Processing XPath Queries with Selective Parsing using XHints

Akhil Gupta Sudarshan S. Chawathe

Department of Computer Science, University of Maryland, College Park, MD-20742, USA

{akhilg, chaw}@cs.umd.edu

November 25, 2003

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, making such processing worthwhile. In this paper, we focus on hints that are designed to improve the throughput of a streaming XML query engine. We present a set of hint schemes and describe how can a query engine improve its performance by taking advantage of the hints. Finally, we demonstrate the benefits of our approach using an experimental study.

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 relevant 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, 18] spend a large fraction (typically well over 50%) of their CPU resources on simply parsing the input stream. Clearly, this fact limits the amount of further improvement achievable by techniques that operate post-parsing. Thus, there is a need for methods that can sidestep the cost of parsing data that is irrelevant to a query. We define irrelevant data as the data whose presence in the input stream does not affect the results of the query in any way.

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 is relevant if and only if it contains both *title* and *discount* child elements. Thus, the second *book* element (lines 21-30 of Figure 1) is irrelevant and can be skipped entirely by a query engine as it does not affect the query result. Similarly, even in the first *book* element, elements other than the *title* element do not affect the query result and can be skipped by the query engine reducing the parsing cost.

Indexes have been typically used to avoid parsing irrelevant data by providing direct access to the elements. But traditional approaches for indexing semi-

```
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

structured data [6, 7, 10, 17] cannot be applied in streaming environment since the data is unbounded. Moreover, since the data stream in many applications is generated and sent to clients in real-time, the indexes have to be generated on the fly.

An early example of an index for streaming data is the stream index (SIX) [11] for XPath queries on XML data. The index stores pointers to the beginning and end of each element in a compact binary form and is used by the query processor to directly skip to the elements that match the query.

SIX has been shown to have a very low overhead and can provide significant speedup to streaming XML query processors for certain queries even with these simple hints. But SIX has very limited utility for more complex queries containing closures and predicates. Since the index does not incorporate any ancestor-descendant relation, hints with more information are required to efficiently process even moderately complex queries such as */book//address*. In addition, as the index contains the start and the end offsets for all elements, it cannot be generated for partial streaming XML data.

In this paper, we propose placing strategically designed annotations or hints in the stream. These hints, called XHints, may be viewed as a temporally distributed index on the input stream. They store structural information about the data which can be used by a query engine to identify the irrelevant portions and avoid parsing them. We also describe a well-defined mechanism that can be used by a query engine to process XHints in a transparent and modular fashion.

Since XHints are inserted as part of the input data stream, the structure of XHints depends on the data representation scheme. Various schemes such as OEM and XML have been suggested to represent semistructured data[**Add references to this line**]. Most of these schemes model the data in form of a tree with nodes representing elements or objects and the edges defining the hierarchical relationship between different elements.

XHints can be used with any such tree-based representation scheme. The only requirement on the scheme for our approach to work is that the tree-structured data is sent across with child elements completely nested between the parent elements. However, for concreteness, we discuss XHints in context of streaming XML data. XML is one of the most popular data representation scheme and has come out as the *de facto* standard for representing semi-structured data in recent years. Moreover, XML represents the data in a nested fashion making it an ideal choice for XHints.[**Reword this para. Also not sure of the exact placement of these two paras (this one and the one above it. Do I put it in the end or here?**]

The XHints can be generated at the server end and sent along with the data stream to the clients. Generating XHints does not require access to the entire data stream. They are well adapted to being generated in a windowed manner, where portions of data are buffered and augmented with XHints, allowing generation of the XHints on-the-fly in real-time.

Note, however, that XHints involve sending additional data to clients and thus do not save on network transmission costs in a unicast network. In a multicast networks, 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).

The main contributions of this paper are summarized below

1. To the best of our knowledge, this work is the first which attempts to make semistructured query processing more efficient in a streaming environment by allowing the parser to identify and skip irrelevant data.

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 that illustrates the benefits of our approach.

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 and possible future work is presented in section 9.

2. System Architecture

A normal streaming XML processor uses an XML parser which generates SAX events for every element in the data stream. Thus, the processor has to parse and process SAX events for all data elements, even though a large portion of the data may not be part of the query result. This extra processing of irrelevant elements results in a high overhead and subsequently, a lower query result throughput.

XHints are designed to reduce this overhead by allowing the parser to skip portions of data which do not contain any query result. The processing of XHints is completely separated and hidden from the query engine. They are handled by a XHint Manager that provides a common interface to the query processor. As described later, the query processor is only expected to perform minimal additional processing to assist the XHint Manager. Figure 2 displays the system architecture of a XHint-enabled query processor.

The XHint Manager acts as a proxy between the query processor and the XML parser. The parser generates the SAX events for the processed data and sends them to the XHint Manager for processing. The XHint Manager may handle the event internally (if it is a XHint) or forward it to be processed by the query engine (if it is a data element). It also maintains a list of *interesting* SAX events referred as the EventList.

An *interesting* SAX event is informally defined as an event which the query processor has to process in order to evaluate the query correctly. For example, a query engine with the query */book/discount* on the example XML data (Figure 1) has to 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 *interesting* event for this particular query.

If the query contains a predicate as in */book[price < 20]/pub//name*, the SAX event corresponding to the element with the predicate is *interesting* if and only if the predicate is satisfied by the element. In the case of example query, the *book* element is not interesting if 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 is only interested in elements containing a descendant with that particular label. For example in the above query, any child element of the *book* element that may contain a descendant with label *name* is an *interesting* SAX event.

The XHint Manager uses the EventList along with the information provided by the XHints to identify the irrelevant portions of data in the stream (as explained later) and request the parser to skip them by providing appropriate offsets.

The list of *interesting* events changes temporally as the data is processed by the query processor and has to be updated accordingly. For example in case of the query mentioned above, when the query processor is at the start of the document shown in Figure 1, the SAX event corresponding to the *book* element is an interesting event. But when the XML parser parses the start tag of *book* element at line 5, the *book* SAX event is replaced by the *discount* SAX event as the interesting event. The *book* element again becomes the interesting event when the end tag of the *book* element is processed at line 20.

The query engine is responsible for maintaining the

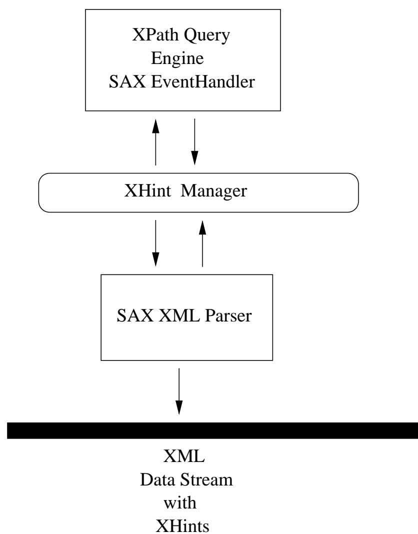


Figure 2: System Architecture

list of SAX events in the EventList. In order to do so, it should be capable of identifying the interesting SAX events which may contain the query result. The exact mechanism by which it performs this identification depends on its design but we outline the general idea by describing how can be done for two common query system architectures.

The query engine interacts with the XHint Manager using a simple well-defined API. The API consists of two functions which may be used by the query engine to update the EventList. These functions are:

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

   This function lets the query engine to register an event with the XHint Manager that needs to be processed by it. It returns an unique identifier for the registered event. The *interesting* SAX event is identified by its URI and the tag label of the element called localName. In addition to these identifiers, if there is a predicate associated with the event, it is represented as a tuple consisting of the element label in the predicate, the operator such as $<$ or $=$ and the constant value. The integer Type is used to denote whether the query engine is interested in the SAX event as a descendant or an immediate child of the current element parsed.

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

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

Although the API described above can be used by any query engine to update the list of *interesting* events, we illustrate the use of XHints for two query engines based on different architectures. We focus primarily on XSQ [18], an automaton based streaming XML processor and provide a detailed description of how can XHints improve processing efficiency for different types of XPath queries. Further, in order to illustrate the generality of the approach, we present an insight on how XHints can be used in Tukwila [13], an iterator based query engine.

3. XHints

XHints are special XML elements that are used to store structural summary about the data in form of different attribute values. The name of the attribute determines the kind of structural information stored in it. These XHints are inserted in the data itself and can be used by a query processor to process the data more efficiently.

Although the attributes of a XHint can store a variety of information, we only use four kinds of attributes or hints for XPath query processing over streaming XML data. These four types of hints are 1) *End Hint* 2) *Child Hint* 3) *Sibling Hint* 4) *Descendant Hint*. As explained later, these four types of hints are sufficient to store information useful for a variety of queries. An example XML dataset is shown in Figure 4 with the XHints highlighted in italics.

