Skipping Streams with XHints

Akhil Gupta Sudarshan S. Chawathe
Department of Computer Science, University of Maryland, College Park, MD 20742
{akhilg,chaw}@cs.umd.edu

Abstract

When streaming semi-structured data is processed by a well-designed query processor, parsing constitutes a significant portion of the running time. Further improvements in performance therefore require some method to overcome the high cost of parsing. We have designed a general-purpose mechanism by which a producer of streaming data may augment the data stream with *hints* that permit a downstream processor to skip parsing parts of the stream. Inserting such hints requires additional processing by the producer of data; however, the resulting stream is more valuable to consumers (since they have to perform less processing) , making such processing worthwhile. We present a set of hint schemes and describe how they are used by query engines. We demonstrate the benefits of our approach using an experimental study based on a hints-aware XPath query engine. Our results show that XHints can improve the performance of XPath query engines by as much as 100%.

1 Introduction

Streaming semi-structured data processing has recently gained immense importance, particularly in the area of publishing and subscription services. In most of these applications, the data is generated and sent by a server to a large number of subscribed clients in form of a stream. The clients may be interested in different portions of the data which can be represented in form of a query (e.g. XPath expression) and has to be evaluated on the data stream to obtain the desired portions of data.

A simple architecture for such an application is a centralized system where the clients submit their queries to a central data server. The server performs the necessary query evaluation and sends the appropriate portions of the data to each client. Although this scheme has a low overhead in terms of amount of data sent across network, it requires a large overlay of resources at the server side and is not scalable.

An alternative method is to send the data stream to each client using either unicast or multicast network methods and leave it to each client to pick the data it needs. The advantage of this approach is its simplicity, low processing cost at the server and scalability to a large number of clients. However, it suffers from the disadvantage of requiring each client to perform potentially large amount of redundant work. This problem is exacerbated by the presence of low-power clients such as PDAs and Web-enabled phones and requires a mechanism to reduce the computational load on the client query processors.

It has been observed that even well-implemented stream processors [2, 20] spend a large fraction (typically well over 50%) of their CPU resources on parsing the input stream. As a result, any post-processing optimization technique can only provide limited gain in system performance. Further improvement can only be attained by reducing the cost of parsing the data. One possible way of achieving it is by restricting the parsing to those portions of data stream that contain the query result. We refer to such data as *relevant* data with respect to the query. Clearly, skipping irrelevant data does not result in any loss in terms of the correctness of the output, but does help reduce the workload of the query processor.

For example, consider the XPath query `/book[discount]/title` on the sample XML data shown in Figure 1. For this particular query, a `book` element without a `discount` child element does not contain the query result and is irrelevant. Thus, such elements, e.g. the second `book` element (lines 21–30), can be skipped by a query engine. Similarly, within the `book` element, elements other than the `title` element do not affect the query result and can be skipped by the query engine.

Skipping data requires some form of direct access to relevant data. Traditionally, this access is usually provided with the help of indexes. Unfortunately, methods for indexing semi-structured data (e.g., [6, 7, 10, 19]) are not easily adapted to a stream-

```
1.<root>
2. <mag>
3.  <name>Times</name>
4. </mag>
5. <book>
6.  <title>
7.   Modern Information Retrieval
8.  </title>
9.  <discount> 10 </discount>
10. <price> 15 </price>
11. <year> 1972 </year>
12. <edition> 3 </edition>
13. <pub>
14.  <name>Addison Wesley</name>
15.  <address>
16.   34 Broadway, N.Y. U.S.A
17.  </address>
18. </pub>
19. <author> Ricardo Baeza-Yates </author>
20.</book>
21.<book>
22. <title>
23.  Database Systems:The Complete Book
24. </title>
25. <price> 60 </price>
26. <edition> 2 </edition>
27. <author> Hector Garcia-Molina </author>
28. <author> Jeffrey D. Ullman </author>
29. <author> Jennifer Widom </author>
30.</book>
31</root>
```

Figure 1: Example XML data

ing environment in which indexing information is required before the entire stream (which may be unbounded) is available. Further, unless the stream data is available a priori, indexes for streaming applications must be generated on the fly.

An early example of an index for streaming data is the *stream index (SIX)* for XML [11]. This index augments a stream with pointers to the beginning and end of each element. These pointers are stored in a compact binary form and may be used by a query processor to skip to the end of elements that are deemed irrelevant. SIX has a very low overhead and can significantly increase the throughput of streaming XML query processors for simple queries. However, SIX is of only limited use for more complex queries, such as those containing closures and predicates. The SIX index provides offsets to XML elements that matches a query label. However, the index does not contain sufficient information to infer if the element satisfies a predicate or contains a descendant specified in the query. As a result, the query processor may have to parse additional data resulting in an overhead.

For example, even in the case of moderately complex query such as `//book[price<10]//address`, the query processor may use the SIX index to jump to the first `book` element at line 5 in Figure 1. But as the index does not contain any additional information about the data, it has to parse the entire element even though it does not satisfy the predicate.

Furthermore, SIX has been designed for XML data that is available off-line for pre-processing. The index generation requires both the start and the end offsets of all elements, but in case of streaming data, some XML elements may not be completely available to compute these offsets.

We propose a flexible scheme for placing indexing information in a stream, in the form of annotations or *hints*. These hints, called **XHints**, store information about the stream data such as the offsets to data elements and summaries of data values. They can be inserted in the stream as special elements in the same format as the data.

XHints are generated at the server end and sent along with the data stream to the clients. The generation of XHints does not require access to the entire data stream. They may be generated using only partially buffered streams, allowing near-real-time processing of the data.

Since XHints constitute additional data that must be sent to clients, they do not result in any savings in network transmission costs in a unicast network. In a multicast network, savings may result from the fact that clients that would otherwise receive distinct streams now receive the same one. Further, XHints also imply some additional computation at the server (albeit simple, as described later). However, these additional costs at the server may be worthwhile because not only do they improve the efficiency of the system as a whole (server and many clients), they also increase the value of the data provided by the server to a client (because it is easier for the client to use it).

Our methods are applicable to any data stream representing tree-structured data that is serialized in pre-order. However, for concreteness, we focus on processing XPath queries on XML in this paper. Since our methods are based on the idea of skipping input data without parsing, a downstream query engine (at the client site) may not be able to check the well-formedness of the portions of stream that are skipped. However, we assume a trust relationship

between the producer and the consumer of the data in this regard because in any case, the consumer has to rely on the correctness and validity of the XHints inserted by the producer. The system also does not depend on the character encoding used in the data stream since the offsets to the elements are defined in terms of number of characters instead of raw bytes.

We may summarize the main contributions of this paper as follows:

1. To the best of our knowledge, the methods we present here are the first that permit a stream processor to systematically and flexibly skip parsing parts of semistructured data streams. Since parsing frequently accounts a large fraction of processing time, methods such as these are important if throughput is to be improved beyond the maximum throughput of a parser.

2. We describe a generic framework for XHints that allows any query engine to process streaming semistructured data more efficiently. We describe the application of XHints for an automated XPath query processor and an iterator based query engine.

3. We present an experimental study of our methods, quantifying the costs and benefits of inserting hints in data streams.

The rest of the paper is organized as follows. Section 2 describes the architecture of the XHint system and the API provided to the query engine. A detailed description of XHints is presented in section 3. The processing and generation of XHints are described in section 4 and 5 respectively. The application of XHints on two query engines is presented in section 6. Section 7 presents the performance evaluation of XHints. The related work is described in section 8. Finally, the conclusion is presented in section 9.

2. System Architecture

Figure 2 depicts the system architecture of a XHint-enabled query processor. The system consists of three modules, 1) the XPath query processor, 2) an XHint processing unit called XHManager and 3) a SAX XML parser. The solid lines in the diagram represent the flow of control between the three modules and the flow of data is shown by dashed lines.

The parser generates SAX events for the input data, which are sent to the XHManager. The XHManager acts as a proxy between the query processor and the XML parser. It may handle the SAX event internally (if it is an XHint) or forward it to the query engine (if it is a data element) for processing. It is also responsible for computing the offsets to various elements in the stream which are used by the parser to skip data.

XHManager determines the portion of the data that has to be processed by the parser with the help of XHints and the query engine. The XHManager assumes that the query engine has a mechanism to identify the elements specified by the XPath query (including any conditions such as predicates) that have to processed. The query engine also has the responsibility of registering these events with the XHManager in a list referred as the `EventList`. The `EventList` acts as a medium through which the query engine provides a list of labels of elements that should not be skipped in order to maintain the correctness of the output. As described more in section 3, XHints provide information such as offsets to various elements to the XHManager. The XHManager uses this information to skip as much data as possible while guaranteeing that no SAX event corresponding to the elements in the `EventList` is skipped.

The list of essential events changes as the stream is processed and must be updated accordingly by the query engine. For example, in case of the query `/book/discount`, when the query processor is at the start of the stream of Figure 1, all SAX events corresponding to the `book` elements have to be processed. Thus, the query engine registers this event in the `EventList`. After parsing the start tag of the `book` element in line 5, the query engine should not skip any `discount` element. However, it can skip any other element including a `book` element. Thus, the `book` SAX event is replaced by the `discount` SAX event in the `EventList`. The `book` event replaces the `discount` event to the `EventList` when the end tag of the `book` element is processed at line 20.

The XHint-enabled query engine process only selective SAX events, called **essential** SAX events. An essential SAX event corresponds to an element about which the XHints do not provide sufficient information to infer if it does not contain any of the elements of the `EventList`. Thus, an essential event cannot be skipped by the query engine. All relevant portions of the data stream are essential events though the converse is not true. Ideally, we would like XHints to contain sufficient information to restrict essential SAX events to relevant portions of the stream but it is not possible due to large data overhead costs. As a result, the processor might also have to process irrelevant elements.

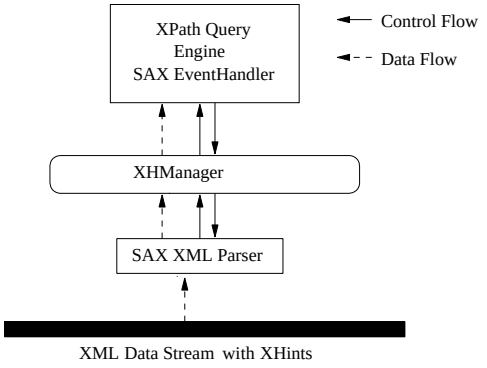


Figure 2: System Architecture

For example, a query engine with the query `/book/discount` on the data of Figure 1 must necessarily process every `book` element with a `discount` child element. Thus, the SAX event corresponding to the `book` element at line 5 in Figure 1 is an essential event for this query. On the other hand, the `book` element at line 21 is not an essential event if the XHints let the query engine infer that it does contain a `discount` element and can be skipped safely.

If the query contains a predicate, as in `/book[price<20]/pub//name`, an element contains query result if only if satisfies the predicate. If an XHint provides information that the element does not satisfy the predicate, it is not an essential element and can be skipped. In case of the example query, the `book` element is not essential if the XHint provides the information that it does not contain a `price` element with value less than 20. If the query expression has an element label following a closure axes, the query engine may restrict its attention to elements containing a descendant with that label. For example, any child element of the `book` element is an essential SAX event for the query `/book//name` unless XHints explicitly provide information that it does not contain a `name` element as its descendant.

