Chapter 5

Exploiting Query Redundancy

In this chapter, we present a new XPath [78] processor. Chapter 4 detailed how to develop multicast applications on Appcast. Appcast uses XML [80] to exchange data and SOAP [18-19] as middleware. XPath, short for XML Path language, is a querying language used to select specific parts of very large XML documents. Many algorithms have been proposed for processing multiple XPath expressions in the context of streaming XML documents such as Xfilter[82], Yfilter[83], IndexFilter[85], Xtrie[86] and CQMC[84]. But these algorithms do not handle backward axes like parent and ancestor. XAOS[79] deals with XPath expressions with forward and backward axes. But XAOS handles only one query at a time. As a result a document is parsed \( q \) times for a set of \( q \) queries. In case of large documents, this parsing time goes out of bounds. Moreover, research suggests that there is bound to be significant commonality among different queries in a large-scale system. By exploiting this commonality we can reduce a lot of processing and runtime memory consumption. In this chapter we present a method called YALXP (Yet Another Light Weight XPath Processor), to evaluate multiple queries, with both forward and backward axes, in a single pass exploiting commonality among the queries. YALXP is built upon XAOS algorithm. We make the following contributions in this approach.

1. A concise representation of all XPath expressions, with out loss of information, called combined x-dag, where all backward constraints in all the queries are converted into forward constraints.
2. Procedure to create combined x-dag.
3. Data structure called combined matching structure that represents all the matches of all XPath expressions in the document.
4. Method to evaluate Xpath queries using the above structures.

Section 1 introduces XML and its related processing languages like XPath, parser like SAX [104] and DOM [91]. Section 2 surveys the algorithms proposed for efficient XPath processing. Section 3 presents YALXP – Yet Another Light Weight XPath Processor.
proposed by us. Section 4 simulates YALXP, XAOS [79] and compares the results. Section 5 concludes this chapter.

5.1 Introduction

Search tools till date have insufficient capabilities to keep pace with the information generated. Particularly getting right and relevant information has become a nightmare. In this scenario, SDI - Selective Data Information Dissemination, a popular concept used by libraries is gaining importance. In SDI, users register with servers that are nearer to them with their interests - profile. Based on the profiles, the servers filter right information and push the same to the user. Definitely, a centralized server cannot scale up to the requirements of large number of users spread all over Internet.

Figure 5.1 Overlay

Overlays can help meet the requirement of distributing information to a large user base on Internet, however, we have to devise ways to reduce the latency/delay introduced by overlays. Figure 5.1 shows as an example of overlay. Stream processing (explained below in detail) allows data to be processed as it streams in thereby not adding any delay at processor side. Using overlays, we can register profiles with the servers nearer to the consumers and these servers in turn send the combined profiles to the servers above them. This profile grouping and sending can further go up the tree till the producer. Now, when the producer sends info, this has to be filtered as per their profiles at each server at lower levels of the tree, till it reaches the consumer. Data is exchanged in the form of XML[80] in SDI systems and each profile is represented in the form of an Xpath[78] query and we filter the information based on multiple profiles (queries), registered at each
server. In following sub sections we deal with the fundamental concepts, notations, functions etc of XML and its related technologies like XPath, DOM, SAX etc.

5.1.1 XML (Extensible Markup Language)
XML [80] is a simple and standard way to describe structured data. Markup is a way of conveying metadata (that is, information about a dataset). A collection of this markup that conforms to a defined syntax and grammar may be called a language. Markup languages use string literals or tags to delimit and describe data. XML is extensible because it involves a standard mechanism for defining new tags and their usage. The XML markup describes the contents of the document, both through explicit descriptions (the tag names and attributes) and through implicit structure (how the tags are nested within one another). XML was developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

Nature of XML
XML is a text-based format, making it a great solution for exchanging information across platforms. In addition, XML can be extended to meet the needs. The extension mechanism being standard, can be easily described to anyone-programmer or machine who reads the data. A well-formed XML document can be used by any XML-enabled system to provide the required information or transformations to the document.

Although XML was designed primarily for the Web, its usefulness extends beyond the confines of a browser's window or an HTML page. XML separates data from layout in a way that hides both the data source and the formatting from one another. It does not matter whether the XML came from a database or from someone typing it in Notepad; if the file is well-formed (that is, conforms to XML syntax rules), any XML parser can read it. This is not always the case with other types of data exchange. Often, a developer must deal with the back-end data structure in order to extract the data he or she wants. With XML, data can be locally consumed, created, or modified from a logical data structure that is independent of all back-end implementation. Multiple data sources may feed data into a single type of XML structure, allowing seamless integration of disparate systems. Because XML is plain text, this integration may take place through the Web via HTTP.
XML documents contain information about themselves—metadata. Well-designed tags and attributes can be read, understood and used by both people and computers. Document Type Definitions (DTDs) or XML Schemas formally declare the type of XML document. XML can be transformed using the Extensible Style Language (XSL). XSL is of two types of XSL: XSL for formatting and XSL for data transformation. XML documents may be converted into a formatted, human-readable document, or transformed into another data structure, including another XML document with a different DTD. XML documents heavily use the concept of 'name space' to uniquely refer to parts of XML documents. A namespace is a set of names in which all names are unique. The members of a namespace can be referred to without ambiguity through namespace-qualified names. Namespaces allow for shorter names and help in processing XML documents without any name collisions.

5.1.2 XML Parsers

Document Object Model (DOM) and SAX (Simple API for XML) are the most common parsers available in the industry.

Document Object Model (DOM)
The Document Object Model (DOM) is a standard for programmatically accessing the structure and data contained in an XML document. The DOM is based on an in-memory tree representation of the XML document. When an XML file is loaded into the processor, it builds an in-memory tree that correctly represents the document. The DOM defines the programmatic interface (including the names of the methods and properties) that should be used to programmatically traverse an XML tree and manipulate the elements, values and attributes. When DOM is used to manipulate an XML text file, it parses the file, breaks it into individual elements, attributes and so on. It then creates a representation of the XML file as a node tree in memory. The contents of the document may then be accessed and manipulated through the node tree using the DOM interfaces.

SAX
The "Simple API" for XML (SAX) is an event-driven, serial-access mechanism that does element-by-element processing. SAX works through callbacks: call the parser, it calls methods that are supplied. A SAX parser reads the XML document as a stream of
XML tags and generates events such as starting elements, ending elements, text sections, etc corresponding to various components of an XML document.

**Difference between DOM and SAX**

- DOM reads the entire XML document into memory and stores it as a tree data structure where as SAX reads the XML document and sends an event for each element that it encounters.
- DOM provides "random access" into the XML document where as SAX provides only sequential access to the XML document.
- SAX is fast and requires very little memory, so it can be used for huge documents (or large numbers of documents). This makes SAX much more popular for web sites.
- Some DOM implementations have methods for changing the XML document in memory where as SAX implementations do not have such methods.
- SAX parsers generally require one to write a bit more code than the DOM interface.
- Limited API in SAX i.e., every element is processed through the same event handler. One needs to keep track of location in document and in many cases store temporary data.
- DOM is slow and requires huge amounts of memory, so it cannot be used for large XML documents.

**5.1.3 XPath**

XML does not specify how data is transmitted over the wire and it does not specify how data is stored. XML simply determines the format of the data. XPath [78], short for XML path language, has been specified by the W3C for addressing fragments of an XML document. XPath grew out of efforts to share a common syntax between XSL Transformations (XSLT) [119] and Xpointer [120]. It allows for the search and retrieval of information within an XML document structure. XPath is to XML as SQL is to relational database systems. XPath uses a compact, path-based, rather than XML element-based syntax. It operates on the abstract, logical structure of an XML document (tree of nodes) rather than its surface syntax. XPath uses a path notation (like URLs) to navigate through this hierarchical tree structure. This tree has 7 types of nodes.
Root - corresponds to the root of the document.
Element - corresponding to each element in the document.
Attribute - corresponding to the each attribute of an element.
Namespace - corresponding to the namespace that the sub-tree of an element belongs to.
Processing Instruction - corresponding to each processing instruction.
Comment - corresponding to each comment.
Text - Corresponding to parsed and unparsed character data within each element.