The end hint of a node contains the offset from the end of the XHint to the end of the node. It is stored as the value of attribute *"end."* This hint allows the parser to directly skip to the end of an element if the node or any of its part does not belong to the query result.

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 with l as the attribute name, in form of a list separated by colon. The attribute *author* of the XHint at line 24 in Figure 4 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 an element can potentially contain an unlimited number of child elements with the same label, the size of the XHint can become very large if all the offsets are stored in it. *Sibling Hints* are used in order to limit the maximum size of a 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, say c child node with a particular label. The offsets to the next, say n node are stored in the cth child node. The $(c+n)^{th}$ node contains the offset to the next n nodes and so on. In this manner, the sibling hints allow storing the offsets to a large number of children nodes without making one particular XHint very large. The XHint at line 8 in Figure 4 contains a sibling hint to the next *book* element.

For example, if the *root* element in the dataset shown in Figure 1 contained a large number, say 10000 *book* elements instead of 2, storing the offsets to all of the elements in the XHint of the *root* will result in a large XHint. Additionally, storing all the offsets in the memory is inefficient. Instead, we only store the offsets to, say the first 500 elements, in the XHint of the *root* tag. The offsets to the next 500 elements are stored as a sibling hint of the XHint of the 500th element. The 100th element stores the offsets to the next 500 elements and so on.

XHints can also be used to store information about the text contained in an element. We propose storing a summary of the text node in form of a *descendant digest* along with the offset as part of the child and the sibling hint. The query engine can process queries with predicates more efficiently by using this data digest in XHints to pre-evaluate predicates and skip elements that do not satisfy the predicates.

If the text is an alpha-numeric string, we store the first s characters, typically 3, of the string. 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. For example, the XHint for the first *book* element at line 8 contains the first three characters of the text in the *author* element, which can be used to evaluate predicate beforehand as explained later in Example 3.

If the text of the element in the predicate is numeric, the predicate may contain inequality operator and comparing the first s characters is insufficient to make inferences about it. We use a different scheme to generate the descendant digest for such text nodes. During the XHint generation phase, we obtain the range of numerical constants occurring for each label and store them as an attribute called *Hash* of the special XML element META. The entire range is then divided into a fixed number of equal-sized intervals and the interval index of the numerical text of an element is stored as its descendant digest.

In case of an equality operator, if the interval index of the constant of the predicate does not match the index index of the element text, it does not satisfy the predicate and can be skipped. Similarly in case of inequality operator, the element can be skipped if its index is less than or greater than the index of the constant depending on the type of the inequality operator. An example of the *Hash* attribute can be seen at line 2 in Figure 4. The range of the numerical values occurring at each label is stored as a list along with the label tag.

For queries with descendant axes, the XHint Manager requires additional information about the descendants of elements in order to identify irrelevant data. If the system knows the labels of the descendants of each element, it can avoid parsing the elements that do not contain the descendant label of the query.

XHints provide this information about the descendant 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 on. The bitmap is stored as an integer value of the attribute *desc* of the XHint element. The mapping from label to index is stored as the value of attribute *LIndex* of a separate XML element called META used to store meta-information about XHints. It is a simple list of label and the bitmap index stored as a string as shown in line 2 in Figure 4.

4. XHint Processing

We now describe how are XHints used by the XHint Manager to make processing more efficient. The parser generates SAX events for the data and sends them to the XHint Manager for processing. The XHint Manager handles these events in two possible ways. If the SAX event corresponds to a data element, it is forwarded to the query engine otherwise, if the event is generated by a XHint, it is processed by the manager itself. The pseudo code for the SAX functions of XHint Manager are shown in Algorithm 1.

The XHint Manager use the list of *interesting* SAX events called EventList to process XHints. An *interesting* SAX event is informally defined as an event which may contain or determine the query result and has to be processed by the query engine. The XHint Manager assumes that these events are identified *a priori* and updated in the EventList by the query engine as the data is processed.

Every event in the EventList is associated with a tag label of an element the query engine is interested

in. Further, it also stores whether the element can occur as a descendant or an immediate child of the current element being parsed by the system, depending on closure axis in the query expression. It may also contain an predicate associated with it which has to be satisfied.

If the query contains only child axes without any predicates, the interesting events for the query engine correspond to child element labels. Thus, the relevant elements which need to be processed by the query engine are child elements with the label of the event present in the EventList. During the processing of a XHint, XHint Manager can use the child hint to obtain these relevant offsets. The offsets allow the parser to jump directly to these elements skipping the remaining elements. 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 example data shown in Figure 4. The result of the query consists of the *title* elements at lines $6-8$ and $22-24$ of the original XML data (Figure 1). A normal XPath query engine has to parse and process the entire data before obtaining the query result. A large portion of the computational resources used by it is spent on generating and handling SAX event for the elements that are not part of the output leading to a low throughput.

Fig 4 contains the same example dataset with XHints inserted in it. A query engine can use these XHints to reduce the query processing cost in the following manner. At the start of the data, the query engine registers the SAX event corresponding to a *book* child element with the XHint Manager. When the XHint Manager processes the XHint at line 3, the offsets related to the *book* child is used by XHint Manager to directly skip to the first *book* element at line 7.

The handling of the SAX event for the *book* element is delegated to the query engine. Since the interesting element inside an *book* element is an *title* element, the query engine on processing the *book* element removes *book* from the event list of the XHint Manager and adds *title* to it.

The next XML element to be parsed is the XHint at line 8, which is handled by the XHint Manager internally. As the event list now contains the *title* event, the manager uses the child hint for *title* to skip directly to line 9. After the query processor outputs the *title* element, XHint Manager requests the parser to jump to the end of the *book* element at line 22 since there are no more interesting SAX events i.e. *title* elements). The offset to the end of the *book* tag is calculated using the end hint of the XHint at line 8.

When the query engine parses the end tag of *book* element, it again updates the XHint Manager's event list by removing *title* and adding the *book* event to it. The XHint Manger then processes the second *book* element in a similar fashion.

This scheme allows the parser to process only 6 elements compared to 20 elements processed by a normal query engine. 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 indexes.

■

In case of queries with predicates, an element is only relevant if the predicate associated with it is satisfied. The query engine stores the predicate along with the other details about the SAX event in the EventList. XHint Manager uses the information about the predicate along with the data digest to select the relevant offsets. If a particular element does not satisfy an associated predicate, XHint Manager can avoid parsing the remaining element.

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. In case the XHint does not contain the child hint, XHint Manager can infer that the element is not relevant and skip it.

Example 2 Consider the query */book[discount]/title/text()* on the data in Figure 4. 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 the 20 elements. Note that an XHint contains child hints for all the child elements of a parent element. 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 *discount* element, the parser can skip parsing the remaining element.

The query engine can register an "interesting" events with the XHint Manager with the element tag

label as *title* and a 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 XHint Manager 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*), XHint Manager can infer that this element satisfies the predicate and thus, use the offsets from the child hint for *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 an element *discount* allows the XHint Manager to ignore the *title* child hint and skip directly to end of the *book* element since it does not satisfy the existential predicate.

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 is complex and involves an comparison operator, the XHint Manager uses the data digest stored in the child hint to reduce the number of elements parsed in order to evaluate the predicate. The XHint Manager 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 which cannot satisfy the predicate. It avoids parsing such elements by skipping directly to the remaining elements.

Note that although a mismatch 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 has to parse the element in order to correctly evaluate the predicate.

Example 3 Consider the query */book[author="R. Bazea-Yates"]/title/text()* on the example data in Figure 4. 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 XHint Manager 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 XHint Manager. The XHint of the element contains the first three characters of the text in addition to the offsets to the three *author* elements. The XHint Manager uses this digest to evaluate the predicate *a priori*. In this case, since the descendant digest of any of the