The XHManager does not depend on the particulars of the query engine. It only requires that the query engine correctly update the EventList. The interface for this purpose is composed of the functions summarized below.

1. int **addSAXEvent**(String *uri*, String *localName*, String *PredicateElementLabel*, String *Operator*, String *ConstantValue*, int *AxisType*)

   This function registers a SAX event with the XHManager. It returns a unique identifier for the SAX event. The SAX event is identified by its URI and the tag label of the element *localName*. In addition to these identifiers, information about the event such as predicate and axes type is also provided.

2. void **removeSAXEvent**(int *EventID*)

   This function removes the event corresponding to the EventID from the EventList.

As mentioned before, in order to update the EventList using this API, the query engine should be capable of identifying the essential SAX events as it processes the data. We later show how it can be achieved with the help of some actual query engines.

3. XHints

XHints are special XML elements that are used to store a brief structural summary of a data stream. This summary is encoded using XML attributes. Although the attributes of an XHint can store a variety of information, in this paper we focus on only four kinds of attributes or hints. These four types of hints are *End Hint*, *Child Hint*, *Sibling Hint* and *Descendant Hint*. Figure 3 depicts a sample hints-enhanced data stream, with the XHints marked with $*$.

Each XHint stores information about a portion of the stream known as its **scope**. Although the scope of an XHint can be any arbitrary portion of the data, we restrict it to the data between the end of the XHint to the end of its parent element or the start of a sibling data element. This restriction ensures that an XHint does not contain information about any element outside its parent element. As a result, the system cannot obtain offsets to elements outside the parent element and skip directly to them. It guarantees that either the complete element is skipped or both its start and end tag are parsed to maintain the validity of the processed data. Further, the restriction results in nested scopes i.e. the scope of a XHint of a descendant lies completely within the scope of an XHint of the ancestor. As we shall see later in section 4, this property is crucial for efficient processing of XHints.

If the scope of the XHint ends at the start of a sibling data element, a sibling XHint is inserted after the end of the sibling element to store information about the remaining portion of the parent element. Note that none of these two XHints contain information about the sibling element present between them. Thus, it may have to be parsed even though it might not contain the query result. However, allowing the scope to end at an sibling element instead of the end of the parent element permits on-the-fly generation of XHints for incomplete elements in the data stream.

The XHints store some of the information (such as descendant information) encoded in order to reduce the data overhead. The meta-information about the encoding is provided to the client in a special XHint element called META. Like normal XHints, the META element stores the meta-information as XML attributes and is inserted at the start of its scope. However, unlike normal XHints, the scope is not restricted to the end of its parent element and can extend to an arbitrary length until the next META element. The restriction is not needed for this element since the information in the META element is not used to skip data. Its scope is usually determined by the resources available at the server end to generate the hints.

The end hint of a node contains the offset from the end of the XHint to the end of its scope. It is stored as the value of attribute `‘‘end.’’` This hint allows the parser to directly skip to the end if the node or any of its part does not contain an essential event. The value of the `end` attribute of the XHint at line 3 in Figure 3 gives the offset to the end of the `root` element.

The child hint stores the offsets to different child elements of a node. The offsets to child elements with label l are stored as the value of an attribute named l, in the form of a colon-separated list. The attribute `author` of the XHint in line 24 in Figure 3 is an example of a child hint. It stores the offsets and the data digest (described later) of the three `author` child elements in a colon-separated list. These offsets can be used by the query processor to jump directly to the child elements which may contain the query result.

Since there is no predetermined bound on the number of children of an element, the storage of such child offsets may cause an XHint to become very large. This situation is undesirable because a downstream processor needs to parse the entire hint before using any information (not necessarily the information related to the numerous children).

Sibling Hints are used to limit the size of an XHint. A sibling hint of a node contains offsets to sibling nodes with the same label and is stored as the value of attribute `‘‘sib.’’` The XHint of a parent node is used to store only the offsets to first c (a parameter) children with any label. The offsets to the next say n nodes are stored in the c^{th} child node. The $(c+n)^{th}$ node contains the offset to the next n nodes and so on. In this manner, sibling hints allow storing the offsets to a large number of children nodes without making any one particular XHint very large.

XHints can also be used to store information about the text contained in an element. We store a summary of the text node in form of a *data digest* along with the offset as part of the child and sibling hints. This additional information is useful for queries with predicates. A query engine can use the data digest to pre-evaluate predicates and skip elements that do not satisfy the predicates. If the text is an alphanumeric string, we store the first s characters of the string. (Typically, s is small, say 3.) If the constant specified in the predicate does not match the first s characters of the element, the query processor can skip the element since it definitely does not satisfy the predicate.

We use a different scheme to generate the data digest for text nodes with numeric values. The range of numerical constants occurring for each label in a portion of the data stream is stored in the appropriate META element in the `Hash` attribute. The attribute stores the range as a series of *(element label, minimum value, maximum-value)*. An example of the `Hash` attribute appears in line 2 in Figure 3. The range is then divided into a fixed number of equal-sized intervals and the interval index of the numeric text of an element is stored as its data digest.

The *Descendant Hint* provides information about the descendants of an element in a concise form using a bitmap. Each label occurring in the data is assigned a unique index. If a particular label occurs as a descendant of the node, the bit at the index corresponding to the label is set. The bitmap is stored as integer-valued attribute `desc` in an XHint element. The mapping from labels to indexes is stored as the value of the attribute `LIndex` of the META element, as a list of label-index pairs as suggested by line 2 of Figure 3.

The descendant bitmap is useful for processing queries with closure axes. For such queries, an element may contain the label specified after the descendant axis in the query as a child of any of the descendant of the element. The bitmap allows the XHManager to infer which elements do not contain the label as the descendant and avoid parsing its child elements.

4. XHint Processing

In this section, we describe how XHints are used for processing data more efficiently. In the following discussion, we assume that the query engine updates the list of essential events correctly for the XHManager and focus on the operation of the XHManager. As noted earlier, the parser generates SAX events based on the input stream. The XHManager handles such

```
1.<root>
2.* <META LIndex=''address 0 name 1 pub 2
   edition 3 discount 4 price 5 year 6
   title 7 author 8 mag 9 book 10''
   Hash=''price:15-60 discount:10-10''/>
3.* <Hint end=''768'' desc=''255'' mag=''2''
   book=''67''/>
4. <mag>
5.  <title> Times </times>
6. </mag>
7. <book>
8.*  <Hint end=''320'' desc=''3'' sib=''327''
    title=''2'' discount=''46-0''
    price''92-0'' edition=''129-thi''
    pub=''149'' author=''235-Ric''/>
9.  <title>
10.  Modern Information Retrieval
11. </title>
12. <discount> 10 </discount>
13. <price> 15 </price>
14. <edition> third </edition>
15. <pub>
16.  <name>Addison Wesley</name>
17.  <address>
18.   34 Broadway, N.Y. U.S.A
19.  </address>
20. </pub>
21. <author> Ricardo Baeza-Yates </author>
22.</book>
23.<book>
24.* <Hint end=''213'' desc=''0'' title=''2''
     price=''34-1'' edition=''96-sec''
    author=''123-Hec:165-Jef:198-Jen''>
25. <title>
26.  Database Systems:  The Complete Book
27. </title>
28. <price> 60 </price>
29. <edition>second </edition>
30. <author> Hector Garcia-Molina </author>
31. <author> Jeffrey D. Ullman </author>
32. <author> Jennifer Widom </author>
33.</book>
34</root>
```

Figure 3: XML data with XHints

events in two ways. If the event represents a data element, it is forwarded to the query engine. Otherwise, the event represents an XHint, and it is processed by the XHManager itself. Algorithm 1 provides the pseudo-code for the XHint processing algorithm.

If the XHint element is a `META` element, the XHManager uses the `processMETA` procedure to process the two attributes, `Lindex` and `Hash` to obtain the label-to-index mapping and the range of the numerical values of elements. Otherwise, the XHManager uses the EventList to process the XHint. The events registered in the EventList can be broadly classified into three types: (1) events corresponding to child elements without any predicates; (2) events corresponding to elements associated with predicates; and (3) events corresponding to descendant elements. The XHManager uses these three types of events to determine the offsets to the relevant portions of data.

XHManager uses a stack called **OffsetStack** to store and maintain offsets to the data elements. The offsets are stored in the same order the elements occur in the stream, with the offset to the element occurring first at the top of the stack. The nested scope of XHints helps maintain this order implicitly. When the parser reaches an XHint of an element, the values already present in the stack have been obtained from its ancestors and point to positions in the stream beyond the scope of the XHint. Thus, the offsets from the XHint refer to elements that occur before the elements pointed by the offsets in the stack. So, none of the offsets obtained from the XHint have to be inserted in the middle of the stack. Instead, they can be added to the stack in the increasing order of their offset value maintaining the proper order.

The initial offset values to various elements are relative from the end of the XHint they are obtained from. These have to be updated as the stream is processed. Each offset has to be reduced by the number of characters from the end of the XHint to the position in stream where the offset is used. A naive solution to perform this operation is to traverse the stack after every SAX event and reduce the number of characters skipped or read from each value. Obviously such an approach is highly inefficient and can severely degrade the throughput of the query processor.

We use a more efficient way of maintaining the correct offset values by observing that the parser guarantees that the parser reaches all the positions pointed by the offsets in the OffsetStack. So, the XHManager requires an offset only after the parser reaches the position pointed by the offset stored at the top of it in the OffsetStack. Thus, we only need the difference

between the two consecutive offsets in the OffsetStack instead of their absolute values.

We modify the `push` operator of the OffsetStack to perform this operation. Every time an offset is pushed into the OffsetStack, the previous head of the stack is reduced by its value. It ensures that an entry in the stack contains the offset relative to the position pointed by the value stored above it. For example, if the OffsetStack is {345} and we have to insert {213}, the modified `push` operation results in {132,213}. On inserting another value, {23}, the stack contains {132,190,23}.

However, the updated offsets values from the OffsetStack are not sufficient to jump to the desired positions in the stream. The modified `push` operator only ensures that the offset value is relative to the last position pointed by an XHint. It does not take into account the characters the parser might read after reaching that position. For example, in the example data in Figure 3, the end hint from line 8 gives the number of characters in line 9–21 as 320. The child hint for the `pub` element gives the offset from line 9 to line 15 as 149. If the XHManager insert these two offsets in the OffsetStack, the top of the stack is {...,171,149}. At the end of the XHint, the parser uses the first offset to jump to line 15. Since the `pub` element does not contain an XHint, the parser has to parse the entire element. At the end of the `pub` element, the offset stored at the top of OffsetStack is 171 which is the offset from line 15 to line 21 but the parser is currently at the end of line 20. Thus, additional information is required by the XHint processor to correctly estimate the offset to line 21 from current position.

This information is made available in the form of number of characters processed by the parser for each element. We only require this information for elements which have not been completely parsed yet. Since there can be at most one such element at each depth, we use a separate stack called the **CharStack** to store the number of characters in the decreasing order of the depth of the element they correspond to.

