Efficient Evaluation of Regular Path Expressions on Streaming XML Data

Zachary G. Ives* Alon Y. Levy Daniel S. Weld

{zives, alon, weld}@cs.washington.edu

University of Washington
Seattle, WA USA

Abstract

The adoption of XML promises to accelerate construction of systems that integrate distributed, heterogeneous data. Query languages for XML are typically based on *regular path expressions* that traverse the logical XML graph structure; the efficient evaluation of such path expressions is central to good query processing performance. Most existing XML query processing systems convert XML documents to an internal representation, generally a set of tables or objects; path expressions are evaluated using either index structures or join operations across the tables or objects. Unfortunately, the required index creation or join operations are often costly even with locally stored data, and they are especially expensive in the data integration domain, where the system reads data streamed from remote sources across a network, and seldom reuses results for subsequent queries.

This paper presents the *x-scan* operator which efficiently processes non-materialized XML data as it is being received by the data integration system. X-scan matches regular path expression patterns from the query, returning results in pipelined fashion as the data streams across the network. We experimentally demonstrate the benefits of the x-scan operator versus the approaches used in current systems, and we analyze the algorithm's performance and scalability across a range of XML document types and queries.

1 Introduction

XML, the eXtensible Markup Language standard from the World Wide Web Consortium [XML98], is increasingly being used as a protocol for the dissemination and exchange of information from all types of data sources and applications. XML is quickly becoming the *lingua franca* for data exchange, and nearly every vendor of data management tools has been racing to adopt it. The strengths of XML lie in its simplicity, self-describing nature, and flexibility — particularly in its ability to represent a graph structure, which allows it to encode both structured and semi-structured data.

An XML document (see Figure 1 for an example) consists of pairs of matching open- and close-tags (elements), each of which may enclose additional elements or data values (in the form of "character data" strings). Additionally, an element tag may include attributes further describing the element; attributes are single-valued and may have special meaning (*e.g.*, they may serve as

*Supported in part by an IBM Research Fellowship

```
<db>
 <lab ID="baselab" manager="smith1">
  <name>Seattle Bio Lab</name>
  <location>
    <city>Seattle</city>
    <country>USA</country>
  </location>
 </lab>
 <lab ID="lab2">
  <name>PMBL</name>
  <city>Philadelphia</city>
  <country>USA</country>
 </lab>
 <paper ID="Smith991231" source="baselab"
                          biologist="smith1">
  <title>Autocatalysis of Spectral...</title>
  ...
 </paper>
 <biologist ID="smith1">
   <lastname>Smith</lastname>
   ...
 </biologist>
</db>
```

Figure 1: Sample XML document representing biology labs and publications

element identifiers or references). In particular, XML elements may have special `ID` and `IDREF` attributes, which serve to uniquely identify elements and to form links to them, respectively. This linking capability allows XML to represent not only tree-structured hierarchical data, but also graph-structured information.

Several query languages have been proposed for XML [RLS98, DFF+99, CCD+98, GMW99]. Since these languages treat XML data as a graph, variables in the query are mapped to XML elements, which are nodes in the graph. The main paradigm underlying these languages is that of selecting data by matching patterns described with *regular path expressions* against the XML source. These path expressions describe traversals along subelement, attribute, and `IDREF` edges, and variables get bound to nodes along these paths. Hence, a key operation in query processing over XML is to produce a set of bindings for variables, given a pattern consisting of several regular path expressions.

To date, most efforts to build XML query processors have been based on first loading the data into a local repository, building indexes on the repository, and then processing the query. The approaches differ on whether the repository is a relational database [FK99, SGT+99], an object-oriented database [vZAW99, LAW98] or a repository for semi-structured data [GMW99].

In many applications involving XML, however, we must be able to process queries over streams of incoming XML data, without having the luxury of first loading the data into a local repository. In particular, data integration applications often involve processing data over sources on a wide-area network whose contents change continuously, and hence storing the data locally is not a viable approach. Furthermore, it is imperative that we produce results incrementally as the data streams into the system, since queries are usually ad-hoc and interactive.