```
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=''329''
    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 Wesely</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

three elements does not match the first three character of the constant in the predicate, XHint Manager requests the parser to skip all the child elements and directly go to the end tag of *book* element at line 33.

But note that matching of the two descendant digest does not guarantee that the predicate will be satisfied by the element. For example, if the constant in the predicate was "Jeff Ullman" instead of "R. Bazea-Yates," the descendant digest for the second *author* element at line 31 matches with the descendant digest of the constant though the predicate is not satisfied. ∎

The descendant hint present in the XHint is used for queries with descendant axis. For such queries, an interesting event can correspond to either a descendant or a child. In this case, XHint Manager uses the descendant hint of the XHint to determine if the particular tag label occurs as the descendant of the current element. If it does, the element can occur as a child of any of the complex child elements (ones with their own child elements) and the XHint Manager stores offset to all such child elements.

Example 4 Consider the query *//address* on the data shown in Figure 4. 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 0th bit of the bitmap in the descendant hint is set on.

The first bit in the descendant bitmap is set for the XHint of the *root* tag indicating that it contains at least one *address* label 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 process 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 and skips all the child elements of the first *book* element except *pub* that contains the *address* element.

In case of the second *book* element at line 23, the 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 can jumps directly to the end of the element at line 33.

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

[**Ignore the pseudo-code right now. It is incomplete**]

Algorithm 1 XHint Processing

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

procedure endElement (SAXEvent e)
 1: if SAXEvent E in EventList then
 2:    processXHint(e);
 3: else
 4:    QueryEngine.endElement(e);
 5: end if
 6: parser.skipData(OffsetStack.pop());