At the start of each element, the XHManager inserts a 0 at the top of CharStack, which is updated as its child elements are processed. After processing the last essential child element, the XHManager uses this value along with the offset stored at the top of OffsetStack to jump to the end of the element. In the example mentioned above, the offset from line 20 to the start of the end tag of `book` at line 21 can be obtained by subtracting the number of characters read inside the `book` element from the value at the top of the OffsetStack.

Algorithm 1 XHint Processing

```
procedure startElement(SAXEvent e)
 1: if e is an XHint then
 2:    processXHint(e);
 3: else
 4:    QueryEngine.startElement(e);
 5: end if

procedure endElement (SAXEvent e)
 1: if e is an XHint then
 2:    processXHint(e);
 3: else
 4:    QueryEngine.endElement(e);
 5: end if
 6: parser.skipData(OffsetStack.pop()-CharStack.pop());

procedure processXHint(SAXEvent e)
 1: if e is a META element then
 2:    processMETAElement(e);
 3:    return;
 4: end if
 5: OffsetStack.add(e.getEndHint());
 6: for all Events E in EventList do
 7:    if E is a child Event with label L then
 8:       if E does not have a predicate then
 9:          OffsetStack.add(e.getChildHint(L));
10:       else if E has an existential predicate with label L'
          then
11:          if XHint has a child hint for label L' then
12:             OffsetStack.add(e.getChildHint(L));
13:          end if
14:       else if E has a comparison predicate with label L'
          then
15:          if the data digest of label L' satisfies the predicate
             then
16:             OffsetStack.add(e.getChildHint(L));
17:          end if
18:       end if
19:    else if E is a descendant Event with label L then
20:       if The bit for L is set in the descendant bitmap then
21:          OffsetStack.add(e.getComplexChild());
22:       end if
23:    end if
24: end for
```

During the processing of an XHint, if an event in the EventList is of type 1, the relevant elements which need to be processed by the query engine are the child elements corresponding to the label. The offset to these elements can be obtained from the child hint of the XHint. These offsets allow the parser to jump directly to these child elements, skipping the rest. In addition to the offsets to the child elements, the XHint manager uses the end hint of the XHint to provide the offset from the last relevant child element to the end of current element.

Example 1 Consider the query `/book/title` on the stream of Figure 3. The result of the query consists of the `title` elements in lines 6–8 and 22–24 of the original XML stream (Figure 1).

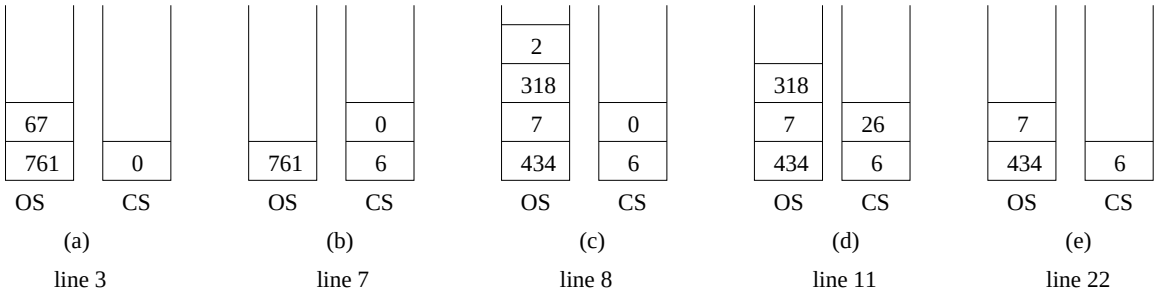


Figure 4: State of OffsetStack (OS) and CharStack(CS) at (a) line 3 (b) line 7 (c) line 8 (d) line 11 (e) line 22 for query `/book/title`

Initially, both the OffsetStack and CharStack are empty. At the beginning of the stream, the query engine registers the SAX event corresponding to a `book` child element with the XHManager. When the XHManager processes the XHint in line 3, the `book` label in the EventList indicates that it is the next essential element that should be processed by the parser. As a result, the offsets related to the `book` child from the XHint are stored in the OffsetStack. The XHManager also stores the offset to the end of the `book` for future use. At this point, the state of two stacks is shown in Figure 4(a). Note that the offset values have been modified by the special `push` operator before inserting.

At the end of the XHint, the XHManager pops the offset at the top of the stack and uses it to jump directly to the `book` element at line 7. The processing of the SAX event for the `book` element is delegated to the query engine. Since the essential element inside an `book` element is a `title` element, the query engine on processing the start tag of `book` element at line 7 replaces `book` from the event list of the XHManager with `title`. An additional entry is added for the `book` element to CharStack after updating the previous value. The state of the stacks is shown in Figure 4(b).

The next XML element to be parsed is the XHint at line 8, which is handled by the XHManager internally. As the event list now contains the `title` event, the manager uses the child hint for `title` to obtain the offset and store it in the OffsetStack along with the offset in the end hint and sibling hint. Figure 4(c) shows the state of the stacks.

At the end of the XHint, the parser uses the offset at the top of the stack to skip directly to line 9. After the query processor outputs the `title` element, the CharStack is updated (shown in Figure 4(d)) to include the number of characters in the element. The XHManager requests the parser to jump to the end of the `book` element at line 22 since there are no more essential SAX events (`title` elements). In this case, the offset at the top of the stack is 318. The offset to the end of the `book` tag from the end of `title` is calculated by subtracting the length of all children of the element processed by the parser (line 9–11) available from CharStack from this offset and jump straight to line 22. The state of the stacks is depicted in Figure 4.

When the query engine parses the end tag of `book` element, it again updates the XHManager's event list by removing the `title` and adding the `book` event to it. Next, it uses the value at the top of the stack to jump to next `book` element. The XHManager processes the second `book` element in a similar fashion using the EventList to skip all child elements of the `book` element except the `title` at line $25 - 26$.

This scheme allows the parser to process only 6 elements compared to 20 elements processed by a normal query engine saving the parsing cost involved. ■

Note that although XHints do not provide direct offsets to the result elements, they provide offset information for all children nodes instead of just one particular type and can be used to skip data for other similar queries like `/book/author` and `/book/discount` without requiring any additional processing of the data.

In case of more complex queries with predicates and queries, the basic algorithm remains the same. The values in the OffsetStack and the CharStack are maintained in exactly the same fashion as explained in Example 1. The only difference arises in the logic the XHManager uses to decide which offsets are relevant and should be inserted in the OffsetStack. In the following discussion, we assume that the offsets obtained from the XHints are updated using the OffsetStack and CharStack and do not provide the exact

details on how is a particular offset stored by the system. Instead, we concentrate on how does XHManager use the descendant bitmap and data digest to identify the irrelevant portions in the stream.

In case an event is associated with predicates, the relevant elements can be identified only after evaluating the predicate. The XHint Manager uses the data digest to pre-evaluate the predicate to select the relevant offsets. If a particular element does not satisfy the predicate, the XHManager can avoid parsing it.

If the predicate is an existential predicate such as in `/book[discount]/title/text()`, the presence of a child hint with the label of the predicate is sufficient to pre-evaluate the predicate. An element can satisfy an existential predicate for an element with particular label l if and only if the XHint of the element contains a child hint with label l.

Example 2 Consider the query */book[discount]/title/text()* on the data in Figure 3. The first *book* element satisfies the predicate and its *title* element belongs to the result. However, the second *book* element does not satisfy the predicate and can be skipped by the query processor.

However, a normal query processor is not aware of this fact and will parse all 20 elements. But XHints provide information about all the child elements in form of child hints. This fact can be used by a query engine to pre-evaluate the existential predicate. If the XHint of a *book* element does not contain a child hint for a *discount* element, the parser can skip parsing the remaining element.

The query engine registers an essential event with the XHint Manager with the tag label *title* and an existential predicate with label *discount* on reaching the start of the first *book* element. When the parser reaches line 8 of the example data, the XHManager processes the child hints present in the XHint of the first *book* element. Since it contains the child hint for the SAX event in the predicate (*discount*), the XHManager infers that this element satisfies the predicate, and thus has to be parsed by the query engine. It uses the offsets from the child hint for the *title* element to skip parsing other elements.

On the other hand, on processing the XHint of the second *book* element at line 24, the absence of a child hint for the *discount* label indicates that this *book* element does not satisfy the predicate and thus, is skipped.

The query processor only parses 8 elements to process the entire data by using XHints saving more than 50% in terms of number of SAX events generated. ■

If the predicate involves a comparison operator, the XHManager uses the data digest and the range value stored in the *Hash* attribute to evaluate the predicate. The XHManager computes the data digest of the constant value in the predicate and compares it with the data digest from appropriate child hints to identify the elements that do not satisfy the predicate.

An element does not satisfy the comparison predicate if the data digest of the constant value does not match the data digest of the element. If no child element satisfies the comparison in the pre-evaluation, the element does not contain the query result and is skipped by the parser.

Example 3 Consider the query */book[author="R. Baeza-Yates"]/title/text()* on the example data in Figure 3. The query contains a predicate with a string comparison operator. If the query engine does not have prior information about the text of the *author* elements, it has to parse the entire *book* element in order to evaluate the predicate.

The XHManager helps avoid the overhead of parsing elements that do not satisfy the predicate by using the descendant digest present in the XHints. At the start of the second *book* element on line 23, the query engine registers the predicate with the XHManager. The XHint of the element contains the first three characters of the text in addition to the offsets to the three *author* elements. The XHManager uses this digest to evaluate the predicate *a priori*. In this case, since the descendant digest of none of the three elements does not match the first three character of the constant in the predicate, XHManager requests the parser to skip all the child elements and directly go to the end tag of *book* element at line 33. ■

Note that although a difference in the data digest guarantees that the element does not satisfy the predicate, a match does not necessarily mean that the element will satisfy the predicate. The processor still has to parse the element and the element may not satisfy the predicate. For example, if the constant in the predicate in the query in the example was "Jeff Ullman" instead of "R. Baeza–Yates," the descendant digest for the second *author* element at line 31 matches with the descendant digest of the constant although the predicate is not satisfied.

Ideally we would like to provide as much information possible in the data digest to minimize the chances of such false pre-evaluations but there is a trade-off involved between the benefit obtained from

avoiding false evaluations and the overhead of processing it. Although we have not conducted a thorough experimental study to obtain an optimal data digest scheme, preliminary results lead us to believe that the current scheme provides a good cost-to-benefit ratio.

If an essential event in EventList corresponds to a label (say l) associated with a closure axis, it can occur as a deeply nested descendant of the current element. The XHManager cannot skip any part of the current element unless it has the information that it does not contain l as its descendant. The XHManager uses the descendant hint of the XHint to determine it. If it does, the element can occur as a child of any of the complex child elements (elements with their own child elements). As a result, the XHManager stores the offset to all such child elements so that the query engine can process them.

Example 4 Consider the query *//address* on the data shown in Figure 3. The *address* label is mapped to index 0 by the LIndex attribute of the META element at line 2. Thus, if an element contains a descendant with label *address*, the 0^{th} bit of the bitmap in the descendant hint is set to 1.

The first bit in the descendant bitmap is set for the XHint of the *root* tag indicating that it contains at least one *address* element as its descendant. As a result, the query engine leaves all atomic child nodes (since they cannot have an *address* element as their child or descendant) and processes the complex child nodes (with non-text child nodes). In this case, all the three child elements of *root* are complex.