Document Order

Informally, document order is the order returned by an in-order, depth-first traversal of the hierarchical tree structure of an XML document. Within a tree, document order satisfies a set of constraints. Given below is a sample XML document

```xml
<?xml-stylesheet type="text/css" href="stylesheet.css"?>
<B xmlns:bk="http://www.books.com/nmsp" id="1">Text B</B>
<!--...comment text...-->
<C>Text C
   <D>Text D</D>
</C>
</A>
```

Figure 5.7: Tree (document-order) representation of the above XML document

Expression vs. Location path

An expression is the most general type of ‘path’ statement. Every XPath query is an expression. An expression can evaluate to yield an object of 4 types – 1. Node-set (unordered collection of nodes without duplicates), 2. Boolean (true/false), 3. Number (float) and 4. String (sequence of UCS chars). An expression that returns only a node-set is called a
location path. Each location path in turn consists of a series of location steps. A location step looks as below.

\[
\text{Location step} = \text{Axis}:: \text{node-test} \ [\text{optional predicate}]
\]

Where \( \text{predicate} = \text{location path} \text{ and/or location path} \)

\[
\text{Location path} = \text{location step} / \text{location step} ....
\]

Location paths can be relative or absolute. Relative location paths consist of one or more location paths separated by backslashes. Absolute location paths consist of a backslash optionally followed by a relative location path. In other words, relative location paths navigate relative to the context node. Absolute paths specify the absolute position within the document. An absolute location path would then be:

\[
/fileys/drive[@letter='C']/folder[@name='XML']
\]

Using an absolute location path, the current context node is ignored when evaluating the XPath query, except for the fact that the path being searched exists in the same document.

**XPath Axis**

The axis component of an XPath query determines the direction of the node selection in relation to the context node. An axis can be thought of as a directional query. In XPath, there are *forward* axes that only select nodes that are after the context node in document-order and *reverse* axes that select nodes that are before the context node in document order. For every reverse axis in XPath there is a corresponding forward axis that is symmetrical to the reverse axis.

The following pairs of axes are defined in XPath:

- **Child / parent:** The *child* axis selects all direct children of the context node, i.e. all nodes directly contained in the context node. The *parent* axis selects the parent node of the context node.

- **Descendant / ancestor:** The *descendant* axis selects all nodes in the subtrees starting with the children of the context node, i.e. the children of the context node and all descendants of those children. The *ancestor* axis selects all the nodes that the context node is a descendant of, i.e. all nodes that are on the path from the context node to the root of the document tree.

- **Descendant-or-self / ancestor-or-self:** This pair of axes is equivalent to the *descendant / ancestor* pair, except that also the context node is selected.
• **Following-sibling / preceding-sibling:** The siblings of a context node are all nodes in the document tree that have the same parent node as the context node. The following-sibling axis selects all nodes that follow the context node in document order and that are siblings of the context node. The preceding-sibling axis selects all siblings that precede the context node in document order.

• **Following / preceding:** The following axis selects all nodes that follow the context node in document order starting with the first following sibling, i.e. the descendants of all following siblings of the nodes selected by the ancestor-or-self axis relative to the context node (including those siblings).

• **Self:** The self-axis is symmetrical to itself and selects only the context node.

• **Attribute and namespace:** These special axes are used for selecting the attribute nodes of the context node and the namespaces in scope at the context node.

Following, following-sibling, preceding and preceding-sibling are also called as horizontal axes because the elements corresponding to these axes are found at the same level as that of the elements corresponding to the context node.

**XPath Node-Test**

The XPath node test does just what its name implies, i.e., it tests nodes to determine if they meet a condition. By specifying the node name in the node test component of the XPath statement, we limit the results so that only a single node is returned. A node-test may be: a node name, "prefix:*", "text()", "node()", "processing-instruction()" or comment() etc.

**Predicates**

A predicate acts as a further filter on the node set selected by the axis and node test. This is optional in any location step. There is no restriction on one location step to contain more than one predicate.

**Abbreviated Syntax**

In order to simplify the representation of most frequently used syntactic fragments of XPath, short notations are introduced in Xpath, like 1) V represents child. 2) attribute:: can be abbreviated to @. 3) // is short for /descendant-or-self::node(). 4) A location step of ‘.’ is short for self::node(). 5) Location step of ".." is short for parent::node()
Core Function Library

XPath defines a core set of functions and operators that are classified into Node-set, String, Boolean and number functions depending on the type of data they operate on. Few examples are given below.

1. Count(/descendant::X) is a node-set functions that returns the no of nodes in the argument node set.
2. Number() is a number function that converts the argument to number.
3. Starts-with() is a string function that takes two string arguments and checks to see if first starts with the second argument.
4. not(boolean) is a boolean function that returns true if its argument is false, and false otherwise.

Importance of Reverse Axes [102]

Current evaluation methods such as the one followed in DOM are very inefficient when evaluating reverse axes such as parent and ancestor. The fundamental difficulty caused by the reverse axes is that they incur bottom-up traversal in the document tree while the tree is usually traversed in pre-order. In streaming environment, since seeking-back in the stream is usually not allowed or very expensive, this difference (between the semantics of the reverse axes and the restrict of pre-order traversal) seems to be conflicting. Though theoretically it is possible to express every XPath query with forward axes alone (not using backward axes), it would reduce the expressiveness of the XPath query language. Moreover, reverse axes are important for the users. Firstly, it empowers the user to specify more complex patterns. Without reverse axes, XPath queries can only specify tree patterns (where the edges are of forward axis), while with reverse axes the pattern could be a graph instead of tree.

The reverse axes are also very convenient for user to specify query that can fit data of various DTDs or schema. For example, in the scenario of information dissemination, it is very likely a query issued by the user will be applied to heterogeneous data sources. For example, the query “//book[subject="XML" or parent::pub="O'Reilly"]”, which can be executed on both the following XML fragments, cannot be expressed in single query with only forward axes.

```
<book>
  <subject>XML</subject><subject>...</subject>
</book>
```
5 Approaches to handle reverse axes

Following are three principal options how to evaluate reverse axes in a stream-based context: (proposed in [98]).

- Storing in memory sufficient information that allows to access past events when evaluating a reverse axis. This amounts to keeping in memory a (possibly pruned) DOM representation of the data.
- Evaluating an XPath expression in more than one run. With this approach, it is also necessary to store additional information to be used in successive runs. This information can be considerably smaller than what is needed in the first approach.
- Replacing XPath expressions by equivalent ones without reverse axes.

Our approach closely follows the third approach.

5.2 Existing XPath Engines

There are two approaches of processing XML documents namely, index-based and navigation-based. Index-based approaches such as index-filter build an index over the input XML document and probe the same for matching with queries. Most of existing XML processing engines follow the other approach, i.e., navigation-based approach. In this approach the queries are evaluated as the processor navigates through or parses the input document. Proposed YALXP follows navigation-based approach.

Table 5.1 Existing XPath engines and their scope

<table>
<thead>
<tr>
<th>Engine</th>
<th>Streaming XML</th>
<th>Single</th>
<th>Multiple</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xfilter</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Yfilter</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Xtrie</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Mtrie</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>MQSPEX</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPEX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XSQ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>XAOS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Xalan</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

*$Single$ - Single-query processing system, $Multiple$ - Multiple-query processing system. All systems invariably handle forward axes.

The table 5.1 gives a list (though not exhaustive by any means) of existing work done in the area of XPath processing. Xfilter[82], Yfilter[83], Xtrie[86] and Mtrie[87] do not focus on backward axes. SPEX[101], XSQ[97][102], MQSPEX[92] and XAOS[79]
are streaming XML parsers that handle both forward and reverse axes. Apart from above systems, several algorithms have been proposed supporting different fragments of XPath. Substantial work has also been done to systematically assess time and space complexities for XPath evaluation. The paper "XPath Query Evaluation: Improving Time and Space Efficiency" [93] presents some improved time and space efficient algorithms on non-streaming XML documents. The paper “On the Memory Requirements of XPath Evaluation over XML Streams” [103] does the first formal analysis of the time and space complexities and presents the minimal space and time requirements of XPath evaluation on streaming XML documents. In this section, we give a brief introduction to different approaches to XPath processing. We give a detailed description of XAOS as our method is built upon the same.

5.2.1 Forward-only XPath Processors
In this section we describe XPath engines, which take care of only forward axes of XPath.