procedure processXHint(e)
 1: for all Events E in EventList do
 2:    if E is a child Event with label L then
 3:       if E has an existential predicate with label L' then
 4:          if XHint has a child hint for label L' then
 5:             OffsetStack.add(e.getChildHint(L);
 6:          end if
 7:       else if E has an comparison predicate with label L'
          then
 8:
 9:       end if
10:    else if E is a descendant Event with label L then
11:       OffsetStack.add(e.getComplexChild());
12:    else if E is a predicate event with value v then
13:       if v is null then
14:
15:       end if
16:    end if
17: end for
```

Figure 1 provides the pseudo-code for the XHint processing algorithm.

5. XHint Generation

[**Probably a subsection**]

Since XHints contain offset information about the child and descendant nodes of an element, the child elements have to be processed before the parent element. Typically, a DOM tree of the XML data can be generated in the memory and processed in a bottom-up fashion to generate the XHints. However, DOM trees require the entire data and are not suitable for unbounded streaming data. Moreover, this scheme requires pre-processing of the data and is not applicable apply in scenarios which require real-time generation of XHints.

The alternative is to parse the stream and generate the hints on-the-fly. The XHint Generator uses a fixed-size buffer to parse and store information about

the XML elements in memory. When the parser reaches the end of an element, the generator uses the information about the child and descendant nodes to generate XHint for the element. These XHints are stored in the memory along with the element. When the generator reaches its buffer limit, it inserts the XHints at appropriate places in the data and outputs it.

Note that since we use fixed sized buffer, the XHint generator may not read complete elements before the buffer size limit is reached. In this case, a XHint for an incomplete element can only contain information about the portion of element processed until now. Thus, instead of the offset to the end of the node, the *end* hint contains the offset to the last processed child node of the incomplete element. When the next data chunk is processed, the XHint generator inserts a XHint at the end of the incomplete data node. This XHint is used to store information about the remaining portion of the parent element. We also insert a new META element containing meta-information about the XHints at the start of the data chunk.

For each data chunk, a bottom to up approach is followed to generate the various offsets stored in the XHints. The offsets and descendant digest of child elements are generated after each child node is processed. Once the entire parent element is parsed, these offsets and descendant digest are inserted in the XHint. The length of the element is also computed, which includes the length of the child elements along with their XHints.

However, XHints of all elements are not useful for the query processor. For example, XHint does not save on any SAX event for elements with no child element. Instead, processing of XHints for such elements results in an overhead. In order to avoid this overhead, we only insert XHints for elements containing more than one child element in the stream.

6. Application of XHints

XHints require the query engine to identify the *interesting* SAX events for the query and update the EventList as the data is processed. Although the exact mechanism the query engine performs this update depends on the architecture of the engine, we use two query systems, XSQ and Tukwila, based on different architectures to outline the process and demonstrate the generic appliacability of XHints.

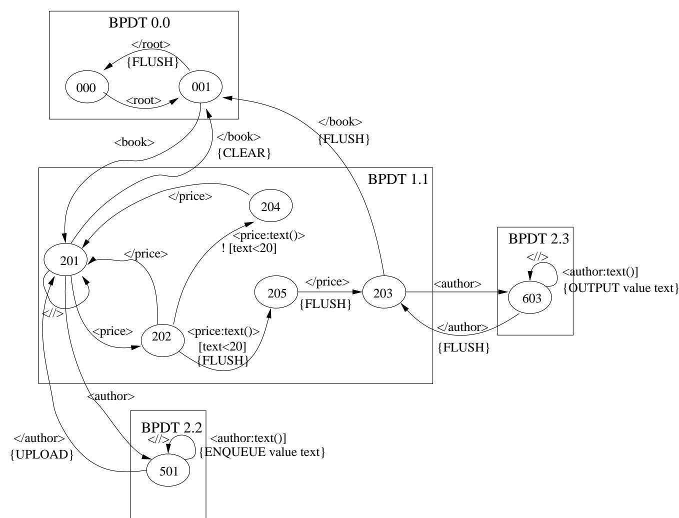


Figure 4: HPDT for /book[price ¡ 20]//author

6.1 XHints and XSQ

[**Have to work on this section**] XSQ is an automaton based streaming XML query processor which can evaluate a broad range of XPath expressions. It constructs a hierarchical automaton called HPDT from smaller finite state machines called BPDTs. Each BPDT corresponds to a location step in the XPath query expression and has a buffer which is used to store potential query results.

The arcs between the states of HPDT are associated with element labels and actions. If a SAX event matches the label associated with an arc, the HPDT makes a transition along the arc and executes the corresponding action. Figure 6.1 shows the HPDT for the query */book[price¡20]//author*.

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 a simple mechanism to identify the *interesting* events using the current states of HPDT. The *interesting* events are only those events which result in any transition in the HPDT. They are easily identified using the labels of the arcs from the set of current states.

XSQ-H is a modified version of XSQ which uses the HPDT to identify the *interesting* SAX events and updates them in the EventList of the XHint Manager.

Example 5 *Consider the query* /book[price¡20]//author *on the XML data of Figure 4. The HPDT for the query is shown in Figure 6.1. Initially, the set of current state is {001}. The arcs from this set of states correspond to the end tag of* root *tag and the start tag of* book *element. Thus, the* interesting *events are the end of the* root *element and the start of* book *element. The XHint Manager processes the XHint at line* 2 *to obtain the offsets to these two SAX events. 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 arcs with label* author *and* price*. The closure axes of the* author *label is identified by the arc with // label in the HPDT from the state* 201*. The predicate constant and the operator associated with* price *element are stored in the arcs from state* 202*.*

XSQ-H can use this information to provide the XHint Manager with correct interesting *events. Since the*

6.2 XHints and Tukwila

Tukwila [13] is an iterator-based query engine capable of evaluating XQuery expressions on streaming XML data. The Tukwila engine processes XQuery expressions in a manner very similar to how queries are handled in relational databases. The query optimizer uses basic operators to construct and optimize a plan for the query which is passed to the execution engine. Figure 6.2 shows an example XQuery and the corresponding query plan.

The execution plan uses a special operator called X-scan which is responsible for reading, parsing and matching the XML data with the regular expressions in the query. It assigns appropriate binding values 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 X-scan operator consists of a series of finite state machines (FSMs) which are matched against the XML data to produce the bindings for the XQuery variables. It converts all the XPath expressions (which are a restricted form of regular expressions) in the XQuery into state machines. Figure 6.2 shows the state machines for XPath expressions in the example XQuery. 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 of the variable associated with

```
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 5: Example XQuery

it. The machine then activates the dependent machines, which remain active while X-scan is scanning the value of binding.

In absence of any prior information of the XML data, X-scan operator has to parse every element in the stream. XHints can be used to avoid this extra overhead cost by replacing X-scan operator with an XHint compatible operator called XH-scan.

The XH-scan operator uses the state of the FSMs to identify the *interesting* 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 machine makes a transition to a new state, the label on the arc from the new state corresponds to an *interesting* SAX event.

Some of the transitions defined in the state machines may correspond to an predicate evaluation which is done by a selection operator in the query plan. In order to allow the XHint Manager pre-evaluate the predicate, XH-scan can obtain the information about the predicates from the selection operators using simple query plan rewriting rules. The*interesting* SAX events are registered with the XHint Manager which uses XHints to skip other irrelevant elements.

Example 6 *Consider the execution of the sample XQuery shown in Figure 6.2 on the streaming XML data of Figure 4. The state machines representing the XPath expression are shown in Figure 6.2. The processing of XHints by these state machines is very similar to the processing done by XSQ-H. Since both are essentially automatons, the interesting SAX events are defined by the labels on the arcs from the current state. Additional information about these SAX events such as the type of axes (child or descendant), predicates can be stored along with the label on the arcs as in XSQ-H.*

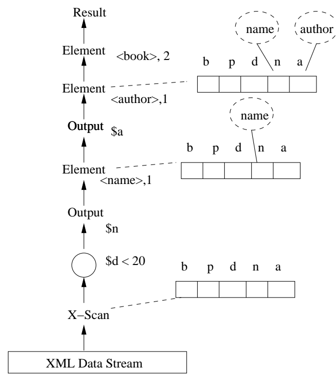


Figure 6: Query Plan for the Example XQuery

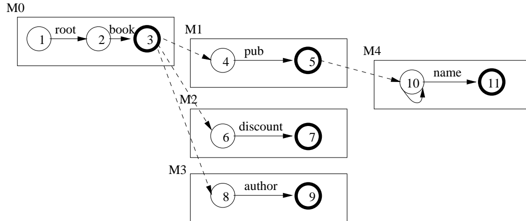


Figure 7: State Machines for the Example XQuery

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 top most* root *element, it reaches state* 2. *This state has an arc with the label book which also corresponds to the interesting SAX event. The XHint at line* 2 *provide the offsets to the two* book *elements in the data which can be used to avoid parsing the* mag *element. When the first* book *element is parsed, 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 interesting events correspond to the labels on the arcs from the current states of the activated machines. The arc of* M_2 *also contains the information (due to query plan rewriting) that this SAX event is required for a predicate evaluation and XH-scan accordingly registers the event by using the XHint Manager API function with appropriate parameters.*

7. Experimental Results

We implemented a prototype of XSQ-H using Java 1.4 and used it to conduct an experimental study to evaluate the performance of XHints. Xerces 2.4.0 was used as the XML parser for XSQ-H. It was modified to support data skipping.