In this paper we describe *XML-Scan*, or *x-scan*, an operator that is used at the lowest level of an XML query plan and supplies data to other operators. The input to x-scan is an XML data stream and a set of regular path expressions occurring in a query; x-scan's output is a stream of bindings for the variables occuring in the expressions. A key feature of x-scan is that it produces these bindings incrementally, as the XML data is streaming in; hence, x-scan fits naturally as the source operator to a complex pipeline, and it is highly suited for data integration applications.

X-scan is motivated by the observation that `IDREF` links are limited to the scope of the current document, so in principle, the entire XML query graph for a document could be constructed in a single pass. X-scan achieves this by simultaneously parsing the XML data, indexing nodes by their `ID`s, resolving `IDREF`s, and returning the nodes that match the path expressions of the query. The key challenges involved in designing x-scan stem from need to (1) deal with possibly cyclic data, (2) preserve order of elements, and (3) remove duplicate bindings that are generated when multiple paths lead to the same data elements. We present a series of experiments to evaluate x-scan's

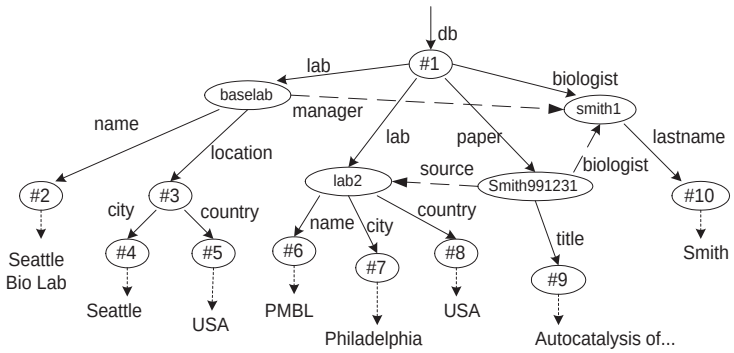


Figure 2: XML-QL graph representation for Figure 1. Dashed edges represent `IDREF`s; dotted edges represent `PCDATA`.

performance. The experiments show that the algorithm scales very well to handle XML files of significant sizes (*e.g.*, up to 14MB). An experimental comparison of x-scan with two systems (Lore and a commerical XML query processor based on an object-oriented repository) shows that x-scan significantly outperforms both of them — sometimes even when the expensive loading time of the other systems is ignored.

The organization of this paper is as follows. Section 2 provides a context for the path expression evaluation problem by reviewing how XML is queried. Section 3 presents the x-scan algorithm and its components, and Section 4 describes our experimental results. Section 5 discusses how the x-scan operator relates to previous work. Finally, we conclude in Section 6 and suggest avenues of future research.

2 A Data Model and Query Language

We begin by briefly discussing the issues in choosing an XML data model and XML-QL, the language we use for querying XML.

2.1 Data Model for XML

Several proposals have been made for data models for XML. They are all based on representing XML as a graph, and differ on whether they consider the order of the XML document, whether they distinguish between subelement edges and attribute edges, and how they represent `IDREF`s in the graph. In our discussion, we represent XML data as a graph, where each XML tag is an edge (labeled with the tag name) that is directed towards a node (with a label equal to the tag's `ID`)[1]. A given element node will have labeled edges directed to its attribute values, sub-elements, and any other elements that it references via `IDREF` attributes. Figure 2 shows the graph representation for the sample XML data of Figure 1. Note that `IDREF`s are shown in the graph as dashed lines and are represented as edges labeled with the `IDREF` attribute name; these edges are directed to the referenced element's node. In order to allow for intermixing of "parsed character" (string) data and nested elements within each element, we create a `PCDATA` edge to each string embedded in the XML document. These edges are represented in Figure 2 as dotted arrows pointing to leaf nodes.