5.2.1.1 Xfilter

Xfilter [82] is a benchmark for all the XML processing systems. Based on the XFilter system, several filtering engines for the selective dissemination of information (SDI) represented in XML have been proposed recently. These systems have focused on efficient filtering of (relatively small) XML messages or documents according to subscriptions expressed as XPath queries. The Xfilter system sets up the use of deterministic finite automata for filtering of XML data and proposes a novel query index optimizing state transitions of the DFAs: An incoming element label is used as key in a hash of all element labels occurring in any subscription. In a hash table the states (representing a step in the XPath expression) reachable from the current state by the associated hash key are noted. For each such state \( s \) in the hash table of the incoming element, all states corresponding to steps following the associated step of \( s \) in the subscription are added to the appropriate hash table. For the average case this leads to a very efficient selection of the state transitions in the DFAs. XFilter does not perform multi-query optimization. CQMC (Continuous Queries for Mobile Clients)[84] modifies Xfilter approach to exploit commonalities among queries.
5.2.1.2 Yfilter

The optimization of multiple queries by sharing common prefixes is the principal contribution of Yfilter [83]. To enable prefix sharing, an NFA is used instead of multiple DFAs as in XFilter. Experimental evaluation shows a considerable lower processing time compared to XFilter on large number of subscriptions. Furthermore, two optimizations for the handling of predicates and nested path expressions (such as in $/a[b/c]/d$) are proposed. Selection postponement delays the evaluation of value-based predicates until a structural match is reached (thus avoiding the evaluation of predicates where no structural match is reached for the remaining expression). Both optimizations are based on the assumption that it is affordable to store possibly all nodes in a document for further processing, an assumption invalid for unbounded streams. In YFilter, the authors argue that their experimental evaluation shows that the cost for matching a subscription is no longer the dominant cost if compared to parsing and further processing and thus no further optimizations (e.g., to avoid the exponential complexity) is necessary.

5.2.1.3 Xtrie

Xtrie[86] decomposes complex, tree-structured XPath expressions (XPE) into a set of simple and linear patterns (sub-strings) and indexes the sub-strings in a trie structure. This trie facilitates detection of sub-string matches in the input XML data. Each sub-string is a sequence of element names along some path in XPE tree, where each consecutive pair of nodes is related by a “/” operator (without any “*” or “/”). Information about each sub-string of each XPE is stored in a sub-string table (ST). The information in ST is used to check for partial matches. A matching algorithm using these two structures does query evaluation.

5.2.2 Backward-Axis XPath Processors

Two methods (other than XAOS[79]) have so far been proposed that handle backward axes on streaming XML. One is XSQ[97,102] and other is SPEX[101]. Both works have similarities in their approach. In this section we give a brief description of the two and the non-streaming backward-axis XPath processing engine Xalan.

5.2.2.1 XSQ

XSQ[97,102] system uses a pushdown transducer (PDT) - an abstraction of the input query, to process the events that are generated by a SAX parser[104]. PDT is a pushdown automaton with actions defined along with the transition arcs of the automaton. In the
start state, the PDT is initialized. At each step, based on the next input symbol and the symbols in the stack; it changes state and operates the stack according to the transition functions. The PDT also does an output operation, which could generate output during the transition enabling eager emission of output. PDT makes use of buffer to support these operations. XSQ was later extended for reverse-axes. The extended XSQ addresses a large segment of XPath and employed a totally new approach.

5.2.2.2 SPEX
SPEX-Streaming and Progressive Evaluation of Xpath [101] employs almost the same mechanisms as followed by XSQ (forward-only), but addresses broader fragment of XPath such as nested predicates etc. SPEX also rewrites a query with reverse axes to reverse-axis free one.

\[ Q1 \text{(Query): } /desc::b[desc::d \text{ or child::e}]/parent::a/fsibl::c \]

\[ Q2 \text{(Rewritten Query by SPEX): } /desc::a[child::b[desc::d \text{ or child::e}]]/fsibl::c \]

A query graph is created for each query, where syntactical details of XPath are abstracted out. The query graph (or query plan) is implemented by a network of deterministic pushdown transducers. The collective incremental work of transducers yields the answers to the original query.

5.2.2.3 MQSPEX
This method is an extension of the previous work, SPEX - Streaming and Progressive Evaluation of Xpath, for multiple queries. MQSPEX[92] finds the optimal cost query plan that allows the simultaneous evaluation of multiple queries against the same stream. MQSPEX optimizes processing for multiple queries by taking advantage of the arbitrary commonality among different queries along with common prefixes by building optimized query plans for multiple queries. Their experiments show that sharing arbitrary operators under a realistic cost function results in query plans that have consistently lower cost for reasonable sets of queries than query plans where only common prefixes are considered. MQSPEX also claims to have addressed a larger and complex fragment of XPath than many other engines.
To explain the evaluation of a query in Xalan, let us consider the query

"Descendant::U/descendant::Y | descendant::V and ancestor::W/descendant::X | descendant::Z"

on the document in first column of figure 5.3. The tree corresponding to the document looks as in second column of figure 5.3. The query evaluation takes place in 3 traversals. In first traversal, elements 2, 6 and 8 are selected. In second traversal, 3, 7 and 9 are selected. In the next traversal, 5, 13 and 16 are selected, which are the output of the query. The performance of Xalan is worse when the query has more descendant and ancestor axes. This is because, for descendant of an element Y all the elements within the sub-tree of Y have to be searched and for the ancestor of Y all the ancestors of Y up to Root have to be searched.

5.2.2.5 XAOS

XAOS[79], pronounced as Chaos, is an algorithm to evaluate XPath expressions having both forward and backward axes on streaming XML data. Though it was originally designed for only two backward axes namely parent and ancestor, it is extensible to other
backward axes. XAOS caters to only location paths of the XPath specification. This particular subset of XPath is termed as Rxp (Restricted XPath). XAOS works on two views of the input query namely x-tree and x-dag. X-tree is a rooted tree that has x-nodes corresponding to each node test in the query. Each x-node, but for the ‘Root’, has a unique incoming edge that is labeled with the relationship (such as descendant/parent etc.) it shares with the source x-node. For example, figure 5.5 depicts the x-tree for a query. There exists one output x-node in this x-tree. (XAOS is extensible for multiple output nodes). X-node labeled E(5) is the output node in figure 5.5.

Figure 5. 9: (a) An XML Document (b) Tree representation of the same
The number in parentheses next to the tag of each element is the id of the element. Query in figure 5.5 is based on this document

The backward axes such as ancestor and parent in x-tree are converted to forward axes and the resulting view or data structure is called x-dag (x-directed, acyclic, graph). This conversion is done in three steps. First step reverses the direction of backward edges. The labels of these edges are renamed with corresponding forward axes. Ancestor relation is renamed as descendant and parent relation is renamed as child. For example, in figure 5.5, the parent edge from 'E' to 'D' is changed to a child edge from 'D' to 'E'. The last step consists of adding an incoming edge with label descendant from 'Root' to each x-node (other than 'Root') that is left with no incoming edge. In figure 5.5, a descendant edge is added from 'Root' to 'D'.
After the two views x-tree and x-dag are ready, execution of XAOS proceeds with the filtering of events. Matching structures are created for matched elements and finally the output is emitted at the end of the document. Filtering of events consists of construction of a looking-for-set at the end of each event making use of the x-dag view of the query. Looking-for-set consists of the set of relevant elements to be searched for in the next event. An element is open if we saw the start tag of the element and we did not yet see the end tag of the same. An element is relevant if this element matches with one of the elements we are looking for. For an element to be looked for in next event, it should have an open and relevant element for each of the remaining parents in x-dag.

Table 5.2: Walk through of evaluation of XPath on XML document of Figure 5.4. In the first column, Start/End:A(x,y) denotes the start/end element event for an element A(x,y) in XML document with x as node id and y as level. The Looking-for-set column shows L (set of (element, level)) at the end of processing the event.
Each item in the looking-for-set has the name of the element to be looked for and the
level at which it is to be found. If the outgoing edge is "descendant", then the element we
are looking for can be found at all levels above the level of current element before we see
the end tag of current element. In above example, the start document event matches with
'Root' of x-dag. 'Root' of x-dag has two outgoing edges namely descendant to x-nodes
'C' and 'D'. So looking for set at the end of this event includes the items (C, *) and (D,
*). Similarly if the outgoing edge is "child", then the element we are looking for can be
found at a level equal to (1 + the level of the current element). In above example, the start
event of C(3,3) matches with 'C' of x-dag. Since 'C' has an outgoing edge namely child
to x-node F, looking for set at the end of this event includes the item (F, 4). 'C' also has
an outgoing edge namely descendant to x-node E. But we did not see an open and
relevant element that matches with 'D', which is a parent of 'E'. So looking for set does
not include the item (E, *). By doing this, most of the irrelevant elements are discarded.