We measured the throughput of XSQ-H for different kinds of XHints and compared it with other systems which do not use XHints for query processing. We also conducted experiments to study the effect of query characteristics on the throughput gain. Furthermore, we investigated the effect of the buffer capacity in XHint generation phase on the throughput gain of the system. Finally, the overhead cost of generating XHints for streaming XML data was measured.

7.1 Experimental Setup

We conducted the experiments on a PC-class machine with an Intel Pentium III processor with 1 GB of main memory running the Red Hat 7.2 distribution of GNU/Linux (kernel 2.4.9-34). The maximum amount of memory available to Java Virtual Machine was set to 512 MB.

We used three real test datasets for our experiments. The characteristics of the datasets are provided in Table 1.

7.2 Throughput

In the first set of experiments, we investigated the throughput of the query system for sample queries on the test datasets. We measured the performance gain achieved by XSQ-H on data with different types of XHints. Four kinds of XHints were used to evaluate the performance of the system; 1) XHints generated offline without descendant hint (XHint-NS), 2) XHints generated in a streaming fashion 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 fashion (XHint-SB).

In order to benchmark the performance of the various type of XHints, we compared the performance of XSQ-H with systems processing data without XHints. In addition to XSQ, we chose XMLTK [2], a streaming query engine implemented in C++ for the performance comparison. However as XMLTK does not support query with predicates, we only present results for XSQ and XSQ-H for such queries.

We measured the throughput of the systems for 14 sample queries on each of the three test datasets. The results for the SwissProt dataset are shown in Figures 8 and 9.

For simple queries such as Q2 and Q5 in Figure 8, XSQ-H performs better than XSQ for all four types

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

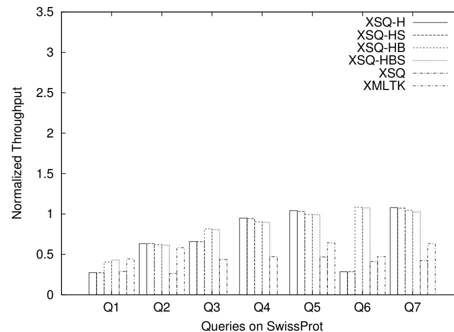
of XHints. However, XHint-NS and XHint-S perform marginally better than their counterpart containing the descendant hint. It is expected since XSQ-H does not use the descendant hint for processing such queries and the additional data overhead in case of XHint-NSB and XHint-SB result in a slight performance degradation.

But the benefit of the descendant bitmap can be observed for closure containing queries such as Q1 and Q6 in Figure 8. For such queries, XHint-NS and XHint-S do not provide sufficient information for XSQ-H to skip substantial amount of data and the additional cost of parsing XHints lowers its throughput. This information is provided in form of the descendant bitmap by XHint-NSB and XHint-SB which allow the query processor to reduce the parsing cost by a large margin. The benefit of the descendant bitmap is particularly large for Q7 in Figure 9. In case of XHint-NSB and XSQ-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 data digest present in XHint-SB and XHint-NSB improve the throughput of XSQ-H for queries predicates such as Q3 in Figure 8 and Q2 in Figure 9. The pre-evaluation of the predicate allows parser to skip more data in case of XHint-NSB and XHint-SB and provide an higher throughput.

XSQ-H performs better than XMLTK for most of the queries such as Q5, Q6 and Q7 in Figure 8 but has a lower throughput than XMLTK for queries such as Q1 in Figure 9. This query has a very low throughput in case of XSQ-H because the query result contains the entire data stream. As a result, the XHints do not provide any benefit and are instead an overhead on the system.

The throughput for the systems for the sample queries on the DBLP dataset are shown in Figures 10 and 11. As with the SwissProt dataset, XSQ-H outperforms XSQ by a significant margin for the sample queries. However, we can observe small difference

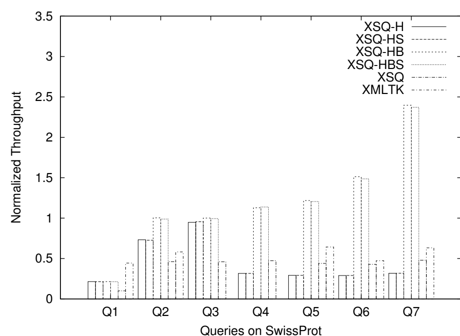


Q1:`//Author`
Q2:`/Entry/Features`
Q3:`/Entry[Org=Muridae]/Ref[Medline=9225337]/Cite/text()`
Q4:`/Entry/Ref[MedlineID=9225337]/Cite/text()`
Q5:`/Entry/Ref/Cite/text()`
Q6:`//Entry/Features//DOMAIN//Descr/text()`
Q7:`/Entry/Mod`

Figure 8: Normalized Throughput for different queries on SwissProt

in the performance of XSQ-H for different kinds of XHints in case of simple queries such as Q6 and Q7 in Figure 11. XSQ-NS has the highest throughput out of all the systems followed by XSQ-S, XSQ-NSB and XSQ-SB in that order. The offline generation of XHints allow faster processing of XHints compared to the on-the-fly generation of XHints in a streaming fashion. This difference in the throughput is expected as offline generation of XHints allow XHint to store information about the complete data instead of only a part of it. However, the degradation in the performance in case of on-the-fly generation of XHints is very small and is an acceptable trade-off for a pure streaming system.

The descendant hint in XHint-NSB and XHint-SB are responsible for extra 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 can be justified by the performance gain provided by



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 9: Normalized Throughput for different queries on SwissProt

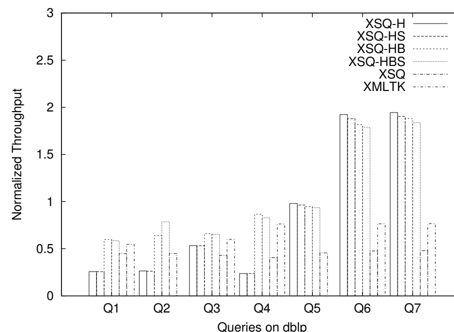
the descendant bitmap in case of queries containing closure as can be seen in case of Q1 and Q4 in Figure 10.