When the processor reaches the first *book* element at line 7, it again checks the descendant bitmap of the XHint at line 8. As the appropriate bit is set on indicating that this *book* element contains an *address* element, the XHManager skips to the complex child elements. In this case, the only complex child element is the *pub* at line 15 which is parsed to obtain the *address* element.

In case of the second *book* element at line 23, these descendant hint of the second *book* element has the value 0 indicating that it does not contain any descendant. As it also does not have a child hint for a *address* label, the query processor jumps directly to the end of the element at line 33.

The total number of elements parse by the query engine using XHints are 11 compared to 20 elements parsed by a normal query processor.

■

5. XHint Generation

As with traditional indexes, the benefit of an XHint varies based on the kind of XHint, the characteristics of the input stream (e.g., relative sizes of elements), and the intended use of the stream (query mix, access patterns, etc.). In general, we may use such information to guide the selection of XHints to be inserted. This task is analogous to the index selection task for traditional indexes. However, when such information is not available or, equivalently, when the expected uses of a stream vary widely, a reasonable policy is to insert hints that are likely to benefit a large class of applications. Again, this policy is analogous to the commonly used policy of building indexes on primary key attributes for traditional databases. In our current implementation, we use the following policy: We insert XHints for only elements that have at least two children. Intuitively, this policy is motivated by the observation that an XHint for an element fewer than two children does not reduce the number of SAX events generated by the parser.

The offsets stored in the XHint for an element are affected by the lengths of the XHints of that element's descendants. For example, the offset of an element's end tag is changed by every XHint inserted in the subtree rooted at that element. If the entire the data stream is available, a simple solution is to construct a DOM tree for the stream and generate XHints in a bottom-up manner. However, this approach is not suitable for streaming data when only a limited amount of the data is available (limited by, say, buffer capacity and timing constraints).

In such a streaming environment, we split the stream into chunks and generate the XHints one chunk at a time. When the hint generator encounters the end of an element, it uses the information about that element's descendant's (which have been encountered earlier) to generate its XHint. These XHints are buffered along with the element itself. When the buffer is filled, its contents (original data along with the inserted XHints) is output to downstream components. The META element containing meta-information about the XHints is added at the start of each data chunk. The information from the META element such as the label to bitmap index mapping is relevant only for processing of that particular data chunk and may change from one data chunk to the next.

Since the buffer size is fixed, the end tag of some elements may not be reached before the buffer is filled. Similarly, some of the elements in the beginning of the buffer may not have start tags. The XHints for

such elements contain only partial information about its contents and descendants.

The end hint of an element e whose end tag is not present in the buffer holds the offset to the start of the last child element encountered instead of the end (which has not yet been encountered). In turn, if the end tag of the last child element too has not been reached, the end hint of the child element points to its last child element. If e has no child elements and only contains text, the end hint stores the offset to the end of text. The other types of hints for e contain information about only the subtree rooted at e that was encountered before the buffer was filled.

Additional XHints are also inserted for elements whose start tags are not present in the current buffer. The position where the XHint is inserted depends on whether the element has an incomplete child element (whose start tag is also absent) or not. If it contains an incomplete child element, the XHint is added immediately after the end of it, otherwise the XHint is added at the top of the part of the element in the buffer.

XHints with partial information about the element result in multiple XHints for a single element. The end hint of a partial XHint provides the offset to one of the child elements instead of the end of the element. The scope of the each partial XHint is defined from the end of the XHint to the start of the child hint. In that case, the partial XHint does not provide information about the child element (since it is out of its scope) and as a result, has to be processed even though it might not contain query result. However, the XHint inserted at the end of the child element provides information about the remaining portion of the element.

6. Using XHints

In order to use XHints efficiently, the query engine must identify essential SAX events at each stage in query processing and update the EventList accordingly. We use two query engines to illustrate this mechanism. Our focus is on XSQ [20], an automaton-based streaming XPath query processor. We have implemented the mechanisms described here to generate a hints-aware version of XSQ, called XSQ-H. However, in order to better illustrate the mechanism for using Xhints, we also describe how XHints may be used by an XQuery engine that is based on the standard iterator model: Tukwila [14].

6.1 XHints and XSQ

XSQ evaluates an XPath query on streaming data by generating an automaton called a Hierarchical Push-down Transducer (HPDT) from a query. A HPDT is a collection of smaller finite state machines called Buffered Push-Down Transducers (BPDT) that are arranged in a hierarchical manner. Each BPDT has its own buffer, which is used to store potential query results. Figure 5 depicts the HPDT for the query `/book[price<20]//author`.

The transitions, represented by arcs, are associated with a SAX event and an action. The action manipulates the buffer associated with the BPDT by uploading its content to the parent BPDT (`UPLOAD`), add (`ENQUEUE`) data, clear (`FLUSH`) the content or output it as query result (`OUTPUT`). If the SAX event generated by the XML parser matches a SAX event defined on an arc from the current state, the HPDT makes the transition to the new state and executes the action defined with it.

Note that if a SAX event does not match any arc from the set of current states, the HPDT does not perform any transition or action and maintains the same configuration it was in before processing the event. In other words, the absence of such SAX event would not affect the query processing and thus, can be ignored safely by the XML parser.

This observation provides us with a simple mechanism to identify the essential events using the current states of HPDT. The essential events are those events that result in a transition in the HPDT and are easily identified by the SAX events defined of the arcs from the current state.

XSQ-H is an XHint compatible version of XSQ that performs the additional task of identifying the essential SAX events and updating the EventList of the XHManager.

Although the HPDT contains sufficient information to identify the essential events, the information may not be immediately available to XSQ-H from the current state. For example, in Figure 5, the arc from state {201} with the `price` does not contain the constant value of the predicate. This information is instead present on the arc from state {202}. However, this information is required to pre-evaluate the predicate and identify the essential `price` elements when the system is in state {201}.

In addition to the constant value, the system also requires information about the operator of the predicate. The HPDT contains two arcs for every predicate (e.g. arcs from the state {202} for the predicate `[price<20]` from Figure 5), one corresponding to the transition when the predicate is true and the

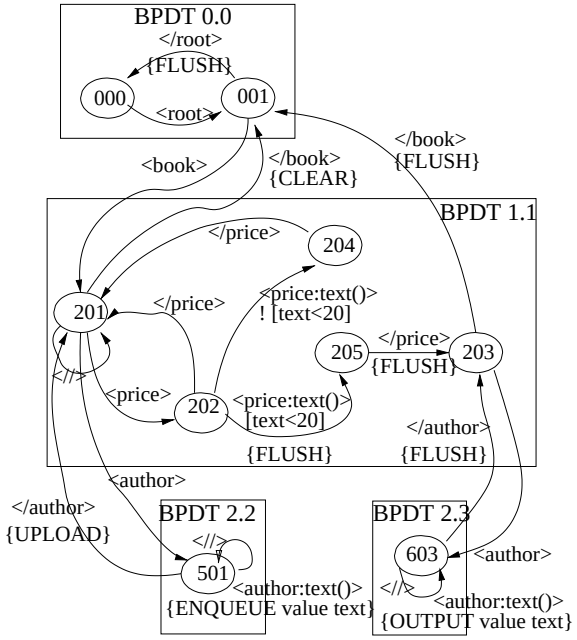


Figure 5: HPDT for `/book[price<20]//author`

other when it is false. As a result, the arcs in the HPDT are unchanged even if a predicate is replaced by its negation. However, the system has to make distinction between the case when the query contains a predicate and when it contains its negation instead in order to correctly identify the essential events.

The HPDT has to be pre-processed in order to provide the required information to XSQ-H. The information about the constant value in the predicate (e.g. `price`) is provided to the appropriate state (e.g. {201}) by simply traversing the automaton and propagating the constant values back. The information required to identify the arc that corresponds to the transition when the predicate is true can be obtained by observing that a true evaluation of a predicate implies that the data may contain the query result without any additional evaluations of the predicate. Thus, the state reachable from the arc corresponding to a true evaluation of the predicate has an acyclic path to an arc with the action `OUTPUT`. The arc corresponding to the false evaluation would have no such path since that would imply the automaton can produce query result by reaching the arc with the `OUTPUT` action without satisfying the predicate. A simple breadth-first search can be employed to determine the arc with this path in the automaton.

Example 5 Consider the query `/book[price<20]//author/text()` on the XML data of Figure 3. The HPDT for the query is shown in Figure 5. Initially, the set of current states is {001}. The arcs from this set of states correspond to the end tag of the `root` element and the start tag of `book` element. Thus, the essential events are the end of the `root` element and the start of `book` element, and they are added to the EventList. The XHManager processes the XHint in line 2 to obtain the offsets to these two SAX events. At the end of the XHint, the offset to the first `book` element is used to skip directly to line 7. When XSQ-H processes the start tag of the `book` element, the HPDT makes a transition from state 001 to 201. The state 201 has outgoing arcs with labels `author`, `price`, and `book`. The closure axes of the `author` label is identified by the arc labeled `//` from the state 201. The predicate constant and the operator associated with `price` element are stored in the arcs from state 202. XSQ-H uses this information stored in the HPDT to add an event corresponding to an `author` element and a predicate for the `price` to the event list. When the XHManager processes the XHint in line 8, the EventList consists of two events corresponding to `author` and `price` element. As the system can infer that the element does not contain any descendant with label `author` (from the descendant hint), it does not require to parse complex child elements. The offsets to the two essential elements, `price` and `author`, are provided by their respective child hints. The XHManager uses the first offset to jump to the `price` element. As this element satisfies the predicate, the set of current states changes to $\{201, 203\}$. The XHManager skips to the next essential element, `author` element on line 21. The actions defined on the state 603 output the text of the `author` element as the query result. In case of the second `book` element, the XHManager uses the XHint in line 24 to determine that the `price` element does not satisfy the predicate, and to infer that the `book` element does not contain any essential events. It therefore jumps directly to the end of that element, to line 33.

XSQ-H generates only 9 SAX events for processing the entire data as compared to 20 SAX events that would be generated by XSQ, resulting in a saving of more than 50%.

■

6.2 XHints and Tukwila

Tukwila [14] is an iterator-based streaming XQuery engine that processes XQuery expressions in a manner similar to standard relational query processing. The query optimizer uses basic operators to build and optimize a query plan for the query, which is

```
FOR $b IN datastream/root/book,
 $p IN $b/pub
 $d IN $b/disc
 $a IN $b//author
 $n IN $p/name
WHERE $d < 20
RETURN <publisher>
         <name> { $n } </name>
         <author> { $a } </name>
       </publisher>
```

Figure 6: Example XQuery

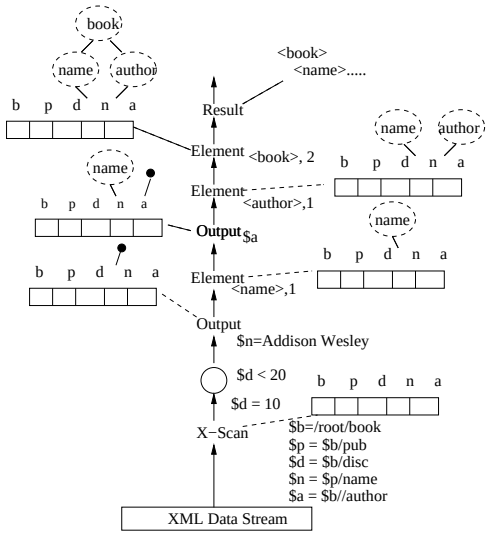


Figure 7: Query Plan for the Example XQuery

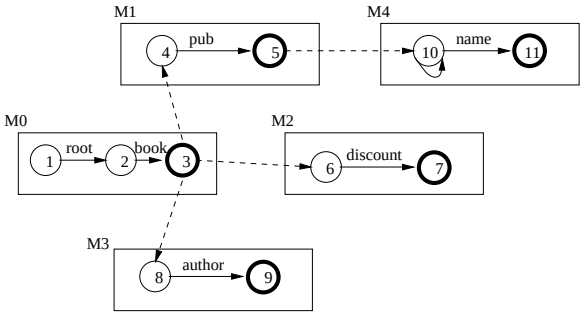


Figure 8: State Machines for the Example XQuery

passed on to the execution engine. Figure 7 depicts the query plan for the XQuery of Figure 6. It also shows the output subtree at different stages.