Figure 5.11: Matching Structure after evaluation of query in figure 5.5 on document in figure 5.4

Matching structures are built using x-tree representation of the query. Each
matching structure is represented by \( M_{e,i} \), where 'e' is the element name and 'i' is the id
of the element. (XAOS uses another representation interchangeably with this, that is \( M_{v,e} \)
- where V is x-node and 'e' is element in XML document that matches. If a start event
of an element is found to be relevant (i.e. matches with an x-node in the x-tree) a
matching structure is created for the match. Each matching structure has the node test
(name) of the element and pointers to sub-matches corresponding to each child x-node in
the x-tree. At the end element event of this element, a key step called propagation is done
to see if there is total matching at the matching for this element according to three conditions. If the corresponding x-node is a leaf in the x-dag, there is total match for the corresponding matching structure. In above figure 5.6, MF,6 corresponds to a leaf in x-dag. So, at the end element event for F(6,4), this matching is propagated to its parent matching M_{C,3}. If the corresponding x-node is not a leaf in the x-dag and all its submatches are non-empty and total, there is total match at the corresponding matching structure. In figure 5.6, Mc,3 corresponds to a non-leaf in x-dag. And there is total match for its sub-matches ME,5 and MD,4. SO, at the end element event for C(3,3), M_{C,3} is propagated to its parent matching M_{Root,0}. At the end of document event, the same check is done at the 'Root' matching structure.

There is a catch in this process. If the x-tree does not have ancestor and parent labels, this process is straightforward as above. Otherwise, i.e. when there is/are ancestor and parent outgoing edges from an x-node X to x-node X' for which matching occurred at an event, the propagation is optimistic at the corresponding end of element event. This special case is dealt in two steps.

1. At the start event of an element Y that matches X, if the sub-matching M_{Y,i} (at an element Y' with id 'i') corresponding to the x-node X' is empty, it is optimistically propagated that M_{Y,i} is total. Similarly if other sub-matches of Y are non-empty and total it means there is a total match at Y. This total match is propagated to appropriate parent-matches.

2. At the end event of Y', if it can now be determined conclusively that M_{Y,i} is not total, the optimistic propagation should be recursively undone from its parent and other superior matches.

In above example, the x-node 'E' has an outgoing edge labeled 'parent'. So by the time the end of element event for E(5,5) occurs, we have not seen the end tag for D(4,4). So, at this point, logically there is no total matching at D(4,4). According to XAOS, it is assumed that MD,4 has total matching and the same is propagated upwards to M_{E,5}. However, since there is total matching at MD,4 (i.e. we saw end tag of D(4,4)), we do not
need undo this optimistic propagation at end of element event of \( D(4,4) \) from \( M_{E,S} \) and its superior matches.

At the end of document event, \( M_{\text{Root},0} \) has total matching since its sub-matching \( M_{C,3} \) is non-empty and total. If there are one or more, then these matches should be suitably emitted by traversing the whole matching structure. The solution for the example is \( \{E(5,5)\} \). And total matching at Root is \( \{\text{Root}\rightarrow 0, C\rightarrow 3, F\rightarrow 6, D\rightarrow 4, E\rightarrow 5\} \).

5.3 YALXP - Yet Another Light Weight XPath Processor

We continue to call each XPath Expression as an Rxp (Restricted Xpath), since we cater to essentially that fragment of XPath which XAOS addresses. YALXP operates on a combined view of all the input XPath expressions called combined x-dag or c-dag. Combined matching structure that represents the matching elements for c-nodes in the c-dag is built as the execution of the algorithm progresses.

YALXP can be viewed as an application built on top of an event-based XML parser such as SAX. It consumes the events sent by SAX parser and deals with them appropriately. At the end of every event we maintain a looking-for-set that has the set of elements expected in the next event, along with ‘qids’ (query identifier) for which they are likely to match. For an element to be relevant it has to match with at least one of these items. This way, we can filter out irrelevant events. Relevant elements along with the links among them are stored in combined matching structure. YALXP uses combined x-dag to build combined matching structure. At the end of the algorithm, we emit the output matches for different queries with the help of individual x-tree for each query. We stick to the lazy emission of output (at the end document event), a concept originally followed by XAOS. This chapter describes the construction of combined x-dag, the structure of combined matching structure and complete process of query evaluation.

5.3.1 Combined X-Dag

YALXP operates on a combined representation of the input Rxp set that is called combined x-dag or c-dag. Combined x-dag is a directed and acyclic graph that is obtained by combining individual x-dags constructed for each Rxp in the input Rxp set. Combined x-dag is built on the premise that there exists a significant commonality among different Rxp’s in large-scale systems.
Commonality not only exists in the string form of the XPath queries. As we explain below commonality may exist in two Rxp's, which need not have any common sub-string. In figure 5.7, the two Rxp’s do not have any commonality in the string form of the Rxp’s. But after the x-dags are drawn, they look like mirror images of each other. This is the kind of commonality we are going to exploit by building combined x-dag for multiple Rxp’s.

**Figure 5.12: Commonality in two queries looking different**

\[ Q1 : /\text{descendent} : Z \preceq /\text{descendent} : Y \preceq /\text{parent} : X \]  
\[ Q2 : /\text{descendent} : X \preceq /\text{child} : Y \preceq /\text{ancestor} : Z \]

Each node in c-dag is called c-node. Each c-node represents a set of x-nodes of different queries. We define info, an ordered pair \((qid, x\text{-nodeid})\), which stores the information about each x-node that a c-node represents. For example, if a c-node has the info \((qid, x\text{-nodeid})\) with it, it means that this c-node corresponds to an x-node with id ‘x\text{-nodeid}’ in the x-dag with id ‘qid’. Each edge in c-dag represents edges of different x-dags. At each edge in c-dag, we store the qids of x-dags in which the edge participates. Each edge also has a label associated with it, which is either Descendent or Child. The c-dag is constructed in such a manner that it is possible to reconstruct individual x-dags from the c-dag.

The construction of c-dag starts with the construction of individual x-dag for each Rxp. This involves construction of individual x-tree for each Rxp as explained in XAOS. X-dags for all Rxp’s are built from these x-trees by translating all the backward constraints (‘ancestor’ and ‘parent’) to forward constraints (‘descendant’ — represented by dotted edge and ‘child’ — represented by thick edge in figure 5.7) according to the rules specified by XAOS. Nodes in red in figure 5.7 are the output nodes.
After this step, the c-dag is initialized to the first x-dag (corresponding to the first Rxp). Now the c-dag has a set of c-nodes and edges that correspond to x-nodes and edges of the first x-dag. At each c-node thus created, info (1,x-nodeid), where x-nodeid is x-node that the c-node represents, is registered. qid ‘1’ is registered at each edge in the c-dag. To the ‘Root’ of c-dag, info (qid,0) about Root x-node of each x-dag(qid) is added, since Root of c-dag matches with Root’s of all the x-dags.

After this initialization process, the rest of the x-dags are added to the c-dag by traversing each x-dag in depth-first fashion. The x-node ‘x_j’ whose outgoing edges and outgoing x-nodes (the x-node to which an outgoing edge leads) are going to be added to the c-dag. There are six scenarios listed as cases below in bold in which an outgoing edge and outgoing x-node can be added to c-dag. The steps with small Roman letters under each case in the list are the operations to be performed in that case. We followed the following conventions in the list. ‘x_k’ is the outgoing x-node being added. c(qid,j) is the c-node corresponding to x-node with id ‘j’ in x-dag for qid. We say that a c-node exists for ‘x_k’ if this c-node has info matching with qid and k (id of x_k).

1. If a c-node exists for x_k, i.e., an outgoing c-node of c(qid,j) and an edge with label axis(Xj,Xk) exists between c(qid,j) and c(qid,k)
   i. Register qid at edge between c(qid,j) and c(qid,k);

2. If a c-node exists for x_k, i.e., an outgoing c-node of c(qid,j) and no edge with label axis(Xj,Xk) exists between c(qid,j) and c(qid,k)
   i. Let prevnode = c(qid,j, k);
   ii. Remove info (qid,k) from prevnode;
   iii. Create a new c-node for x_k;
   iv. Add info to this c-node, which is c(qid,k) from now on;
   v. Add edge with label axis(Xj,Xk) between c(qid,j) and c(qid,k) and register qid at this edge;
   vi. Add incoming edges of prevnode w.r.t. x-dag(qid) to c(qid,k) and register qid at the same;
   vii. Remove qid from incoming edges of prevnode;
   viii. Add outgoing edges of prevnode w.r.t. qid, to c(qid,k) and register qid at the same;
ix. Remove qid from outgoing edges of prevnode;

3. If a c-node exists for $x_k$ that is not an outgoing c-node of $c(qid,j)$
   i. Add an edge with label $axis(x_j,x_k)$ between $c(qid,j)$ & $c(qid,k)$;
   ii. Register qid at this edge;