We compared the throughput of XHints with different systems. But, in some sense, it is not fair to compare the performance gain achieved by BPDT based systems like XSQ by using XHints with simpler XPath query engines such as XMLTK due to architectural and implementation differences. XMLTK uses a simple DFA without any buffering to evaluate the query. On the other hand, since XSQ support a wider range of XPath queries, they use buffering and additional computational checks which may not be useful for simpler queries but reduce the processing speed.

An alternative metric that can be used to compare the performance of different systems is the number of SAX events processed. It is reasonable to assume that if two systems have same architecture and backend processing power, the system processing the lesser number of SAX events will perform better.

We measured the number of SAX events generated by the different systems on the three datasets. As both XSQ and XMLTK do not skip any data, they process the same number of SAX events for all queries. XSQ-H used the XHints to skip different number of SAX events depending on the query and the type of XHints available in the data stream.

The number of SAX events for the sample queries on the SwissProt database are shown in Figures 12 and 13. XHints result in a significant reduction in the



Q1:`//ee/text()`

Q2:`//editor/text()`

Q3:`/inproceedings[author]/title/text()`

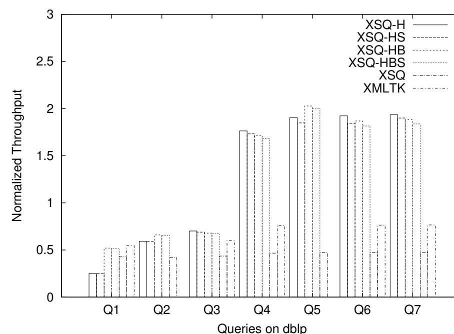
Q4:`//article[year=1997]//cdrom/text()`

Q5:`/article/title/text()`

Q6:`/phdthesis/school/text()`

Q7:`/mastersthesis[url]/title/text()`

Figure 10: Normalized Throughput for different queries on DBLP



Q1:`//url/text()`

Q2:`/inproceedings[url]/title/text()`

Q3:`/inproceedings/booktitle/text()`

Q4:`/proceedings/title/text()`

Q5:`/phdthesis[year=1993]/title/text()`

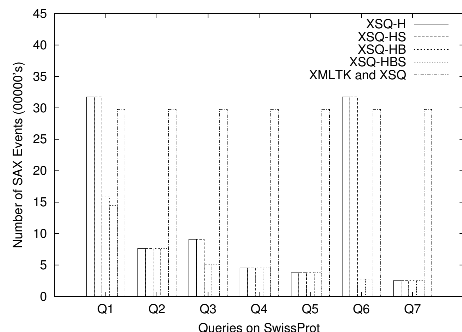
Q6:`/phdthesis/title/text()`

Q7:`/mastersthesis/title/text()`

Figure 11: Normalized Throughput for different queries on DBLP

number of SAX events generated by the parser. As expected, XHint-SB and XHint-NSB provide a larger reduction in the number of SAX events for queries with closures than XHint-NS and XHint-S due to the descendant hint.

The data digest also reduces the number of SAX events generated by the parser as it can be seen for Q3 in Figure 12. The reduction in the number of SAX events is reflected in the increase in the throughput of the system supporting our thesis that SAX event generation and processing constitutes a major portion of query processing.



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

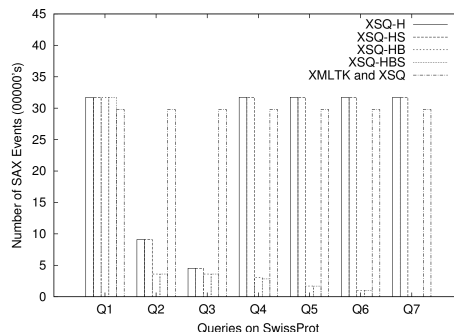
Figure 12: SAX Events Processed for different queries on SwissProt

Figures 14 and 15 show the number of SAX events processed by the query engine on the DBLP dataset. The reduction in the number of SAX events is more

7.3 Query Characterstic

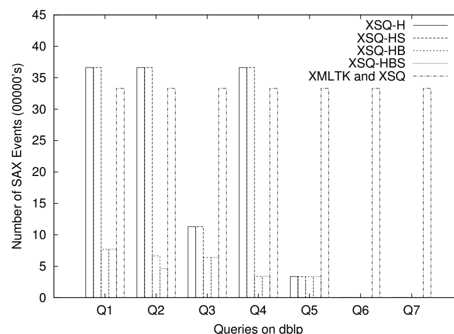
As it can be seen from the throughput results for various sample queries on the test datasets, XSQ-H provides a better throughput than XSQ and XMLTK in most cases. However, the actual gain in the throughput varies significantly and depends on the query. We conducted experiments to observe the effect of the various query characterstics on the throughput gain achieved by XSQ-H.

The length of a query is defined as the number of location steps in the expresions and is an important characterstic. We ran four queries with different length on the SwissProt dataset. It can be observed



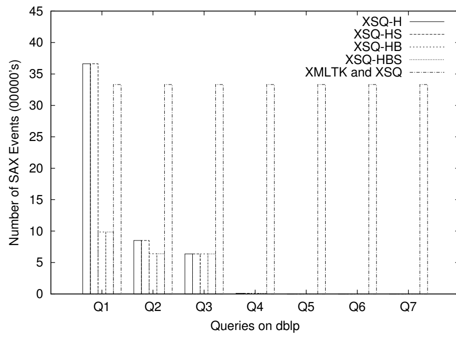
Q1:/Entry
Q2:/Entry[Org]/Ref[MedlineID]/Cite/text()
Q3:/Entry/Ref[MedlineID]/Cite/text()
Q4://CARBOHYD/text()
Q5://Entry[Org=Eukaryota]//MUTAGEN
Q6:/Entry[Org=DISULFID]//Author/text()
Q7://Noresult

Figure 13: SAX Events Processed for different queries on SwissProt



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 14: SAX Events generated for different queries on DBLP



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 15: SAX Events generated for different queries on DBLP

from Figure 16 that the throughput of XSQ-H increases with the length of the query. Longer queries usually have smaller query results and allow XSQ-H skip larger amount of data resulting in a higher throughput.

