt{c}_5.S$, 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 $\texttt{c}_5.T$, 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, $\texttt{c}_5.T$ is a potential query result. We create Buffer b_1 to buffer $\texttt{c}_5.T$, 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).

Uploading. To process $\texttt{c}_5.E$, 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*.

Then with $\texttt{d}_6.S$ and d's *slist*, $\{(s_7,s_8,\emptyset)\}$, we delete the PBT of d_4 and d_3, since $\texttt{d}_6.S$ satisfies s_8. Again, s_8's empty *dlist* avoids the redundant processing of $\texttt{d}_6.T$ and $\texttt{d}_6.E$.

To process $\texttt{b}_4.E$, 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.

Buffer Removing. Then for $\texttt{a}_2.E$, 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 $\texttt{a}_2.E$.

We then create another child, d_7, for d_0 with $\texttt{a}_7.S$. Then corresponding to s_7, $\texttt{b}_8.S$ creates d_8 and d_9 as child of d_7 and d_1 respectively. Although $\texttt{b}_8.S$ is not processed for d_1's PBT, we create d_{10} to evaluate s_3, as shown in Figure 6(d).

Then $\texttt{e}_9.S$ satisfies s_4 and we delete d_{10}'s PBT, while s_4's empty *dlist* avoids $\texttt{e}_9.T$ and $\texttt{e}_9.E$ being redundantly processed. Next $\texttt{c}_{10}.S$ creates a child for d_9 and d_8 respectively, corresponding to s_9. This $\texttt{c}_{10}.S$ 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 $\texttt{c}_{10}.T$, i.e. "C3", we buffer "C3" in Buffer b_2. On the arrival of $\texttt{c}_{10}.E$, the *blist*s are uploaded to d_9 and d_8. Then $\texttt{d}_{11}.S$ satisfies s_8 and we delete the PBT of both d_9 and d_8. Next, $\texttt{f}_{12}.S$ creates d_{11} and d_{12} to evaluate s_6, as shown in the updated DQT in Figure 6(e).

Predicate Processing (Trickle-Down) and Flushing. Then $\texttt{f}_{12}.T$, 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.

Outputting. Then for $c_{13}.S$ 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 $c_{13}.T$ for d_{14}, we immediately output $c_{13}.T$ as a query result. We also set a flag to indicate that $c_{13}.T$ is outputted, so that we do not output it again when we process d_{13} next. The flag is then unset.

Then for $c_{13}.E$, we delete d_{14} and d_{13}; for $b_8.E$, we delete d_9 and d_8; for $a_7.E$, we delete d_7.

Depth Mismatch and Hash-Lookup Filtering. Although s_7 is satisfied again with b_{14} and d_{15}, c_{16} does not match the *depth* of the child of s_7 and is hence filtered out. The elements, x_{17}, y_{18} and z_{19}, have no corresponding hash slots and are hence filtered out. Finally, we delete d_1 when $a_1.E$ comes, while we delete d_0, i.e. the root of the DQT, to terminate the query processing at the end of the stream.

4.5. Processing of Aggregations and Wildcards

In the previous discussion, we only present the handling of text() operation. If the output expression is an aggregation, we use a *statistics-accumulator* to evaluate count() and a *value-accumulator* to evaluate sum(), max() and min(), while avg() is simply the quotient of the contents of the value-accumulator and the statistics-accumulator. Note that count() is processed with the S or A events only, while the other aggregations also require the T event.

If the output expression is not specified, we use a *wildcard* output node that outputs/buffers all elements when a *dnode*, that corresponds to the last *snode* on the primary path, is created. However, if there is a wildcard in the query, a *snode*, that models the wildcard, is used to process every element when the parent of the *snode* is matched. A *wildcard-flag* is also used to activate the processing of the wildcard *snode*, since no element will be hashed into the slot where the wildcard's *slist* is stored.

4.6. Multiple Queries and Multiple Outputs

The SQT allows easy elimination of the redundancies introduced by the common prefixes in multiple queries. The basic idea is to first construct the SQT for the first query. This SQT is regarded as the base SQT, which is then extended to transform the second query and so on. Any common prefix that has already been built for the previous queries is skipped in the extended SQT. An example of the SQT for multiple queries is shown in Figure 7.