The execution plan uses a special operator called *X-scan* which is responsible for reading, parsing, and matching XML data with the regular expressions in the query. It assigns appropriate bindings to each XQuery variable and forwards them to remaining operators, where they are combined and restructured. The predicates declared in the `WHERE` clause are evaluated using a selection operator. The **element** operator constructs an element tag around a specified number of XML elements. The **output** operator is responsible for replicating the subtree value of the current binding to the query's output. The X-scan operator consists of a series of finite state machines that are driven by the input stream to produce the bindings for the XQuery variables. It converts all the XPath expressions (which are restricted forms of regular expressions) in the XQuery into state machines. Figure 8 depicts the state machines for the XPath expressions for the XQuery of Figure 6. Initially, the machine corresponding to the document root (M_0) is in the *active* mode. Whenever a machine reaches its accept state, it produces a binding for the variable associated with it. The machine then activates the dependent machines, which remain active while X-scan is scanning the value of this binding.

In absence of any prior information about the input data, the X-scan operator must parse every element in the stream. XHints can be used to reduce this parsing cost by replacing an X-scan operator with an XHint-compatible operator called *XH-scan*. The XH-scan operator uses the state machines to identify the essential SAX events while parsing the data. These events are identified using the labels of the arcs from the current states of the active state machines. When an active state machine makes a transition to a new state, labels on that state's outgoing arcs are the labels (element names) of the essential SAX events. Some of the transitions defined by the state machines may correspond to an predicate evaluation, which is performed by a selection operator in the query plan. In order to allow the XHManager to pre-evaluate such a predicate, the XH-scan operator is enhanced with information from the predicate's selection operator using simple query plan rewriting rules. The essential SAX events are registered with the XHManager which uses XHints to skip other irrelevant elements.

Example 6 Consider the execution of the XQuery of Figure 6 on the streaming XML data of Figure 3. The state machines representing the XPath expression are depicted in Figure 8. The processing of XHints by these state machines is very similar to the processing done by XSQ-H. The essential SAX events are defined by the labels on the arcs from the current state. Additional information about these SAX events, such

Database Name	Size (MB)	Text Size (MB)	Number of Elements (K)	Average Depth	Max. Depth	Average Tag Length	Xerces Parsing Time (s)	Expat Parsing Time (s)
SwissProt	109	37.1	2,977	3.56	5	6.58	23.7	5.81
DBLP	119	56.7	3,332	2.90	6	5.81	27.6	7.53
PSD	716	105.2	21,305	5.15	7	6.33	170.2	66.40

Table 1: Test Datasets

as the type of axes (child or descendant), along with predicates, can be stored with the label on the arcs.

The essential events correspond to the labels on the arcs from the current states of the activated machines. Initially, the state machine M_0 corresponding to the `/root/book` is activated. At the start of the document processing, the machine M_0 is in state 1. After parsing the topmost `root` element, it reaches state 2. This state has an arc with the label *book*, which is the essential SAX event at this point. The XHint at line 3 provides the offset (67) to the first `book` element in the data which can be used to avoid parsing the `mag` element. When the first `book` element is parsed at line 7, the machine M_0 reaches its accept state 3. At this stage, it binds the variable $b with the `book` element and activates the three dependent machines M_1, M_2 and M_3 for the expressions `$b/pub`, `$b/discount`, and `$b//author$` respectively. Now, the essential events correspond to the `pub`, `discount` and `author` labels. The arc of M_2 also contains the information (due to query plan rewriting) that the SAX event for the `discount` is required for a predicate evaluation and XH-scan accordingly registers the event by using the XHManager API function with appropriate parameters.

The XHManager obtains offsets to these elements from the XHint at line 8 and avoids parsing non-essential elements such as `title` and `price`.

■

7. Experimental Evaluation

We studied the throughput of XSQ-H using different kinds of hints and compared it with that of other systems, which do not use hints for query processing. We also studied the effect of query characteristics on throughput. Further, we investigated the effect of the buffer capacity in the XHint-generation phase on the throughput in the query-evaluation phase. Finally, we study the trade-off between the cost of generating hints and the throughput gain.

7.1 Experimental Setup

Our implementation of XSQ-H uses Java 1.4 and Xerces 2.4.0 (as the XML parser). We made a minor modification to Xerces to allow XSQ-H to instruct Xerces to skip a specified amount of data while parsing. We conducted the experiments on a PC-class machine with an Intel Pentium III processor and 1 GB of main memory, running the Red Hat 7.2 distribution of GNU/Linux (kernel 2.4.9). The maximum amount of memory available to Java Virtual Machine was set to 512 MB. The characteristics of the three real datasets used for our experiments are summarized in Table 1.

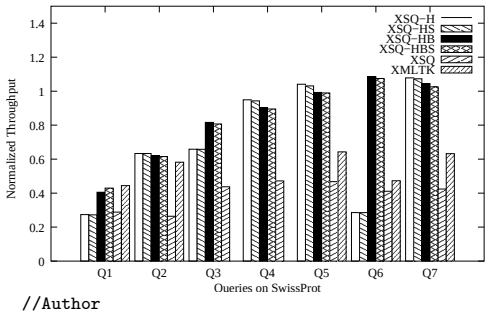
7.2 Throughput

The throughput gain provided by XHints depends on the stored hints as well as its scope. A XHint with richer information about the data allows the XHManager to skip more data and improve the throughput gain. Similarly, an XHint with a larger scope should provide a higher throughput since it allows the XHManager to skip more data.

In the first set of experiments, we measured the throughput gain achieved by XSQ-H on data with different kinds of XHints along these two axes. The effect of additional information provided by the descendant hints and XHints with partial scopes is evaluated using four kinds of XHints : (1) XHints generated offline without descendant hints (XHint-NS); (2) XHints generated in a streaming manner with end, child, and sibling hints (XHint-S); (3) XHints generated offline with descendant hints (XHint-NSB); and (4) XHints with descendant hints generated in a streaming manner (XHint-SB).

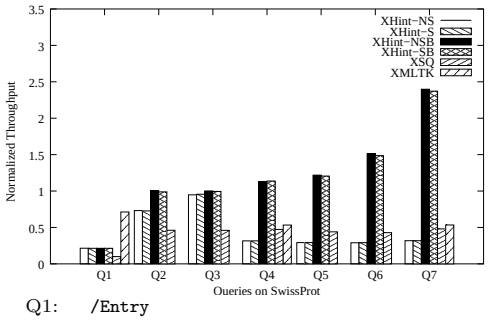
We compared XSQ-H with XSQ 1.0 [20] and XMLTK 1.0.1 [2], a streaming query engine implemented in C++. However, for queries with predicates, we compare only XSQ-H and XSQ because XMLTK 1.0.1 does not support such queries.

The throughput of an XPath query engine depends greatly on the parser. Engines such as XMLTK have a higher throughput than XSQ and XSQ-H since the



Q1: `//Author`
Q2: `/Entry/Features`
Q3: `/Entry[Org=Muridae]/Ref[Medline=9225337]/Cite/text()`
Q4: `/Entry/Ref[MedlineID=9225337]/Cite/text()`
Q5: `/Entry/Ref/Cite/text()`
Q6: `//Entry/Features//DOMAIN//Descr/text()`
Q7: `/Entry/Mod`

Figure 9: Throughput on SwissProt (1)



Q1: `/Entry`
Q2: `/Entry[Org]/Ref[MedlineID]/Cite/text()`
Q3: `/Entry/Ref[MedlineID]/Cite/text()`
Q4: `//CARBOHYD/text()`
Q5: `//Entry[Org=Eukaryota]//MUTAGEN`
Q6: `/Entry[Org=DISULFID]//Author/text()`
Q7: `//Noresult`

Figure 10: Throughput on SwissProt (2)

parser of XMLTK is faster than the parser of XSQ and XSQ-H. Thus, the difference in the parser performance effects the comparison between the throughput of these systems. In order to remove this effect, we normalize the throughput of a system with the throughput of its parser. The throughput of a system is divided by the throughput of the parser used by the system to obtain *Normalized Throughput*. This metric is used to evaluate the performance of all the systems.

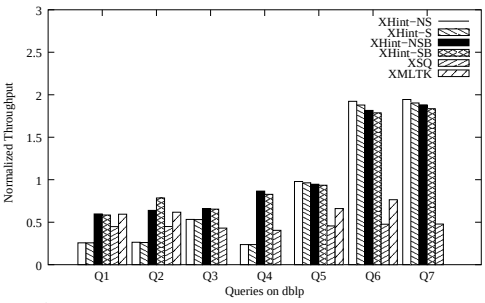
We measured throughput for 14 sample queries on each of the three test datasets. The queries were selected to represent the wide range of XPath queries including complex queries with both predicates and closures. The sample queries for the SwissProt are shown in Figures 9 and 10. The queries used on other two datasets are shown in Figures 11, 12 and Figures 13, 14 respectively. The buffer capacity used for streaming hint generation (for XHint-S and XHint-SB) was set to 50 KB.

We ran each query ten times and calculated the 95% confidence interval to evaluate the statistical significance of the results obtained. In all the measurements, the interval width was less than 1% of the measurement implying a high confidence in the experiment results. However we do not show the confidence interval in the result plots due to its small value. In addition to the small value, the confidence intervals of the throughput measurement of different systems do not overlap indicating that the comparison between different systems is statistically valid.

The results for the SwissProt dataset are summarized in Figures 9 and 10. For simple queries, such as Q2 and Q5 in Figure 9, XSQ-H performs better than XSQ for all four types of XHints by as much as 100%. However, XHint-NS and XHint-S perform marginally better than their counterparts that include the descendant hint. This difference in the gain is expected since XSQ-H does not use the descendant hint for processing such queries and the overhead of processing additional data in case of XHint-NSB and XHint-SB results in the slight performance degradation. The benefit of the descendant bitmaps can be observed for closure-containing queries such as Q1 and Q6 in Figure 9. For such queries, XHint-NS and XHint-S do not provide sufficient information for XSQ-H to skip substantial amounts of data, and the additional cost of parsing XHints lowers throughput. This information is provided in form of the descendant bitmap by XHint-NSB and XHint-SB, allowing the query processor to reduce the parsing cost. In case of Q6, the throughput of XSQ-H increases approximately 2.5 times compared to XSQ.