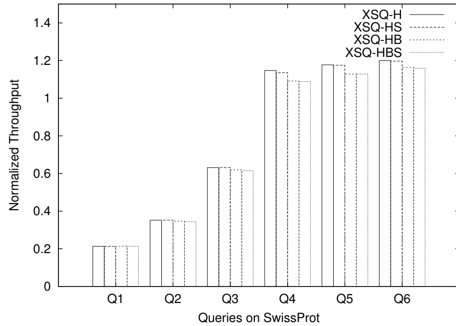


Figure 16: Normalized Throughput for queries with different length

Q1:`/Entry`

Q2:`/Entry/text()`

Q3:`/Entry/Features`

Q4:`/Entry/Features/DOMAIN`

Q5:`/Entry/Features/DOMAIN/Descr/text()`

The throughput of XSQ-H also greatly depends on the presence of descendant axis in the query expression. We used a set of queries different in the number and position of descendant axis on the SwissProt dataset to study this effect. The queries and the

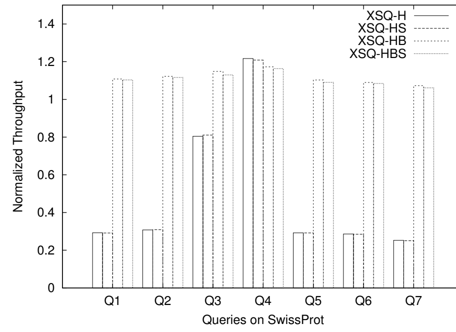


Figure 17: Normalized Throughput for queries with closures on SwissProt

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()`

throughput of XSQ-H on the queries are shown in Figure 17.

As expected, XHint-S and XHint-NS perform very poorly on all queries except Q3 and Q4. In case of these two queries, the descendant axis is present deep in the query expression reducing the overhead incurred due to absence of information about descendants. On the other hand, the throughput of XHint-SB and XHint-NSB is consistently high. It is slightly higher for queries with 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 as in Q1 and Q5.

We also studied how presence of multiple predicates in the query effect the throughput of XSQ-H. Figure 18 presents the throughput of the system for the four type of XHint schemes on sample queries with predicates. The data digest present in the XHint-SB and XHint-NSB allow XSQ-H to pre-evaluate the predicates and reduce the number of SAX events. As a result, these two XHint schemes have a higher throughput compared to XHint-S and XHint-NS. However, XHint-SB and XHint-NSB do not outperform the other two schemes in case of Q1. In case of this query, the number of SAX events skipped using the data digest is relatively very small since the label in the predicate does not occur frequently in the dataset. Instead, the overhead due to extra data processing in XHint-NSB and XHint-SB

result in performance degradation.

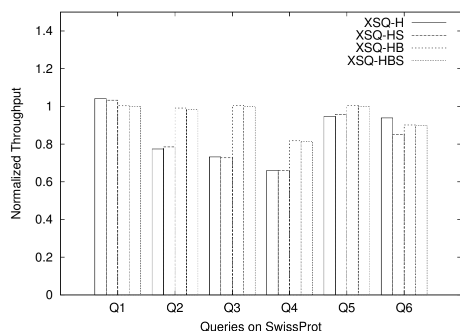


Figure 18: Normalized Throughput for queries with predicates

Q1:`/Entry[DISULFID]/Reference/Author/text()`

Q2:`/Entry[Org=Eurkaryota]/Reference/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=9225337]/Cite/text()`

Q7:`/Entry/Ref[MedlineID]/Cite/text()`

The information contained in a XHint depends on the size of the buffer used to store the data during the XHint generation phase. A larger buffer can allow the XHint to store additional information about the data allowing the XHint Manager to skip more data. We generated XHints for SwissProt dataset with different buffer size and measured the throughput of four different queries to study how does the efficacy of XHint vary with the buffer size. As Figure 19 shows, the throughput of XSQ-H remains drops sharply when we reduce the buffer size below approximately 10KB. The throughput only increases marginally if we increase the data size beyond 20-30 KB indicating that XHints generated using smaller buffer size of a few KBs are almost as efficient as large buffer size. [**In the actual experiments, we used buffer size ranging from 1KB to 6MB of raw data. I have only plotted from 0K to 100K in order to show the knee point more clearly since the throughput is almost constant for any buffer size beyond a few KBs**]

As we observqed before, there is a correlation between the throughput achieved by XSQ-HB and the portion of data it processes in number of SAX events. We use a metric called *selectivity* defined as the ratio of the number of SAX events in the query result to the total number of SAX events to study the correlation.

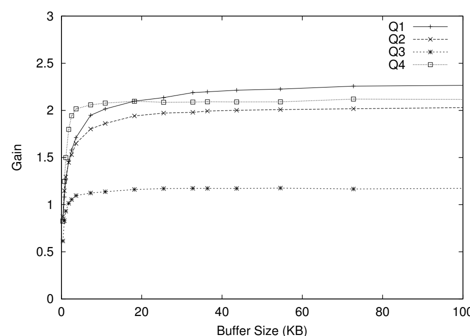


Figure 19: Normalized Query Throughput for different buffer size

Q1:`/Entry/Features/DOMAIN/Descr/text()`

Q2:`/Entry[Org]/Ref[MedlineID]/Cite/text()`

Q3:`/Entry//Descr/text()`

Q4:`//Entry[Org]/Descr//text()`

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

In order to measure the effect of selectivity of the queries on the throughput of the system, 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 four queries (Table 2) of varying complexity on each of the datasets and measured the throughput for different values of the selectivity.

Figure 20 displays the throughput gain of XSQ-HB compared to XSQ and XMTLK for different values of selectivity. Throughput gain of XSQ-HB compared to other system is defined as the ratio of the throughput of XSQ-HB and the throughput of the system. As XMLTK does not support predicates, XSQ-HB is compared with XMLTK for only the first two queries.

It can be observed that XSQ-HB provides a speedup in processing of the data for a wide range of selectivity. As expected, XSQ-HB provides a high throughput gain for low selectivity compared to XSQ and XMLTK. The performance worsens for high selectivity as the processor cannot skip sufficient elements.

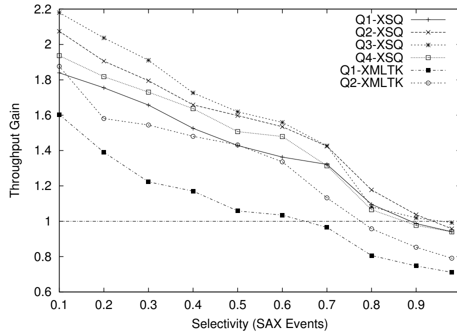


Figure 20: Effect of Query Selectivity

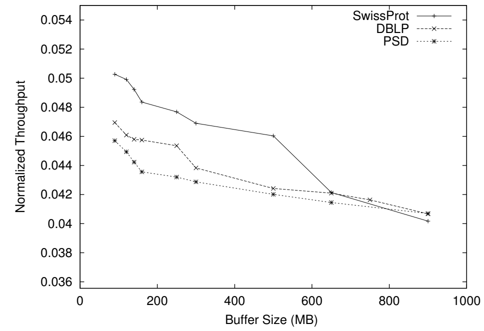


Figure 21: Throughput of XHint Generation

7.4 XHint Generation

We conducted experiments to evaluate the scalability and feasibility of the streaming XHint generation algorithm. One of the important parameters in XHint generation is the size of the buffer allotted to the system. We generated XHints for the test datasets for different values of buffer size and measured the time taken to process the entire dataset. The result is displayed as throughput of the XHint generation system in Figure 21. The XHint generator has to compute and handle greater amount of data if its buffer size is large. If we use smaller amount of buffer size, the computation of the offsets in the XHints are done faster. As a result the throughput falls with increase in the buffersize. We also show the actual time taken to generate hints for two of the test datasets in Figure 22 to provide a different perspective.