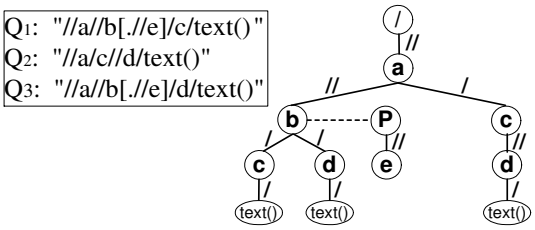


Figure 7. The SQT of Three Queries

Multiple queries are processed in a similar way as a single query, except that an extra query identifier is used to distinguish the query results for buffer handling. However, unlike the processing of a single query, we may output/flush a query result more than once, since the result may also match other queries.

The processing of multiple outputs is simply handled by multiple output nodes, which are processed in the same way as a single output node. To write a query to express multiple outputs, we use the XPath 2.0 union expression [5]. For example, for the SQT in Figure 7, the query is "//a(//b[.//e]/(c/text() | d/text()) | /c//d/text())".

5. Complexity Analysis

In this section, we analyze the complexity of QstreamX's query evaluation mechanism.

SQT Construction. Both the time and space complexity of the construction of the SQT, including the Hashtable, is linear in the size of the input query, since we need only one scan of the query to build both the SQT and the Hashtable.

DQT Construction. The complexity of the query processing is essentially that of the DQT construction, since the query processing is simulated by the dynamic construction of the DQT.

Let D be the tree model of the streaming XML document, $|Q|$ be the set of all *snode*s and Q' be the set of non-leaf *snode*s (note that a leaf *snode* does not create a corresponding *dnode*). For simplicity, we ignore *spnode*s (and hence *dpnode*s) in the analysis, but the total number of *spnode*s are always less than twice the number of *snode*s.

A *matching sequence*, $q \in D$, of a *snode*, s, is a sequence of nodes on the same path in D, such that q matches the path from the root of the SQT to s.

We define the *Average Degree of Recursiveness* of D with respect to a *snode* s, denoted by *ADR*(s), as the total number of matching sequences of s in D divided by the total number of paths that contain the matching sequences of s. For example, consider s_9 in Figure 4 and the document in Figure 1, there are six matching sequences, $\langle a_1, b_4, c_5\rangle$, $\langle a_2, b_4, c_5\rangle$, $\langle a_1, b_8, c_{10}\rangle$, $\langle a_7, b_8, c_{10}\rangle$, $\langle a_1, b_8, c_{13}\rangle$ and $\langle a_7, b_8, c_{13}\rangle$, on three paths, hence *ADR*(s_9) is $6/3 = 2$.

The average size of the DQT, i.e. the average of the size of the DQTs throughout query processing, is $\frac{1}{2}\sum_{s\in Q'}\lceil ADR(s)\rceil$ in the worst-case, since $\lceil ADR(s)\rceil$ indicates the maximum number of *dnode*s created corresponding to s. Intuitively, $\frac{1}{2}\sum_{s\in Q'}\lceil ADR(s)\rceil = \alpha|Q'|$, where α is a small constant in practice. However, if data is not recursive, $\alpha = (\frac{1}{2}\sum_{s\in Q'}\lceil ADR(s)\rceil)/|Q'| = 1$. In the average-case, not all $s \in Q'$ create corresponding *dnode*s. For example, no *dnode* is created for *snode*s under a PBT if the PBT is satisfied. It can also be observed from the example in Section 4.4, that the DQT is as small as $|Q'|$. We ignore the space used for buffering potential query results, since it depends on the query selectivity and is inevitable for any stream querying algorithm.

It takes $O(1)$ time to filter out the irrelevant elements. The worst-case time complexity for processing a relevant SAX-event, x, is given by $\mathcal{A} = \sum_{s\in Q} ADR(s)$, where s matches x, since $ADR(s)$ represents the number of *dnode*s to be created/processed. If data is not recursive and all elements in a query are distinct, $\mathcal{A}$ is 1. In practice, since both the number of s matching x (i.e. the number of duplicate elements in a query) and $ADR(s)$ are small, $\mathcal{A}$ is a small constant and the total query time varies as $\mathcal{A}|D|$, and thus the complexity is $O(|D|)$. In the average-case, the time is further reduced, since a large portion of the matching *snode*s are skipped due to satisfied predicates or depth mismatch. Finally, we note that the complexity of most operations in query processing are negligible, while that of the relatively more expensive operations, such as bubble-up and trickle-down, are also constant due to the extremely small size of the DQT.