4. If no c-node exists for $x_k$, an outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$ and the edge between $c(qid,j)$ and this outgoing c-node has label $axis(x_j,x_k)$
   i. Add info to this c-node;
   ii. Register qid to edge between $c(qid,j)$ and $c(qid,k)$;

5. If no c-node exists for $x_k$, an outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$ and the edge between $c(qid,j)$ and this outgoing c-node has label not equal to $axis(x_j,x_k)$
   i. Add info to this c-node;
   ii. Add an edge with label $axis(x_j,x_k)$ between $c(qid,j)$ & $c(qid,k)$;
   iii. Register qid to edge between $c(qid,j)$ and $c(qid,k)$;

6. If no c-node exists for $x_k$ and no outgoing c-node of $c(qid,j)$ has name equal to name of $x_k$
   i. Create new c-node with name equal to name of $x_k$;
   ii. Add info to newly created c-node;
   iii. Add an edge with label $axis(x_j,Xk)$ between $c(qid,j)$ and $c(qid,k)$;
   iv. Register qid to edge between $c(qid,j)$ and $c(qid,k)$;

In cases 1, 2 and 3 where $x_k$ was not visited in a previous traversal of x-dag, edges and x-nodes out of $x_k$ are added in similar manner as that of $x_j$. In cases 4, 5 and 6, there is no need to add these edges and x-nodes as they would have been added to c-dag when $x_k$ was visited previously. It is understood that while un-registering qid from edges, if the edge is not left with any qid after removal, the edge itself is removed.

Similarly the remaining x-dags are also added to the c-dag. The resulting c-dag after this step has all the commonalities among different Rxps exploited to the optimal extent (to the extent that there is no loss of information).
5.3.2 Sample Construction of c-dag

We explain the construction of c-dag for two queries in given in figure 5.8 below. In figure 5.8, the conventions followed are as follows. A c-node or x-node is represented as \( N(i) \) where ‘N’ is the name of the node and ‘i’ is the id of the node. The notation \( e(N(i),M(j),\text{axis}) \) represents edge between \( N(i) \) and \( M(j) \) with label ‘axis’. For example, \( e(\text{Root}(0),\text{W}(2)),\text{descendant} \) represents edge between \( \text{Root}(0) \) and \( \text{W}(2) \) with label ‘descendant’. The set such as \{1,2\} near an edge represents the qids in which this edge participates. The pairs such as (1,2), (2,1) near each c-node represent the x-nodes to which this c-node corresponds, where first part is qid and second part is x-node id.

Figure 5.13: Construction of c-dag for queries

\textbf{Q1}: /descendant: :U \text{ parent:::W/ descendant::X parent::V/ descendant::Y} \text{ and }

\textbf{Q2}: /descendant::U[ancestor::W/ descendant::X[ancestor::V/ descendant::Z/ descendant::X]]

\textbf{x-dag1 for Q1}:

\textbf{x-dag2 for Q2}:
Step 1:
In this step, \( U(l) \) and \( e(\text{Root}(0),U(1),\text{descendant}) \) of x-dag 2 are added to \( \text{Root}(0) \) of c-dag. There is no \( C(2,1) \), i.e. c-node with info \( (2,1) \) in c-dag. However, \( \text{Root}(0) \) of c-dag has an outgoing c-node with name ‘U’, i.e. \( U(l) \). And \( \text{Root}(0) \) and \( U(l) \) of c-dag have a descendant edge between them. So this step falls into the Case 4. Hence info, i.e. \( (2,1) \) is added to \( U(l) \) in c-dag and qid ‘2’ is registered at \( e(\text{Root}(0),U(1),\text{descendant}) \) of c-dag.
Step 2:
In this step, W(2) and e(Root(0), W(2), descendant) of x-dag 2 are added to Root(0) of c-dag. info, i.e. (2,2) is added to W(3) in c-dag and qid ‘2’ is registered at e(Root(0), W(3), descendant) of c-dag, according to Case 4.

Step 3:
In this step, U(1) and e(W(2), U(1), descendant) of x-dag 2 are added to W(3) of c-dag. There is C(2,1), i.e. c-node with info (2,1) in c-dag, i.e. U(1). And W(3) and U(1) of c-dag have a child edge (added while adding x-dag 1) between them. So this step falls into the Case 2. So a new c-node with name ‘U’ and id ‘6’ is created and an edge with label ‘descendant’ is added from W(3) to U(6) and qid ‘2’ is registered at this edge. The incoming edge to U(1) w.r.t. x-dag 2, i.e. ‘descendant’ from Root(0) is added to U(6) and qid ‘2’ is registered at this edge.

Step 4:
In this step, X(5) and e(W(2), X(5), descendant) of x-dag 2 are added to W(3) of c-dag info, i.e. (2,5) is added to X(5) in c-dag and qid ‘2’ is registered at e(W(3), X(5), descendant) of c-dag, according to Case 4.

Step 5:
In this step, V(3) and e(Root(0), V(3), descendant) of x-dag 2 are added to Root(0) of c-dag. info, i.e. (2,3) is added to V(2) in c-dag and qid ‘2’ is registered at e(Root(0), V(2), descendant) of c-dag, according to Case 4.

Step 6:
In this step, X(5) and e(V(3), X(5), child) of x-dag 2 are added to V(2) of c-dag. There is C(2,5), i.e. c-node with info (2,5) in c-dag, i.e. X(5); and V(2) and X(5) in c-dag have a child edge between them. So this step falls into the Case 1. Hence, qid ‘2’ is registered at e(V(2), X(5), child) of c-dag.

Step 7:
In this step, X(6) and e(V(3), X(6), descendant) of x-dag-2 are added to V(2) of c-dag. There is no C(2,6), i.e. c-node with info (2,6) in c-dag. However, V(2) of c-dag has an outgoing c-node with name ‘X’, i.e. X(5). But, V(2) and X(5) of c-dag have a child edge between them. So this step falls into the Case 5. So a new c-node with name ‘X’ and id
'7' is created, an edge with label 'descendant' is added from V(2) to X(7) and qid '2' is registered at this edge.

**Step8:**
In this step, Z(4) and e(Root(0),Z(4),descendant) of x-dag-2 are added to Root(0) of c-dag. There is no C(2,4), i.e. c-node with info (2,4) in c-dag and Root(0) of c-dag has no outgoing c-node with name 'Z'. So this step falls into the Case 6. So a new c-node with name 'Z' and id '8' is created, an edge with label 'descendant' is added from Root(0) to Z(8) and qid '2' is registered at this edge.

**Step9:**
In this step, X(5) and e(Z(4),X(5),descendant) of x-dag-2 are added to Z(8) of c-dag. An edge with label 'descendant' is added from Z(8) to X(5) and qid '2' is registered at this edge, according to Case 3 (explained below).

**Step10:**
In this step, X(6) and e(Z(4),X(6),descendant) of x-dag 2 are added to Z(8) of c-dag. There is C(2,6), i.e. c-node with info (2,6) in c-dag, i.e. X(7); and Z(8) of c-dag has no outgoing edge to X(7). So this step falls into the Case 3. Hence, an edge with label 'descendant' is added from Z(8) to X(7) and qid '2' is registered at this edge.

### 5.3.3 Combined Matching Structure

We extend the concept of matching structure of XAOS to combined matching structure in case of multiple Rxp's. A combined matching structure represents the matches for all the input Rxp's (i.e. for different c-nodes) at the Root of the c-dag. We store the information of an element that matches a particular c-node C in a matching structure represented as $M_{x,(qid's)}$ where 'x' is name and 'qid's' are the ids of the input Rxp's for which $x$ matches. This matching structure has pointers to matching structures of elements that match with outgoing c-nodes of C.

---

1 We use the term combined matching structure to denote the whole structure and matching structure to denote individual matching structure within combined matching structure.
Figure 5.14: Combined Matching structure

$M_{\text{Root}, 0}(\text{qid's})$ represents the Root matching structure, where qid’s are the ids of the input Rxp's for which there is a matching at the Root of the c-dag. A matching structure $M_{x, i}(\text{qid's})$ is said to be a parent-matching of a matching structure $M_{x', i}(\text{qid's})$ if $x$ corresponds to a c-node that is a parent of the c-node for which ‘$x$’ has matching. $M_{x', i}(\text{qid's})$ is said to be the child-matching or sub-matching of $M_{x, i}(\text{qid's})$.

Diagrammatically, there are 3 rows in a matching structure. First row has the element name along with its id and level. Second row has the qids for which the matching takes place. Third row has slots, which contain pointers to sub-matchings for the children of respective c-node in the c-dag. There is no need to store the edge information (qids at these pointers) in combined matching structure (since these pointers match for the same set of queries, id’s of which are present at the destination sub-matching). As the query evaluation progresses, combined matching structure is built based on c-dag.