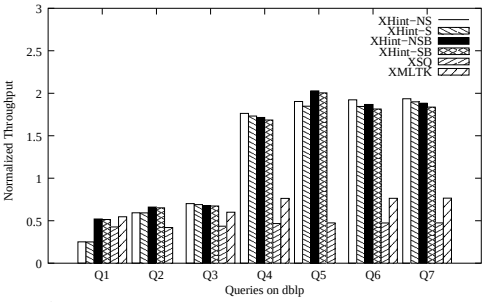
The benefit of the descendant bitmap is particularly large for queries that include labels that do not occur in the stream, as exemplified by Q7 in Figure 10. In case of XHint-NSB and XHint-SB, the descendant hint at the top level is used by XSQ-H to infer that the tag label `NoResult` does not occur at all in the data stream and skip the entire data resulting in a very high throughput not possible in case of XHint-NS, XHint-S or XSQ with no XHints. The other extreme, of a query that returns the entire stream, is exemplified by Q1 in Figure 10. (`Entry` is the top-level element.) In such a case, no data can be skipped and XHints do not provide any benefits to overcome their overheads.

The data digests used in XHint-SB and XHint-NSB improve the throughput of XSQ-H for queries with predicates, such as Q3 in Figure 9 and Q2 in Fig-



Q1: `//ee/text()`
Q2: `//editor/text()`
Q3: `/inproceedings[author]/title/text()`
Q4: `//article[year=1997]//cdrom/text()`
Q5: `/article/title/text()`
Q6: `/phdthesis/school/text()`
Q7: `/mastersthesis[url]/title/text()`

Figure 11: Throughput on DBLP (1)

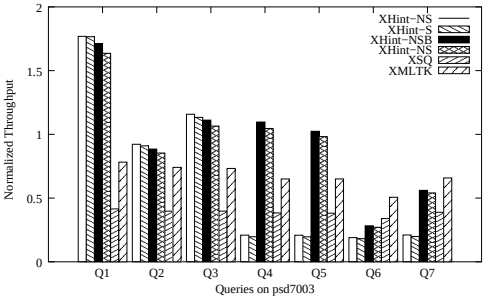


Q1: `//url/text()`
Q2: `/inproceedings[url]/title/text()`
Q3: `/inproceedings/booktitle/text()`
Q4: `/proceedings/title/text()`
Q5: `/phdthesis[year=1993]/title/text()`
Q6: `/phdthesis/title/text()`
Q7: `/mastersthesis/title/text()`

Figure 12: Throughput on DBLP (2)

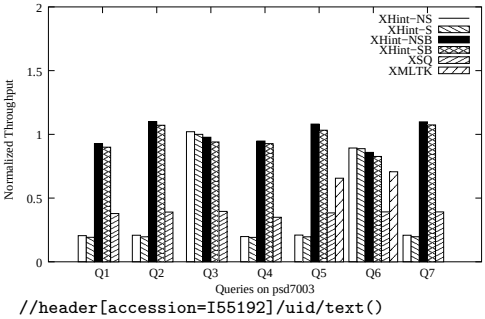
ure 10. The pre-evaluation of the predicate allows parser to skip more data in case of XHint-SB and XHint-NSB, resulting in higher throughput.

The throughput results for the DBLP dataset are summarized in Figures 11 and 12. As with the SwissProt dataset, XSQ-H outperforms XSQ by a significant margin for the sample queries. However, we observe relatively smaller differences in the performance of XSQ-H for different kinds of hints in case of simple queries, such as Q6 and Q7 in Figure 12. XHint-NS has the highest throughput, followed by XHint-S, XHint-NSB, and XHint-SB, in that order. The higher throughputs of hints generated offline is expected as for the SwissProt dataset. Further, as was the case with the SwissProt dataset, the performance degradation due to the limited buffer used for streaming generation of hints is small. The descendant hints in XHint-NSB and XHint-SB result in ad-



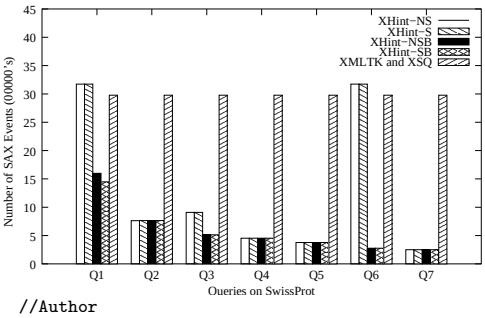
Q1: `/Database/text()`
Q2: `/ProteinEntry/header/text()`
Q3: `/ProteinEntry/sequence/text()`
Q4: `//protein/name/text()`
Q5: `//organism/formal/text()`
Q6: `//author/text()`
Q7: `//title/text()`

Figure 13: Throughput on PSD (1)



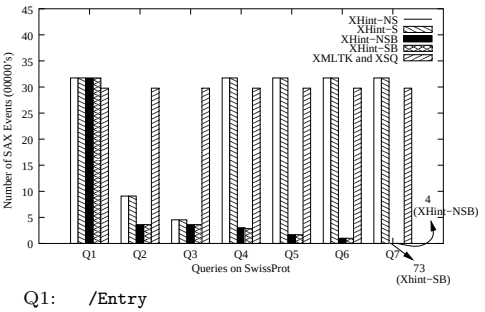
Q1: `//header[accession=I55192]/uid/text()`
Q2: `//feature[seq-spec=2]/status/text()`
Q3: `/ProteinEntry/summary[type=''complete'']/length/text()`
Q4: `//reference[contents=annotation]/refinfo//author/text()`
Q5: `//organism/common/text()`
Q6: `/ProteinEntry/reference/accinfo/accession/text()`
Q7: `//feature[seq-spec=1]/status/text()`

Figure 14: Throughput on PSD (2)




```
Q1:  //Author
Q2:  /Entry/Features
Q3:  /Entry[Org=Muridae]/Ref[Medline=9225337]/Cite/text()
Q4:  /Entry/Ref[MedlineID=9225337]/Cite/text()
Q5:  /Entry/Ref/Cite/text()
Q6:  //Entry/Features//DOMAIN//Descr/text()
Q7:  /Entry/Mod
```

Figure 15: SAX Events processed on SwissProt (1)




```
Q1:  /Entry
Q2:  /Entry[Org]/Ref[MedlineID]/Cite/text()
Q3:  /Entry/Ref[MedlineID]/Cite/text()
Q4:  //CARBOHYD/text()
Q5:  //Entry[Org=Eukaryota]//MUTAGEN
Q6:  /Entry[Org=DISULFID]//Author/text()
Q7:  //Noresult
```

Figure 16: SAX Events processed on SwissProt (2)

ditional computation for XSQ-H, but do not provide any additional benefit for simple queries. However the slight degradation in the performance of XSQ-H in case of XHint-NSB and XHint-SB is justified by the performance gain provided by the descendant hints for queries containing closure, as exemplified by Q1 and Q4 in Figure 11. The results for the PSD dataset, which is much larger than the other two test datasets, are qualitatively similar for the results on the SwissProt and DBLP datasets, as summarized by Figures 13 and 14.

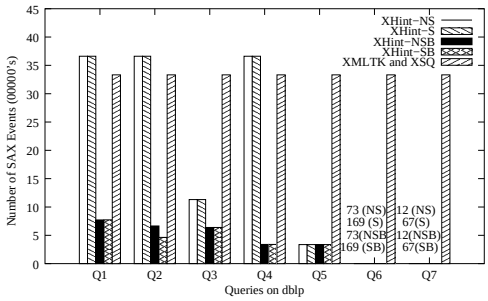
SAX Event Processing Above, we quantified the benefits of XHints by measuring the throughput improvements resulting from hints. However, the two base systems used in the comparison, XSQ and XMLTK, differ considerably in their design. In particular, XSQ uses an automaton to manage complex interactions with buffered data, while XMLTK uses a simpler design that does not use buffers. These system design choices are based on different goals: XMLTK supports a smaller subset of XPath (queries that can be answered without buffering) than XSQ, but is able to do so more efficiently due to the simpler design. The additional logic for processing complex queries in XSQ results in a lower throughput. In order to better isolate the benefits of XHints from the differences due to system design, we studied the number of SAX events processed by the XHint-enabled system, as a fraction of the total number of SAX events. (Systems that do not use hints must process all SAX events.) SAX events are known to be a significant part of query processing overhead for streaming systems [13, 20].

The results on the SAX events processed for the SwissProt database are summarized in Figures 15 and 16. We note that XHints result in a significant reduction in the number of SAX events that must be processed. As expected, for queries with closures, XHint-SB and XHint-NSB provide a larger reduction than XHint-NS and XHint-S, due to the descendant hints in the former pair. Due to the extremely small value, the number of SAX events processed for query Q7 in Figure 16 by XSQ-H for XHint-NSB and XHint-SB (4 and 73 respectively) are shown separately. The data digest also reduces the number of SAX events processed, as can be observed for Q3 in Figure 15.

Results for the DBLP (and PSD) datasets are qualitatively similar, and are summarized by Figures 17 and 18 (respectively, Figures 19 and 20). The number of SAX events generated by XSQ-H for different types of XHints is approximately same for queries with only child axes as it can be seen for Q5 and Q3 in Figure 17 and Figure 18, respectively. Similar observation can be made for queries like Q2 and Q3 in Figure 19 on the PSD database. In case of queries with closures like Q4 in Figure 17 and Q6 in Figure 19, the descendant hint (in XHint-NSB and XHint-SB) reduces the number of SAX elements processed significantly. The number of SAX events for such queries are so small that they cannot be represented graphically. We have instead shown the value itself along with the XHint it corresponds to.