6. Experimental Evaluation

We evaluate QstreamX on two main 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 (version 1.0) [19] and the XAOS system[1] [3]. We do not compare with the filtering systems [1, 9, 7, 2, 10, 18, 11] due to different inputs (a large number of filters *vs* a small set of queries), outputs (identifiers *vs* textual results) and evaluation methods (*no buffering* required for filtering). We use four real datasets [15]: the Shakespeare play collection (Shake), NASA ADC XML repository (NASA), DBLP, and the Protein Sequence Database (PSD), whose characteristics are shown in Figure 8. We also extensively study the factors that affect the performance of QstreamX using large sets of synthetic datasets and queries. We ran all the experiments on a Windows XP machine with a Pentium 4, 2.53 GHz processor and 1 GB main memory.

Dataset	XML Size	Text Size	Max/Avg Depth	No. of Elems.	No. of Attrs.	Parse Time (C++)	Parse Time (Xerces 1.0)
Shake	7.3MB	4.5MB	7 / 5.71	179K	0K	0.23 sec	0.54 sec
NASA	23.8MB	12MB	8 / 5.58	477K	56K	0.92 sec	1.64 sec
DBLP	127MB	68MB	6 / 2.90	3332K	404K	5.96 sec	11.73 sec
PSD	683MB	278MB	7 / 5.15	21306K	1291K	38.27 sec	72.10 sec

Figure 8. Characteristics and DTD of datasets

6.1. 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[2] 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 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 [20] used in XSQ is on average two times slower than the C++ parser used in QstreamX and XAOS, as shown in the last two columns in Figure 8. Therefore, we use the *relative throughput* [19], 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 9, 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. The remarkably high throughput verifies the $O(|D|)$ total query time obtained in the complexity analysis in Section 5. 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

[1] A released version of most published querying systems, except XSQ, is not available for comparison. We implement the XAOS system based on the algorithm presented in [3] except that we ignore the reverse axes.

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

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.

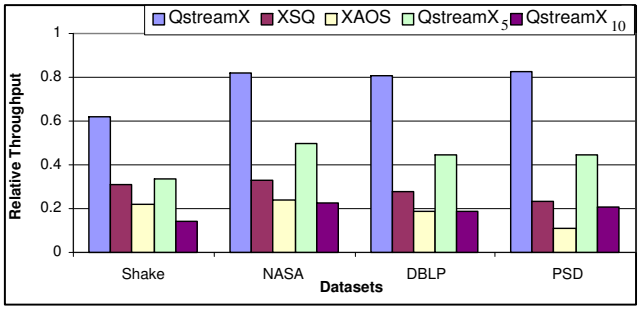


Figure 9. Relative Throughput

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 9. 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).

6.2. 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 the result of the space complexity analysis in Section 5 that the size of the DQT is extremely small (i.e. $O(|Q'|)$). 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.

6.3. Factors Degrading QstreamX's Performance

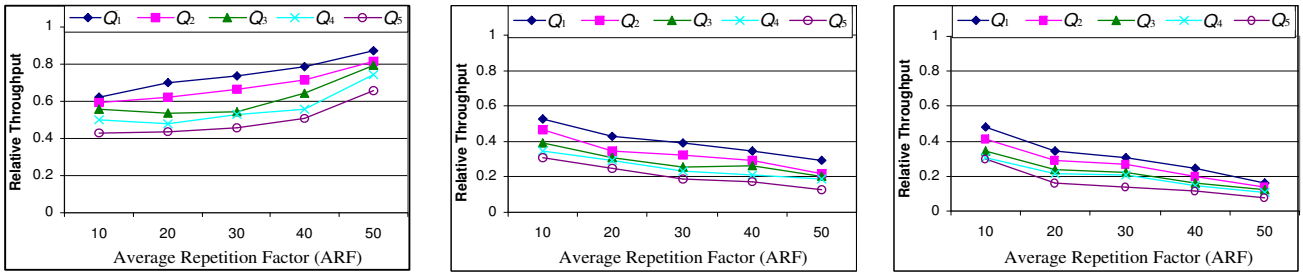
In Section 5, we show that both the average processing time for each SAX-event and the average size of the DQT, which determine the throughput and the memory consumption respectively, mainly depend on the Average Degree of Recursiveness (ADR) of the streaming XML data with respect to the elements of the input query. We now study the effect of the ADR on QstreamX's performance.

We generate three groups of synthetic queries consisting of the elements, `a`, `b` and `c`, and their respective `id` attribute, and the wildcard ($*$). For each group, 20% of the queries do not specify an output expression, while 40% are `text()` and 40% aggregations. Each group of queries contain an average of 0, 5 and 10 closure axes (i.e. `descendant(-or-self)` and `ancestor(-or-self)`) respectively. Each group is further divided into five sets, Q_1 to Q_5, of ten queries; and each Q_i have an average of 0, 3.11, 8.2, 14.34 and 19.67 atomic predicates per query respectively. The atomic predicates may be nested in (up to 10 nests) or connected by `and`/`or` operators with other predicates. Moreover, the atomic predicates in each Q_i have a roughly equal number of structural matches, exact-matches, range-matches and string-matches.