5.3.4 Query Evaluation Algorithm

The query evaluation process occurs at four events namely 'start document', 'start element', 'end element' and 'end document'. This process can be split into two components. One is the matching process, which occurs at all events, that filters out irrelevant events simultaneously storing relevant elements in combined matching structure. The next component is emission of the output, which occurs at 'end document' event using individual x-tree of each Rxp.

5.3.4.1 Matching Algorithm

Following points are to be considered in the matching process.

111
1. An element is open if we saw the start tag of the element and we did not see the end tag of the same.

2. An element $e$ is relevant only if this element matches with one of the elements we are looking for.

3. It is decided that there is a total matching at a matching structure for a particular $Rxp$, if there is at least one sub-matching, that has total matching, corresponding to each of the child c-nodes with respect to that $Rxp$ in the combined x-dag.

**Filtering Events**

Filtering of irrelevant events is enabled by the construction of looking-for-set. At every event, based on c-dag, we construct looking-for-set that consists of elements expected in next event. Each item in the looking-for-set, called Ifitem, corresponds to a c-node. An Ifitem has the name of the element to be looked for, the level at which it is to be found and the $Rxp$ ids (mentioned in table 5.3 with superscripts) with which it matches. For an element to be looked for, with respect to an $Rxp$, in next event, it should have an open and relevant element for each of the remaining parents$^2$ in c-dag with respect to that $Rxp$. If an element matches with a c-node in the c-dag for a particular set of qids, only the outgoing c-nodes in c-dag are considered to be looked for in the next event. While adding an Ifitem to looking-for-set, the set of qids to be considered for this item is obtained by the intersection between the set of qids present at the outgoing edge and the qids for which current element matched. If the outgoing edge of the c-node (for which current element matched) is "child" for a set of qid's, the outgoing c-node will have a matching element at a level = \(+level\) of the current element. Otherwise, if the outgoing edge is “descendant” for a set of qid's, an element that matches the outgoing c-node is at all levels above the level of current element before we see the end tag of current element.

In figure 5.11, we have shown c-dag having only the ids of $Rxp$'s of the x-nodes that a c-node represents without corresponding x-node ids, for easier visualization. This c-dag was constructed in the same manner explained in earlier section. In the c-dag, $W^{1,2,3}$ is the parent of $X^{1,2,3}$ with respect to $Rxp$'s land 2 but not with respect to $Rxp$ 3 since the

---

$^2$ A c-node is said to be a parent of a c-node with respect to an $Rxp$ only if there exists an edge between these two c-nodes that contains the id of the $Rxp$ in question.
child edge between these two c-nodes has only 1 and 2 registered at it. All these cases are clearly described at one place in Step 2 of the table 5.3.

Figure 5.15: Sample XML document and x-trees built

Sample XML document and x-trees built for the three queries Q1:
descendant::X [/ancestor::Y/child::U and /parent::W and /child::A] Q2:
descendant::W [/child::X /ancestor::Y/child::U] Q3: descendant::A
[/ancestor::W and /parent::X/ancestor::Y/child::U]

Figure 5.16: x-dags for x-trees built in figure 5.10 and the e-dag built from x-dags

Building Matching Structures
A new matching structure is created when we find an element matching with an item in the looking-for-set. This matching structure has only those qids for which the match occurred. In this way there may exist two or more sub-matches (each with different set of qid's) for a c-node in the combined matching structure.
Table 5.3: Walk through of evaluation of XPath queries on XML document of Figure 5.10 and 5.11. In the second column, Start (End): A(x,y) denotes the start (end) element event for an element, A(x,y). The last column shows looking-for set at the end of processing the event. Steps from 2 to 12 are described in same format as step 1.

<table>
<thead>
<tr>
<th>Step</th>
<th>Event</th>
<th>Matches</th>
<th>Looking-for set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start:Root(0,0)</td>
<td>(Root$^{1.2.3}$,0)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$)}</td>
</tr>
<tr>
<td>2</td>
<td>Start:Y(1,1)</td>
<td>(Y$^{1.2.3, \ast}$)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$), (U$^{1.2.3, 2}$), (X$^{1, \ast}$)}</td>
</tr>
<tr>
<td>3</td>
<td>Start:U(2,2)</td>
<td>(U$^{1.2.3, 2}$)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$), (X$^{1, \ast}$)}</td>
</tr>
<tr>
<td>4</td>
<td>End:U(2,2)</td>
<td>(U$^{1.2.3, 2}$)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$), (X$^{1, \ast}$), (U$^{1.2.3, 2}$)}</td>
</tr>
<tr>
<td>5</td>
<td>Start:W(3,2)</td>
<td>(W$^{1.2.3, \ast}$)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$), (X$^{1, \ast}$), (X$^{1.2, 3}$)}</td>
</tr>
<tr>
<td>6</td>
<td>Start:X(4,3)</td>
<td>(X$^{1, \ast}$), (X$^{1.2, 3}$)</td>
<td>{(W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$), (X$^{1, \ast}$), (A$^{1, 4}$)}</td>
</tr>
</tbody>
</table>

Comments: Add (W$^{1.2.3, \ast}$) and (Y$^{1.2.3, \ast}$) to L, since Root is an open, relevant element matching Roots of all x-dags. Do not start looking for (X$^{1.2, \ast}$) as there is no open and relevant element matching (W$^{1.2, \ast}$). Do not start looking for (X$^{1, \ast}$) as there is no open and relevant element matching (Y$^{3, \ast}$). Also do not start looking for (A$^{1, \ast}$) as there is no open and relevant element matching (X$^{1, \ast}$).

Start looking for U$^{1.2.3}$ at level 2 since U$^{1.2.3}$ is connected to Y by a child edge having qids 1,2 and 3 in the c-dag, and Y is matched at level 1. Start looking for (X$^{1.2, \ast}$) since X$^{3}$ is connected to Y by a descendant edge having qids 1,2 and 3 in the c-dag. Do not start looking for (X$^{1.2, \ast}$) as there is no open and relevant element matching (W$^{1.2, \ast}$). Continue looking for (W$^{1.2.3, \ast}$), (Y$^{1.2.3, \ast}$) because any element with these tags in the sub-tree of this element will also be a candidate for matching the same.

Stop looking for U$^{1.2.3}$ until we see the end of this element. Because, the level of any element we come across is greater than 2 before we see the end of this element.

There is a total matching at U(2,2) for the three Rxp's 1,2 and 3 represented as $M_{U,(1,2,3)}$ since it is a leaf in the c-dag. This matching is propagated to the appropriate sub-matching of $M_{V,(1,2,3)}$, which is the only parent-matching of $M_{U,(1,2,3)}$. Looking for set is built in same manner as for step 2.

Start looking for (X$^{1.2, 3}$) since W$^{1.2.3}$ has an outgoing child edge having qids 1 and 2 registered at it to X$^{1.2, 3}$.