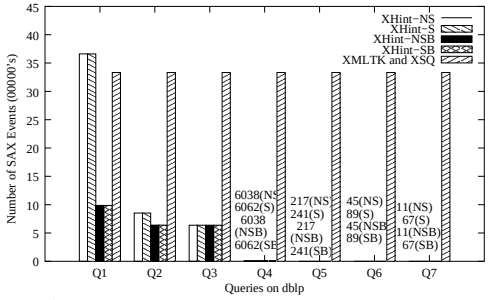
7.3 Query Characteristics

We now outline the results of our experiments studying the effects of various query characteristics on the gain provided by XHints. We have used XMLTK 1.0.1 and XSQ 1.0 for the comparing the performance



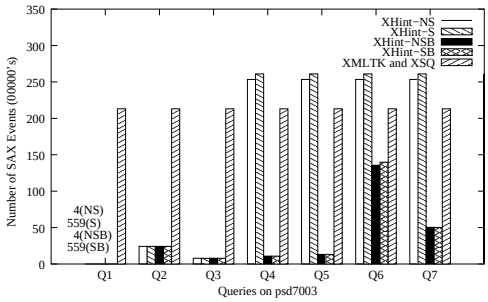
Q1: `//ee/text()`
Q2: `//editor/text()`
Q3: `/inproceedings[author]/title/text()`
Q4: `//article[year=1997]//cdrom/text()`
Q5: `/article/title/text()`
Q6: `/phdthesis/school/text()`
Q7: `/mastersthesis[url]/title/text()`

Figure 17: SAX Events processed on DBLP (1)



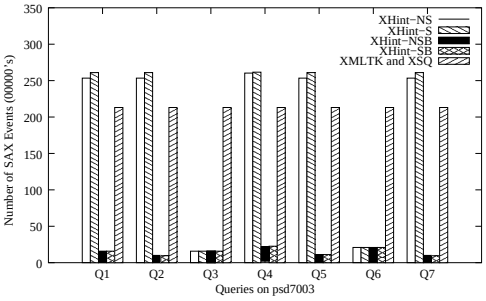
Q1: `//url/text()`
Q2: `/inproceedings[url]/title/text()`
Q3: `/inproceedings/booktitle/text()`
Q4: `/proceedings/title/text()`
Q5: `/phdthesis[year=1993]/title/text()`
Q6: `/phdthesis/title/text()`
Q7: `/mastersthesis/title/text()`

Figure 18: SAX Events processed on DBLP (2)



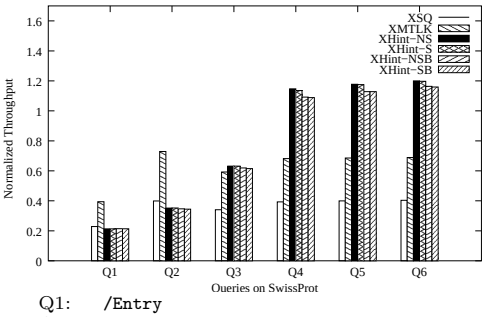
Q1: `/Database/text()`
Q2: `/ProteinEntry/header/text()`
Q3: `/ProteinEntry/sequence/text()`
Q4: `//protein/name/text()`
Q5: `//organism/formal/text()`
Q6: `//author/text()`
Q7: `//title/text()`

Figure 19: SAX Events processed on PSD (1)



Q1: `//header[accession=I55192]/uid/text()`
Q2: `//feature[seq-spec=2]/status/text()`
Q3: `/ProteinEntry/summary[type=''complete'']/length/text()`
Q4: `//reference/refinfo//author/text()`
Q5: `//organism/common/text()`
Q6: `/ProteinEntry/reference/accinfo/accession/text()`
Q7: `//feature[seq-spec=1]/status/text()`

Figure 20: SAX Events processed on PSD (2)



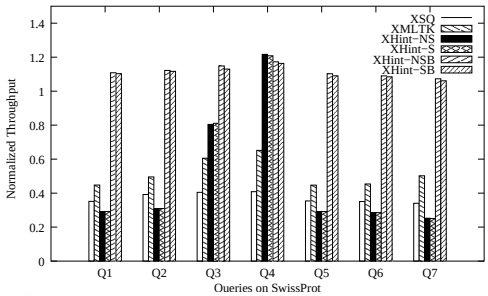
Q1: `/Entry`
Q2: `/Entry/text()`
Q3: `/Entry/Features`
Q4: `/Entry/Features/DOMAIN`
Q5: `/Entry/Features/DOMAIN/Descr/text()`

Figure 21: Effect of query length on throughput

of XSQ-H.

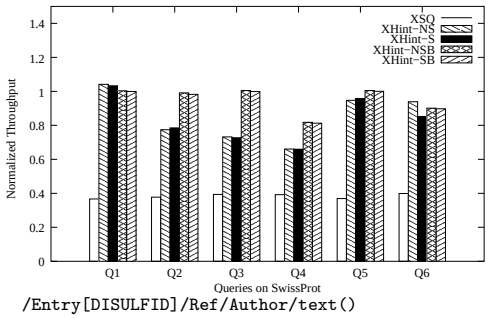
Query Length We define the length of an XPath query to be the number of location steps in the query. The effect of increasing query length on throughput for the SwissProt dataset is summarized by Figure 21. We observe that the throughput of XSQ-H increases with query length. This result is intuitive because longer queries usually typically produce smaller query results and allow XSQ-H skip larger amount of data resulting in a higher throughput. XHint-NS and XHint-S provide a slightly better throughput compared to XHints with descendant hints (XHint-NSB and XHint-SB) since the queries do not contain closure and the descendant hints constitute an extra processing overhead.

Descendant Axes Since evaluating the descendant axis entails a subtree traversal, it is reasonable to expect a lower throughput as the number of loca-




```
Q1:  //Entry/Features/DOMAIN/Descr/text()
Q2:  /Entry//Features/DOMAIN/Descr/text()
Q3:  /Entry/Features//DOMAIN/Descr/text()
Q4:  /Entry/Features/DOMAIN//Descr/text()
Q5:  //Entry//Features//DOMAIN//Descr/text()
Q6:  //Entry/Features//DOMAIN//Descr/text()
Q7:  //Entry/Features/DOMAIN//Descr/text()
```

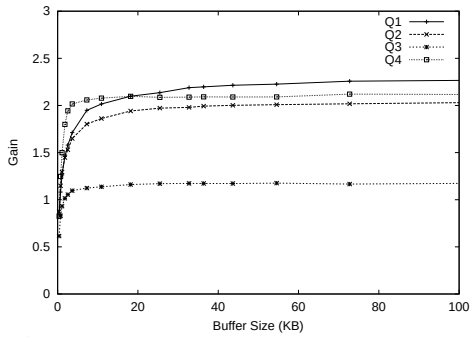
Figure 22: Effect of descendant axes on throughput




```
Q1:  /Entry[DISULFID]/Ref/Author/text()
Q2:  /Entry[Org=Eurkaryota]/Features/MUTAGEN
Q3:  /Entry[REPEAT]/PROPEP
Q4:  /Entry[Org]/Ref[MedlineID]/Cite/text()
Q5:  /Entry[Org=Muridae]/Ref[MedlineID=9225337]/Cite/text()
Q6:  /Entry/Ref[MedlineID]/Cite/text()
```

Figure 23: Effect of predicates on throughput

tion steps using the axes rises. The effect of varying the number and position of descendant axes for queries on the SwissProt dataset is summarized by Figure 22. As expected, the throughput of XHint-S and XHint-NS is low for queries with descendant axes. Exceptions are the results for queries Q3 and Q4. This result is explained by noting that the descendant axes in these queries are deep (to the right) in the query, entailing traversals of smaller subtrees than those necessary when the descendant axis is higher (to the left) in the query. In contrast to XHint-S and XHint-NS, the XHint-SB and XHint-NSB provide consistently high throughput. The throughput is slightly higher for queries with a closure axis deeper in the expression (such as Q4). A deeper descendant axis allows XSQ-H to ignore a larger number of elements as compared to queries containing the descendant axis closer to the first location step (such as Q1 and Q5).




```
Q1:  /Entry/Features/DOMAIN/Descr/text()
Q2:  /Entry[Org]/Ref[MedlineID]/Cite/text()
Q3:  /Entry//Descr/text()
Q4:  //Entry[Org]/Descr//text()
```

Figure 24: Throughput gain for different buffer sizes

Multiple Predicates Figure 23 summarizes the results of some experiments studying the effect of predicates on throughput. As expected, the data digests used by XHint-SB and XHint-NSB allow XSQ-H to pre-evaluate predicates and reduce the number of SAX events that must be processed. As a result, these two XHint schemes yield a higher throughput than XHint-S and XHint-NS. However, XHint-SB and XHint-NSB do not outperform the other two schemes for query Q1. This result is explained by noting that the labels in the predicate of Q1 does not occur frequently in the dataset. Therefore, the number of SAX events skipped using the data digest is small and does not yield enough benefits to overcome the overhead of processing the additional hint information.

Buffer size The range of information in an XHint depends on the size of the buffer used during XHint-generation at the data source. In general, a larger buffer permits XHints that range over larger amounts of data, allowing the XHManager to potentially skip more data. Thus, it is natural to expect performance to degrade as the buffer size is reduced. For any buffer size, it is easy to construct synthetic scenarios in which XHints perform poorly. From a practical standpoint, the important question here is the relation between upstream buffer size and downstream throughput for realistic datasets and queries. In particular, we are interested in determining the buffer sizes that determine a good performance point. For this purpose we generated XHints for SwissProt dataset with different buffer sizes and measured the throughput on the resulting streams. We have used XHint-SB in the experiment instead of XHint-S since

Q1:	`//red`
Q2:	`/scheme/color/red`
Q3:	`/scheme[code=2]/color/red`
Q4:	`/scheme[code=2]//color/red`

Table 2: Queries used on Synthetic Datasets

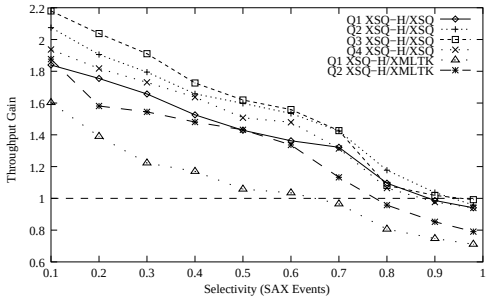


Figure 25: Effect of Query Selectivity

the previous experiments have demonstrated that XHint-SB perform well for all queries with little overhead as compared to XHint-S which do not perform well for queries with closures. The results are summarized by Figure 24. The main conclusion we draw from this experiment is that a very small buffer suffices to provide significant benefits for XHints for this dataset. We obtained similar results for the other datasets.

Selectivity It is natural to expect XHints to yield greater benefit for queries that are highly selective for the input data stream. We define selectivity in the usual manner, as the ratio of the number of SAX events in the query result to the total number of SAX events. In order to measure the effect of query selectivity on throughput, we generated ten synthetic datasets containing elements with `red` and `blue` as labels. All the datasets were similar in their characteristics except in the proportion of the elements with the label `red`. We ran the four queries depicted in Table 2 on each dataset and measured the throughput for different values of the selectivity. Figure 25 displays the throughput gain of XHint-HB compared to XSQ and XMTLK for different values of selectivity. Throughput gain of XHint-HB compared to another system is defined as the ratio of the throughput of XHint-HB to the throughput of that system. As XMLTK does not support predicates, XHint-HB is compared with XMLTK for only the first two queries. As expected, XHint-HB provides a high throughput gain for low selectivity compared to XSQ and XMLTK, with a graceful degradation in performance as selectivity increases. Although we have not shown any results for other XHint schemes, they also follow the same behavior with change in selectivity. Although the actual gain differs depending on the scheme, the gain decays gracefully with increase in selectivity.

7.4 XHint Generation

Hint-Generation Throughput We measure hint-generation throughout as the number of elements processed per second. We studied hint-generation throughput for the test datasets using buffers of different sizes. The results are summarized in Figure 26. We observe that as the buffer size is increased, throughput increases initially, but gradually falls after reaching a peak at around buffer size of 60 KB. When the buffer used during hint generation is small, the data stream is split into a larger number of chunks, resulting in the insertion of a large number of XHints, which in turn could lead to a low throughput. On the other hand, large buffer sizes suffer from the overhead of storing more information in XHints which require additional computation of offsets to various elements.

Size Overhead The insertion of XHints into a data stream increases the size of the stream that must be read by downstream processors. We measured this overhead in terms of the percentage increase in the data due to addition of XHints for datasets of different sizes. We used one real dataset (PSD) and two randomly generated dataset (RAND1 and RAND2) for our evaluation. We generated XHints with descendant bitmaps in a streaming fashion for different dataset sizes. The size of the buffer used to generate the XHints was fixed at 50 KB. We chose XHint-SB to demonstrate the upper bound on the data overhead since it contains the maximum information out of the four types of XHints and thus, incurs the highest data overhead.