We then generate five synthetic datasets containing the elements, `a`, `b`, `c`, `x` and `y`, and their optional `id` attribute. Since it is hard to generate some specific ADR of the datasets with respect to a large group of queries, we simply repeat each distinct element at random positions on the same path many times. The average number, called the *Average Repetition Factor* (*ARF*), of each of the five elements on each path is set to 10, 20, 30, 40 and 50 for each of the five datasets respectively. Since the datasets contain only five distinct elements, the ARF roughly reflects the ADR. Note that the maximum and average depth of the dataset are 10 and 5 times of the ARF respectively.

Figures 10(a)-(c) show the average relative throughput of QstreamX on evaluating the five sets of queries, Q_1 to Q_5, of the three query groups, on the five datasets. In all the cases, the throughput decreases when the number of predicates in the queries increases (Q_1 has no predicate while Q_5 has the greatest number of predicates per query). In Figure 10(a), the throughput increases with an increase in the ARF of the dataset. This is because this group of queries has no closure axis and thus the streaming data of depth greater than the depth of the queries are filtered out, since a greater ARF implies a greater depth. For the other two groups, as shown in Figures 10(b) and 10(c), the throughput drops steadily with the increase in the ARF. On average, the throughput for an ARF of 50 is 59% lower than that for an ARF of 10, while the overall throughput of evaluating queries of 5 closure axes is 23% lower than that of 10 closure axes. However, the drop rates are acceptable, since increasing the ARF from 10 to 50 means increasing the maximum depth from 100 to 500 and the average depth from 50 to 250, and doubling the number of the closure axes can double the operations performed for query processing.

The memory consumption of QstreamX is roughly con-



(a) Queries with 0 Closure Axes (b) Queries with 5 Closure Axes (c) Queries with 10 Closure Axes

Figure 10. Relative Throughput of QstreamX on Synthetic Datasets and Queries

stant even with the complex datasets and queries. In general, higher throughput consumes less memory and vice versa (details thus omitted due to the similar trend). However, for all the datasets and queries, the maximum memory consumption is less than 3 MB.

To conclude, although an increase in the ADR, as estimated by an increase in the number of closure axes and the ARF of a dataset, degrades the performance of QstreamX, the degradation is gradual and acceptable considering that both the datasets and the queries are extremely complex. The experimental results, however, prove that QstreamX's performance is extremely competitive in practice.

7. Conclusions

We have presented QstreamX, an efficient system for processing XPath queries of streaming XML data, by utilizing a novel data structure, Hash-Lookup Query Trees, which consists of a simple hash table (the Hashtable) 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, with experimental evidence, that QstreamX achieves significantly higher throughput and consumes substantially lower memory than XSQ and XAOS. We have also presented a detailed empirical study of the factors that affect the performance of QstreamX. Our result indicates that even in the extreme cases, the system is able to maintain an acceptable performance. For future work we are going to explore more the common subexpressions in multiple queries and extend our algorithms to evaluate the sideways axes.

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] A. Berglund and et al. XML Path Language (XPath) 2.0, 2003. `http://www.w3.org/TR/xpath20`.

[6] S. Boag and et al. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, Nov. 2002.

[7] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of ICDE*, 2002.

[8] J. Clark and S. DeRose. XML Path Language (XPath) 1.0, 1999. `http://www.w3.org/TR/xpath`.

[9] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *ICDE*, 2002.

[10] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of ICDT*, 2003.

[11] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, 2003.

[12] V. Josifovski, M. F. Fontoura, and A. Barta. Querying XML Steams. To appear in *VLDB Journal*, 2004.

[13] 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.

[14] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, 2002.

[15] G. Miklau and D. Suciu. XML Data Repository. `http://www.cs.washington.edu/research/xmldatasets`.

[16] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *Proceedings of ICDE*, 2003.

[17] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XMLDM at EDBT*, 2002.

[18] M. Onizuka. Light-weight XPath Processing of XML Stream with Deterministic Automata. In *Proceedings of CIKM*, 2003.

[19] F. Peng and S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of SIGMOD*, 2003.

[20] Xerces Java Parser. `http://xml.apache.org/xerces-j/`.