Start looking for (A$^{1.3, 4}$) since X$^{1.2, 3}$ has outgoing edges labeled child and having qids 1 and 3 respectively to A$^{1}$ & A$^{2}$.
<table>
<thead>
<tr>
<th>7</th>
<th>Start: A(5,4)</th>
<th>(A$^{1.3}$, 4)</th>
<th>{(W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$, (X$^3,*$))}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continue looking for (W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$) and (X$^3,*$) as these are still candidates to match with elements in the sub-tree that follows in the input document.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>End: A(5,4)</td>
<td>(A$^{1.3}$, 4)</td>
<td>{(W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$, (A$^{1.3}$, 4), (X$^3,*$))}</td>
</tr>
<tr>
<td>There is total matching at A(5,4) for the two Rxp's 1 and 3 represented as M$<em>{A,5(1)}$ and M$</em>{A,5(3)}$. M$<em>{A,5(1)}$ is propagated to the appropriate sub-matching of M$</em>{W,3(1,2,3)}$. M$<em>{A,5(3)}$ is propagated to the appropriate sub-matching of M$</em>{X,4(3)}$, M$<em>{W,3(1,2,3)}$, M$</em>{Root,0(1,2,3)}$ which are parent matchings of M$_{A,5(3)}$. Looking for set is built in same manner as for step 6.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>End: X(4,3)</td>
<td>(X$^3,*$), (X$^{1.2.3}$, 3)</td>
<td>{(W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$), (X$^3,*$), (X$^{1.2.3}$)}</td>
</tr>
<tr>
<td>There is total matching at X(4,3) for the three Rxp's 1, 2 and 3 represented as M$<em>{X,4(1)}$, M$</em>{X,4(2)}$ and M$<em>{X,4(3)}$. M$</em>{X,4(1)}$ has total matching for the Rxp 1 and M$<em>{X,4(2)}$ has total matching for the Rxp 2. Because sub-matching of M$</em>{X,4(1)}$ for Rxp 1 in the form of M$<em>{A,5(1)}$ is total. And M$</em>{X,4(2)}$ for Rxp 2 corresponds to a leaf in the c-dag (x-dag(2)). M$<em>{X,4(1)}$ and M$</em>{X,4(2)}$ are propagated to the appropriate sub-matchings of M$<em>{Root,0(1,2,3)}$, M$</em>{W,3(1,2,3)}$ and M$<em>{Y,1(1,2,3)}$, which are the parent-matchings of M$</em>{X,4(1)}$ and M$<em>{X,4(2)}$ M$</em>{X,4(3)}$ is propagated to the sub-matching of M$<em>{W,3(1,2,3)}$ and M$</em>{Y,1(1,2,3)}$. Looking for set is built in same manner as for step 5.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>End: W(3,2)</td>
<td>(W$^{1.2.3,*}$, *)</td>
<td>{(W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$), (U$^{1.2.3}$), (X$^3,*$))}</td>
</tr>
<tr>
<td>There is total matching at W(3,2) represented as M$<em>{W,3(1,2,3)}$ for Rxp's 1, 2 and 3. Because M$</em>{W,3(1,2,3)}$ has sub-matchings in the form of M$<em>{X,4(1)}$, M$</em>{X,4(2)}$ and M$<em>{X,4(3)}$ which are total. This matching is propagated to the appropriate sub-matching of M$</em>{Root,0(1,2,3)}$, which is the only parent-matching of M$_{W,3(1,2,3)}$. Looking for set is built in same manner as for step 2 or 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>End: Y(1,1)</td>
<td>(Y$^{1.2.3,*}$)</td>
<td>{(W$^{1.2.3,<em>}$, (Y$^{1.2.3,</em>}$))}</td>
</tr>
<tr>
<td>M$<em>{Y,1(1,2,3)}$ has total matching for the three Rxp's since all its sub-matchings M$</em>{X,4(1)}$, M$<em>{X,4(2)}$, M$</em>{X,4(3)}$ and M$<em>{U,2(1,2,3)}$ are total. So this matching is appropriately propagated to the sub-matching of M$</em>{Root,0(1,2,3)}$. Looking for set is built in same manner as for step 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>End: Root(0,0)</td>
<td>(Root$^{1.2.3,*}$, 0)</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>There are total matchings at M$_{Root,0(1,2,3)}$ for the three Rxp's 1, 2&amp;3.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the example, the three matching structures $M_{x,4(1)}$, $M_{x,4(2)}$ and $M_{x,4(3)}$ were created in this fashion. Also with respect to same set of qid’s, there may be more than one element that match with a c-node and a matching structure is created for each such match i.e., there may exist another $M_{x,i(3)}$ (element X with id i different from 4) as a sub-matching of $M_{Root,0(1,2,3)}$. The sub-matchings are initially empty for a newly created matching. In figure 5.11 showing combined matching structure for the example, we grouped pointers related to sub-matchings for a single c-node in one slot of the third row of the matching structure.

One of the key steps in this algorithm is propagation. At the end element event of an element $x$ with id ‘i’, a check is performed to see if there is a total matching at $M_{x,i(RxpSet)}$ for each of the $Rxp$’s in $Rxp$’set. This propagation has two cases.

1. If the corresponding c-node is a leaf with respect to an $Rxp$ in the combined x-dag$^3$, the matching at this c-node is total for that $Rxp$. We propagate this match to its parent matchings appropriately (A point to be noted here is that the set of qid’s of a parent matching of a sub-matching is always a superset of the qid’s of sub-matching).

2. If it is not a leaf with respect to an $Rxp$, this matching represents a total matching with respect to that $Rxp$ if and only if all its sub-matchings with respect to that $Rxp$ are total. If we found appropriate total matching with respect to that $Rxp$ for all the child sub-matchings, we propagate this matching to its parent-matchings appropriately.

In the example, the first case occurs at step 9 where $M_{x,4(2)}$ is a leaf and has total matching with respect to qid 2. So this is propagated to its parents $M_{W,3(1,2,3)}$, $M_{Y,1(1,2,3)}$ and $M_{Root,0(1,2,3)}$. The second case also occurs in step 9 for the $Rxp$ 1. $M_{x,4(1)}$ is a non-leaf with respect to qid 1 and has total matching with respect to qid 1 for sub-matching $M_{A,5(1)}$, so this is also propagated to its parents $M_{W,3(1,2,3)}$, $M_{Y,1(1,2,3)}$ and $M_{Root,0(1,2,3)}$.

---

$^3$ A c-node can be a leaf with respect to one $Rxp$, i.e. has no outgoing edge with respect to that $Rxp$ and at the same time a non-leaf with respect to another $Rxp$, i.e. has an outgoing edge with respect to this $Rxp$. 

116
Emission of output

At the end document event, a check is done to see if total matching exists at ‘Root’ matching with respect to all Rxp’s. After this, before emitting the output, we traverse through the combined matching structure with respect to all individual x-trees in an iteration and determine total matching with respect to x-tree at each matching structure for all qids present at the matching structure. Then we traverse through the combined matching structure again with respect to each individual x-tree to emit the correct output when we first visit a matching structure that corresponds to the output x-node.

Consider the following scenario in emission of output. M(xl,qid) is a matching structure corresponding to x-node xl of x-tree(qid) where xl has an outgoing edge labeled ancestor or parent to an x-node x2. In this case, a matching structure for x2 is found as a parent matching structure of M(xl,qid). In the example, there arc total matchings for all 3 Rxp’s at MR Root,0(1,2,3), which are given in figure 5.11.

Figure 5.17: Combined matching structure at the end of executing algorithm on example

![Combined matching structure](image)

The total matchings at Root with respect to each Rxp are,

- For 1st Rxp:
  \{Root→0, Y→1, W→3, U→2, X→4, A→5\}

- For 2nd Rxp:
  \{Root→0, Y→1, W→3, U→2, X→4\}

- For 3rd Rxp:
  \{Root→0, Y→1, W→3, U→2, X→4, A→5\}

Elements in bold are the output.

5.4 Experimental Results

We performed our experiments on Windows 2000 Advanced Server with 450 MHz Intel Pentium 2 processor and 1024 MB as Main memory. We implemented XAOS and YALXP in Java 1.4.1. We used xml4j_2_0_9, developed by IBM as SAX parser. We implemented various versions of XAOS and modified Xalan 2.5 to perform our experiments. We also implemented custom XML generator that generates XML
document based on the queries generated by custom XPath generator (YALXP). Below we present details of each implementation.

**XAOS (OSPCN) - One SAX Parsing Commonality Not Exploited**

This version of XAOS has arrays of structures for multiple queries. Data structures required for processing each query such as x-tree, x-dag, looking-for-set and matching structure are maintained individually for each query. Processing at each event takes place in iterations for each query individually using the respective data structures for that query.

**XAOS (MSP) - XAOS (Multiple SAX Parsing)**

XAOS executed individually for each query in a loop. For n queries, SAX parsing takes place n times.

**Xalan (ODP) - Xalan (One DOM Parsing)**

This is the version of Xalan in which only one in-memory DOM representation of the input document is built. The same will be queried individually for each query one after the other.

**Xalan (MDP) - Xalan (Multiple DOM Parsing)**

In this version of Xalan (MDP), construction of in-memory DOM takes place for each query. Hence evaluation of a query takes a lot of time, thereby increasing the total time taken for multiple queries.

**Custom Xpath Generator (YALXP)**

We customized the custom Xpath Generator used by XAOS to generate multiple queries with commonality. Custom Xpath Generator (XAOS) takes “[-c<max_children>] [-e<number of elts>] [-nr (no-recursion)] [-na (no ancestors or parents)]” as parameters and generates a single query based on these parameters. Each query has unique node tests and each node test has alphanumeric characters generated randomly.