As Figure 27 indicates, the percentage overhead in the data decreases with increase in the dataset size. Small sized datasets have a low number of elements and the XHint constitute a significant portion of the data in terms of size. As the size of the data increases, the number of XHints needed to store offset summary of the data does not increase in the same proportion as the data elements since the atomic and text nodes of the data do not contain XHints. As a result, the percentage overhead of inserting XHints decreases as the data size increases.

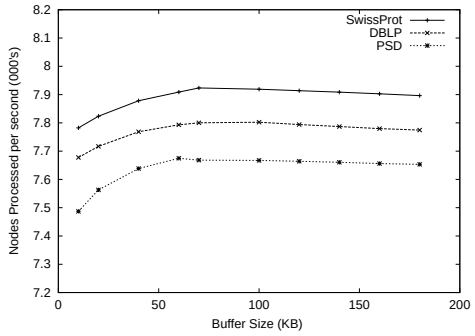


Figure 26: Throughput of XHint Generation

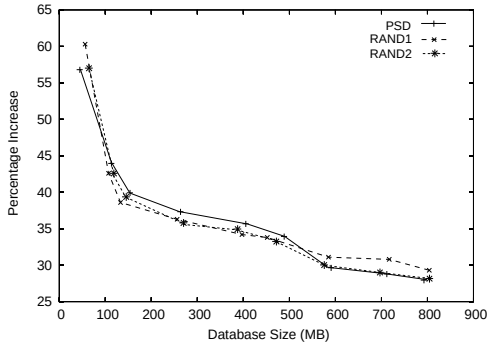


Figure 27: Increase in data size

8. Related Work

The idea of augmenting a data stream to assist query processing was first proposed as *punctuations* [21]. Punctuations are designed to assist blocking operators read the entire data before emitting an output. The punctuations are in the form of predicates that are not seen in the stream following the punctuation. It allow a query processor to infer the absence of certain elements in the data and output the result early. A binary-encoded index called *SIX* [11] stores the offsets to the beginning and end of elements in the stream. A query processor can use these offsets to skip processing data in much the same way as our XHint-NS scheme. The main difference between XHints and SIX is that XHints store a much greater variety of information, permitting skipping data in a greater number of situations.

The *MatchMaker* system [16] addresses a similar problem of matching an incoming data stream to a large number of queries. Unlike conventional query processing problem where the number of queries is small compared to the size of data, the system processes a large repository of queries on relatively small data. The system stores the query patterns in an index that supports efficient lookup for queries satisfying a particular pattern. It uses this index to tag each XML node with the queries that it satisfies.

Several query engines have been presented for streaming XML data. The XML Streaming Machine (XSM) system [18] translates an XQuery expression into a network of transducers that correspond to basic subexpressions of XQuery and reduces the network to a single XSM by repeatedly merging transducers. The final XSM is optimized with respect to time and space using both data and query characteristics. XSQ [20] and XPush [13] use an automaton-based approach to process streaming XML data. XSQ builds a hierarchal automaton called HPDT from a given XQuery expression and returns the portions of a large streaming XML document that match the query. On the other hand, XPush processes a given set of XPath filters with predicates on a stream of XML documents to compute which filters are satisfied by each document. It builds a lazy deterministic finite automaton with each state representing a set of predicates. This state machine stimulates the execution of the workload of the XPath filters on a XML document and returns a set of XPath ids satisfied by the document.

There are several methods for indexing semistructured data in a non-streaming environment. *Dataguides* [10] provide a concise structural summary of semistructured database by storing all

label-paths occurring in the database. The data structure can be incrementally updated and provides a dynamic schema of the underlying database. A Template Index or *T-Index* [19] consists of an automaton corresponding to a data-path template and list of nodes in the database that satisfy the path expression. The automaton accepts data paths belonging to the set defined by the template and returns pointers to the relevant nodes. *Index Fabric* [7] uses a similar principle by storing data paths in indexes encoded as strings. The index is highly optimized for string search and provides offsets to the data nodes stored in a relational database.

The XML Indexing and Storage System (*XISS*) [17] indexes and stores XML data using a numbering scheme. Each element is identified by a tuple that can be used to determine the ancestor-descendant relationship between two nodes efficiently. The system also stores indexes (implemented as B^+-trees) for searching documents with a given name or attribute. The indexes along with the numbering allow efficient processing of regular path expressions on an XML database. A complex query is decomposed into simpler expressions and joining algorithms are employed to merge the results. An $A(k)$-index [15] is based on the concept of k-bisimilarity and maintains, for each node, a record of incoming paths to a node of length at most k (local structure). The system assumes that long complex path queries are rare and reduces the index size by grouping nodes into equivalence classes based on local structure.

An adaptive indexing scheme for non-streaming XML data is presented in APEX [6]. The index consists of two structures. A graph is used to store the structural summary of the data and a hash tree stores label-paths. Each node in the hash tree is a hash table with entries pointing to a node of either the graph or the hash tree. The hash tree is useful for determining the nodes in the graph for a given label path and for updating the index. APEX stores indexes for only the most frequently used paths, which can be updated incrementally depending on changes in the query workload. It would be interesting to use this idea and study how query workload can be used to estimate the utility of an XHint in terms of the speedup it provides and insert only the most useful XHints based on this estimate. More recently, another dynamic index called *ViST* was proposed [22]. It represents XML databases and queries as structure-encoded sequences, reducing the query-evaluation problem to the problem of matching subsequences. Unlike other indexes like XISS, ViST processes the query as whole without decomposing it into sub-queries, saving on expensive join operations required to merge the subquery results. An adaptive version of $A(k)$-indexes, called $D(k)$-indexes [5], provides an updating mechanism storing only the most useful path indexes depending on the query workload. The $D(k)$-Index is also based on k-bisimilarity and constructs equivalence classes of nodes based on local structural information. Instead of a global value for the similarity parameter k, a $D(k)$-index uses different values for different equivalence classes, depending on the query workload.

The XPath accelerator [12] provides an index that handles all XPath axes, in contrast to the methods mentioned above, which use regular expressions. It uses four major axes, ancestor, descendant, following, and preceding, to divide the XML document into four regions. The pre-order and push-order rank of a node is used to encode the information about the region to which it belongs. Standard database indexes such as B-Trees and R-Trees are used to index the position of the node.

A number of systems have been developed to address the closely related problem of *filtering* XML documents based on XPath queries. An *index-filters* [3] uses an idea very similar to XHints to skip irrelevant data. It constructs an inverted index on the XML data tagging each element with a unique identifier. The identifier contains information that can be used to efficiently infer ancestor and descendant relationships between elements. The union of multiple XML queries is represented as a prefix tree. Each node of the tree contains a list of indexed positions of elements that match the node label. The index-filter algorithm uses the prefix tree with the index to identify elements that may not satisfy any query expression and skips them, thus avoiding the cost of parsing. The index-Filter requires access to the complete XML dataset in order to generate identifiers and cannot be used in a streaming environment. It also needs a sorting phase which may prove expensive for large XML databases. The information stored in these indexes is more restricted than that in XHints. For example, it excludes information about the data stored in elements that can be used to avoid unnecessary processing in case of queries with predicates.

XFilter [1] and *YFilter* [8, 9] process multiple queries on XML documents using finite automatons. XFilter uses finite-state automata and an inverted index to match XML documents with queries. The XFilter engine translates each XPath expression into a separate finite-state automaton. An inverted index is built over the states of these automata, essen-

tially producing a hash table over the element labels. This index allows the system to simultaneously execute all the automata and identify the set of queries relevant to a data element. YFilter improves on this scheme by exploiting the commonality among XPath expressions. It combines all the XPath query expressions into a single non-deterministic automaton and merges common query prefixes into a single state, thus avoiding redundant processing. An *XTrie* [4] indexes substrings of XPath expressions that contain only the parent-child operator. The index consists of a table and a trie. The indexing scheme is used to filter the documents by first detecting the occurrence of matching substrings using the trie and then obtaining the matched queries from the table. Some additional run-time information is used to avoid redundant matching of the query with the document.

9. Conclusion

Semistructured data is often serialized as text and presented in a streaming form (e.g., news feeds presented as streaming XML). In many situations, it is impracticable or undesirable to instantiate this data (e.g., due to high data rates of the source or resource constraints at the client, which may be a small, mobile device). In these situations, query processing requires that the stream of serialized data be parsed on-the-fly, and parsing consumes a large fraction of the processing time. Therefore, further improvements in query-processing efficiency require methods that reduce the parsing costs. We proposed an a indexing framework in which an XML stream is augmented with indexing information (XHints) at or near the source. Although this task requires additional processing at the source of the stream, it results in lower processing costs at the stream's consumers, or other downstream processors. This transfer of processing costs is beneficial because not only may the source have access to significantly greater computing resources than the consumer (server vs. handheld) but a large number of consumers can benefit from the indexing at the server (e.g., one server that streams data to hundreds of handheld units).

XHints may be regarded as indexes that are interspersed with the data in the stream. Specifically, XHints store four kinds of hints (offsets): end, child, sibling, and descendant. We described how query processors may take advantage of these hints, using XSQ (which uses an automaton-based approach) and Tukwila (which uses the standard iterator-based approach) as specific examples. We quantified the costs and benefits of XHints using an experimental study on real and synthetic datasets using our implementation of a XHint-enabled version of XSQ. Our results indicate that XHints that are generated on-the-fly by the data source using very little buffer space provide significant benefits to downstream query processors.

The hints described in this paper are inserted in a manner oblivious of the query load. Although our experiments indicate that such hints are beneficial, even greater benefits may be realized by tuning the hints to the query load (when such information is available). In continuing work, we are studying this hint-selection problem, which is essentially the index-selection problem for streaming data.

References

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, September 2000.

[2] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *Proceedings of Programming Language Technologies for XML (PLAN-X)*, October 2002.

[3] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation vs. index-based XML multi-query processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, March 2003.

[4] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 235–244, February 2002.

[5] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for the graph-structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 134–144, June 2003.

[6] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 121–132, June 2002.

[7] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 341–350, August 2001.

[8] Y. Diao, M. Altinel, M. J. Franklin, H. Zang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28(4), December 2003.

[9] Y. Diao, P. Fischer, and M. J. Franklin. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 341–344, February 2002.

[10] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 436–445, August 1997.

[11] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 173–189, January 2003.

[12] T. Grust. Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 109–120, June 2002.

[13] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 419–430, June 2003.

[14] Z. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. In *The VLDB Journal (VLDBJ)*, 2003.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 129–140, March 2002.

[16] L. V. Lakshmanan and S. Parthasarathy. On efficient matching of streaming XML documents and queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 142–160, March 2002.

[17] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal (VLDBJ)*, pages 361–370, 2001.

[18] B. Ludäscher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 227–238, August 2002.

[19] T. Milo and D. Suciu. Index structures for path expression. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 277–295, January 1999.

[20] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, June 2003.

[21] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Punctuating continous data streams. Technical report, OGI School of Science and Engineering at OHSU, 1999.

[22] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 110–121, June 2003.