The insertion of XHints in the XML data results in an increase in the data size that has to be sent to the query processor. We measured this overhead in terms of the percentage increase in the data due to addition of XHints for datasets of different sizes. As Figure 23 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.

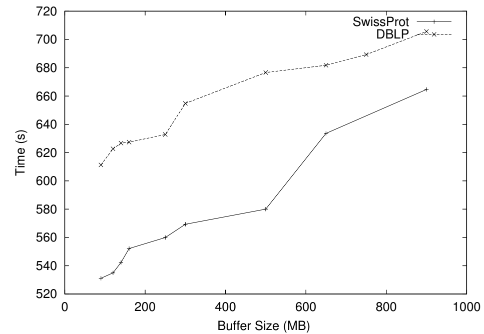


Figure 22: Time taken for XHint Generation

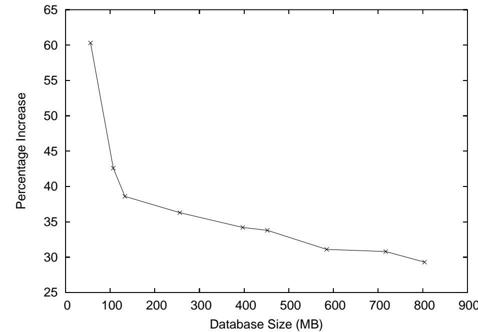


Figure 23: Percentage increase in the data size for buffer size of 50 MB

8. Related Work

A large number of techniques have been proposed in recent years to make query processing efficient on streaming semistructured data. The idea of inserting punctuations in a data stream to assist query processing was first introduced in [19]. The punctuation were in the form of predicates and allowed the query processor to infer the absence of certain elements in the data following the punctuation. A binary encoded index called *SIX* has been used to make processing faster for simple queries in [11]. SIX stores the offsets to the start and end of the elements in the data stream. The query processor can use the offsets to skip processing data in much the same way as XSQ-HB. The MatchMaker system [14] addresses a similar problem of matching an incoming data stream to a large number of queries by using indexes on the query patterns. This problem is dual to the conventional query processing problem in that the size of the data is small compared to the number of queries.

Several query engines have been presented for streaming XML data. The XML Streaming Machine (XSM) [16] decomposes the queries into simpler subexpressions and uses a chaining method to process the subexpressions individually. XSQ [18] and XPush [12] use an automaton based approach to process streaming XML data. XSQ constructs an hierarichal automaton called HPDT from the queries. On the other hand, XPush uses a lazy deterministic finite automaton to process the queries.

A lot of work has been done for non-streaming databases. Dataguides [10] were one of earliest framework designed to provide a structural summary of semistructured data. Template Indexes or T-Indexes [17] and Index Fabric [7] are based upon generating indexes on data paths, which are matched to the query to obtain the offsets to relevant elements. The XML Indexing and Storage System (XISS) [15] employs a numbering scheme to index elements and attributes.

An adaptive indexing scheme for non-streaming XML data is presented in APEX [6]. 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 a 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 in [20]. It represents XML database and the query as structure-encoded sequences reducing the problem to that of matching subsequences. Unlike other indexes, it processes the query as whole without decomposing it into sub-queries saving on expensive join operations required to merge the sub-query results. An adaptive version of the T-Indexes [17] called D(k)-Indexes is proposed in [5]. D(k)-indexes provide an updating mechanism storing only the most useful path indexes depending on the query workload.

A number of systems have been developed to address a closely related problem of *filtering* XML documents based on XPath queries. *Index-filters* [3] use an idea very similar to XHints to skip irrelevant data to process data efficiently. It constructs indexes over the tags of the document in order to identify the portions of the data that are guaranteed not to match the query and does not parse them. *XFilter* [1] and *YFilter* [9, 8] construct finite automaton machines from multiple queries to perform the filtering operation.*XTrie* was proposed in [4] to index the XPath queries based on common subexpressions.

9. Conclusion

XML parsing is responsible for a substantial portion of query processing time. A query engine can significantly improve its throughput if it can skip elements which do not belong to the query result. We proposed an indexing scheme that allows the query engine to skip large portions of irrelevant data improving its processing speed. The index uses special XML elements called XHints, interleaved with the data, to store structural information about the data elements. This information is used by the query engine to identify the elements that can be safely skipped and reduce the overhead of parsing.

We illustrated how four types of offset information or hints can be used by an query processor to improve the throughput for a large variety of simple and complex queries.

We described XSQ-H, a XHint-enabled version of XSQ, a streaming XML query engine as an concrete application of XHints. In order to demonstrate the genericity of the approach, we gave a brief outline of how XHints can also be used in an iterator-based model such as Tukwila to process queries more efficiently.

Finally, we evaluated the benefits of XHints by running several experiments on a prototype of XSQ-H using test datasets. We also conducted comprehensive expriments to measure the overhead cost of generating and inserting XHints in different datasets.

References

[1] Mehmet Altinel and Michael 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] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *Proceedings of Programming Language Technologies for XML (PLAN-X)*, October 2002.

[3] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation vs. index-based XML multi-query processing. In *Proceedings of the International Conference on Data Engineering*, March 2003. To appear.

[4] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering*, pages 235–244, February 2002.

[5] Qun Chen, Andrew Lim, and Kian Win 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] Chin-Wan Chung, Jun-Ki Min, and Kyuseok 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] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe 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] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28(4), December 2003. To appear.

[9] Yanlei Diao, Peter Fischer, and Michael J. Franklin. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the International Conference on Data Engineering*, pages 341–344, February 2002.

[10] Roy Goldman and Jennifer 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] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *Proceedings of the International Conference on Database Theory*, pages 173–189, January 2003.

[12] Ashish Gupta and Dan 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.

[13] Zachary Ives, Alon Halevy, and Dan Weld. An XML query engine for network-bound data. In *The VLDB Journal*, 2003.

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

[15] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[16] Bertram Ludascher, Pratik Mukhopadhayn, and Yannis Papakonstantinou. A transducer-based XML query processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 227–238, August 2002.

[17] Tova Milo and Dan Suciu. Index structures for path expression. In *Proceedings of the International Conference on Database Theory*, pages 277–295, January 1999.

[18] Feng Peng and Sudarshan 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.

[19] Pete Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Punctuating continous data streams. Technical report, OGI School of Science and Engineering at OHSU, 1999.

[20] Haxiun Wang, Shaghyun Park, Wei Fan, and Philip 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.