*Sample query generated by Custom Xpath Generator (XAOS)*

```
//k3//q3 and 10|//c2[.//a2 and 7/11 J//q2"
```

Custom Xpath Generator (YALXP) takes the number of queries 'n' to be generated as input in addition to above parameters and generates 'n' queries. Each query has 6 unique node tests pseudo-randomly generated from the domain {U, V, W, X, Y, Z} so that different queries have significant commonality.
Sample queries generated by Custom XPath Generator (YALXP)
1. /descendant::Y[ancestor::W and /child::X[descendant::V]/descendant::U]/descendant::Z
2. /descendant::Y[descendant::Z and /descendant::V]/child::W/descendant::X
3. /descendant::V/child::X[descendant::W[child::Y]/ancestor::U]/child::Z
4. /descendant::X/child::Z[child::V/ancestor::Y and /child::U]/ancestor::W
5. /descendant::U[ancestor::Y[child::Z and /child::V]/parent::W/ancestor::X]

Custom XML Generator
We developed a custom XML generator that generates an XML document based on the queries generated by Custom XPath Generator (YALXP). For each query a minimal XML document having elements, which conform to the relationships between different node tests within the XPath query, is generated. All the minimal documents generated in this fashion are concatenated and enclosed within a super element to generate the final XML document for multiple queries. This generated document has total matches for at least 9 out of 10 queries.

Figure 5.18: Sample query and doc generated from the same
Results

We define commonality factor (CF) as a measure to indicate the degree of commonality among multiple queries with forward and backward axes. This factor gives some hint about the amount of reduction in the number of matching structures created and the number of traversals (both through dag and matching structures) by creating c-dag.

\[ CF(N) = 1 - \frac{Tcn + Tce}{Tcx + Txe} = 1 - \frac{Tcn + Tce}{7N + 6N + Tb}, \]

where \( N \) is number of queries, \( Tcn \) is Total no of c-nodes in c-dag, \( Tee \) is Total no of edges in c-dag, \( Tcn \) is Total No of x- nodes in \( N \) x-dags, \( Txe \) is Total no of edges in \( N \) x-dags and \( Th \) is Total No of Backward axes in \( N \) x-dags.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( Tb )</th>
<th>( Tcn )</th>
<th>( Tee )</th>
<th>( CF(N) )</th>
<th>Time to build c-dag (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>55</td>
<td>190</td>
<td>239</td>
<td>0.683</td>
<td>&lt;1</td>
</tr>
<tr>
<td>200</td>
<td>97</td>
<td>308</td>
<td>393</td>
<td>0.74</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>153</td>
<td>358</td>
<td>482</td>
<td>0.793</td>
<td>1</td>
</tr>
<tr>
<td>400</td>
<td>206</td>
<td>426</td>
<td>589</td>
<td>0.812</td>
<td>2</td>
</tr>
<tr>
<td>500</td>
<td>267</td>
<td>486</td>
<td>698</td>
<td>0.825</td>
<td>3</td>
</tr>
<tr>
<td>600</td>
<td>312</td>
<td>543</td>
<td>794</td>
<td>0.835</td>
<td>4</td>
</tr>
<tr>
<td>700</td>
<td>355</td>
<td>572</td>
<td>873</td>
<td>0.847</td>
<td>6</td>
</tr>
<tr>
<td>800</td>
<td>401</td>
<td>627</td>
<td>964</td>
<td>0.853</td>
<td>8</td>
</tr>
<tr>
<td>900</td>
<td>447</td>
<td>686</td>
<td>1046</td>
<td>0.857</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>501</td>
<td>743</td>
<td>1130</td>
<td>0.861</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 5.19: Graph for the results in Table 5.4
Comparisons between YALXP, XAOS (MSP), Xalan (ODP) and Xalan (MDP)

In our experiments, we observed that Xalan (MDP) takes the highest amount of time due to construction of in-memory DOM and querying the same for each query individually. We also observed that as the number of queries grows, due to the parsing cost involved for each query and the traversal through all the matching structures for each query, XAOS (MSP) takes more time. By adapting the approach of XAOS for multiple queries, YALXP takes lesser time. This is because parsing cost comes down heavily by virtue of single document-order traversal in YALXP. However, the memory consumption of YALXP grows as the number of relevant elements in the input XML document grows due to creation of matching structures for multiple queries. But, this approach by exploiting commonalities among XPath queries with backward and forward axes, works much better in scenarios where the number of relevant elements is likely to be less and the document size is big as is evident from results. We found that Xalan (ODP) performs closer to YALXP than others in CPU time. (Refer to table 5.5)

Table 5.5 Comparison of CPU & Memory usage of various algorithms on docs of sizes given in Table 5.6

<table>
<thead>
<tr>
<th>No of queries</th>
<th>YALXP</th>
<th>XAOS (MSP)</th>
<th>Xalan (ODP)</th>
<th>Xalan (MDP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>Mem</td>
<td>CPU</td>
<td>Mem</td>
</tr>
<tr>
<td>100</td>
<td>29s</td>
<td>26.5M</td>
<td>81s</td>
<td>9M</td>
</tr>
<tr>
<td>200</td>
<td>115s</td>
<td>73M</td>
<td>411s</td>
<td>12M</td>
</tr>
<tr>
<td>300</td>
<td>426s</td>
<td>148M</td>
<td>1172s</td>
<td>13.8M</td>
</tr>
<tr>
<td>400</td>
<td>723s</td>
<td>25.3M</td>
<td>1818s</td>
<td>13.5M</td>
</tr>
<tr>
<td>500</td>
<td>1429s</td>
<td>569M</td>
<td>3177s</td>
<td>18.15M</td>
</tr>
</tbody>
</table>

Table 5.6 Sizes of input documents for algorithms in table 5.5

<table>
<thead>
<tr>
<th>No of queries</th>
<th>Doc. Size (No of elems)</th>
<th>MX-Stream -Size*</th>
<th>XAOS(MSP) -Size*</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>511K(15906)</td>
<td>4452</td>
<td>512</td>
</tr>
<tr>
<td>200</td>
<td>1.04 M (35066)</td>
<td>13195</td>
<td>1129</td>
</tr>
<tr>
<td>300</td>
<td>1.59M (53348)</td>
<td>24382</td>
<td>1899</td>
</tr>
<tr>
<td>400</td>
<td>2.12M (70669)</td>
<td>36635</td>
<td>2383</td>
</tr>
<tr>
<td>500</td>
<td>2.68M (89349)</td>
<td>50486</td>
<td>3084</td>
</tr>
</tbody>
</table>

*Size - Combined Matching Structure (YALXP) / Max size of Matching Structure for a query (XAOS (MSP))
We also implemented XAOS (OSPCN) with One SAX Parsing and Commonalities among queries Not exploited. We observed that, though the processing time is in between that of YALXP and Xalan (ODP), the memory consumption is prohibitively large from 300 queries on. So we excluded the same from comparison in our experiments.

To see the performance of our algorithm on large documents, we compared Xalan (ODP) and YALXP (can be imagined as XAOS with one SAX parsing and commonalities exploited) on a document of size around 10M with 15,00,000 elements (elements without any attributes, text etc) for different no of queries. (This document has the same number of relevant elements as that for smaller documents but the number of
irrelevant elements is large. The **number** of events fired is equal to twice the number of elements in the document in this case. Even in this case YALXP performs better than Xalan (ODP) in processing time (figure 5.15). We also found that the memory required for Xalan (ODP) is far higher than that for YALXP up to 400 queries. More over, the input document in these comparisons has the property that no irrelevant element has an *ancestor* that is relevant (except Root). Otherwise, the time taken by Xalan (ODP) would be even worse.

Xalan (ODP) failed to complete on documents with more than 52,00,000 elements (size 27M) due to thrashing of Operating System. From these experiments we can conclude that the memory consumption of YALXP depends only on the number of relevant elements and on number of queries, but not on document size. The processing time taken by YALXP increases for large documents due to the minimum processing required at each irrelevant event. We did not compare XAOS (MSP) and Xalan (MDP) on documents of 10 M for the obvious reason that processing time goes out of comparison due to multiple parsing.

In cases where the amount of memory available is less and the number of queries is high, YALXP can be run on a subset of queries at a time and it still performs much better in terms of processing time. We did not compare YALXP with other multiple XPath processing engines such as Xfilter[82], Yfilter[83], Xtrie and Mtrie etc due to the fundamental difference in the fragment of XPath queries we address. Yfilter[87], Xtrie[86] cannot be applied to queries with backward axes. Our main aim is to exploit commonality among queries with both forward and backward axes.

5.5 **Conclusions**

We presented YALXP for multiple queries that have forward and backward axes for streaming XML data with the following advantages.

1. Ability to handle backward axes on streaming XML.
2. Evaluation of multiple queries in single document-order traversal.
3. Exploiting commonality among different queries with forward and backward axes thereby sharing processing and memory among different queries.

Our experiments show that YALXP performs better than Xalan(ODP) for multiple queries with single DOM construction.