In this paper we consider execution over an ordered XML graph, following the established semantics of processing order in XML. There are certain cases in which XML ordering semantics

[1]This data model is derivative of the XML-QL model, but treats both elements and attributes as edges

```
WHERE <db>
        <lab>
             <name>$n</>
             <_*><city>$c</></>
        </> ELEMENT_AS $l
      </>
      IN "fig1.xml"
CONSTRUCT <result>
            <center> <name> $n </>
                     <location> $c </>
            </>
          </>
```

Figure 3: XML-QL query that finds the locations of labs. The `WHERE` clause specifies a graph-structured pattern of nested tags. Variables are prefixed with a dollar sign, underscore denotes a wildcard which matches any element or attribute, and asterisk is the Kleene star meaning "zero or more."

are undefined or ambiguous (*e.g.* how data from different sources should be ordered when it is combined); we do not attempt to address these issues. Moreover, we observe that XML considers subelements to be ordered but attributes to be order-free; in our model, we *preserve* order across both attributes and subelements, but only allow queries to express ordering *constraints* among subelements.

2.2 Querying XML

A variety of XML query languages have been proposed, mostly based on languages for querying semi-structured data (XQL [RLS98], XML-QL [DFF+99], XML-GL [CCD+98], Lorel [GMW99]). These languages are driving the current W3C Query Language Committee whose final recommendation is likely to encompass features from each. The key features these languages have in common is that they enable a user to match *regular path expressions* over the data, and, to varying extents, have the ability to construct XML documents as a result of the query. In this paper we use XML-QL, but the features of the language that are relevant to our algorithm are mostly found in the other languages as well.

XML-QL uses a `WHERE` *pattern1* `IN` *source1*, *pattern2* `IN` *source2*, ... `CONSTRUCT` *result* syntax, in which the *pattern* template is matched against the input XML data graph from *source* (a URI) and the *result* defines the desired structure of the query output graph. An XML-QL pattern is expressed as a set of nested tags with embedded variable names (prefixed by leading dollar-signs) that specify *bindings* of graph nodes to variables. Continuing the example of Figures 1 and 2, we can issue the query of Figure 3, which returns a list of lab names and their city locations.

More precisely, this query searches for all `lab` elements which are immediately inside a `db` element, with a child `name` element and a descendent `city` element. The query's `CONSTRUCT` clause returns a set of name/city pairs. Note that in XML-QL, we can abbreviate each close-tag with a `</>`. The `WHERE` template can be thought of as a set of tree-structured path expressions that get "matched" across the input graph. Each variable name (l, c, and n above) is bound to the matching node at the end of the path. In our example, we take a `db` edge from the document root. From here, we find a `lab` edge and destination; the `ELEMENT_AS` keyword after `lab`'s close-tag causes this destination node to be bound to variable l. Next, a `name` edge is traversed to a node we assign to variable called n. Now, from the same `db` edge traversed earlier, we traverse any number of edges and then a `city` edge, and assign the node to the variable c.

```
<result>
  <center> <name> Seattle Bio Lab </name>
           <location> Seattle </location>
  </center>
  <center> <name> PMBL </name>
           <location> Philadelphia </location>
  </center>
</result>
```

Figure 4: The result of applying the query from Figure 3 to the XML data in Figure 1.

The result of the `WHERE` clause of the query is a set of bindings for every possible combination of path expression matches. Note that for each combination of possible `lab` and `city` edges under a common `lab` node l, that combination of n and c values should be returned; all three variables can be represented as a 3-tuple. In the example, there are two possible binding tuples: $\langle l/\texttt{baselab}, n/\#2, c/\#4\rangle$ and $\langle l/\texttt{lab2}, n/\#6, c/\#7\rangle$. Note that a WHERE clause can consist of several patterns, and each one can be posed over a different document. The result of the WHERE clause in this case would be the join of the binding tuples produced by each of the patterns.

The `CONSTRUCT` clause normally specifies a tree-structured set of edges and nodes to add to the output graph for each tuple of variable bindings. Wherever an input variable appears in the `CONSTRUCT` clause, its associated node is inserted into the output. Additionally, we also "carry forward" all other nodes transitively connected by edges radiating from the original node. In essence, an XML-QL variable bound to an XML graph node always represents not simply the node, but the *entire subgraph* to which the node transitively connects via "forward-pointing" edges. The constructed output for query of Figure 3 over the data in Figure 1 is shown in Figure 4. Note that the outermost (`result`) tag in the `CONSTRUCT` clause only appears once in the output; this is because XML syntax requires a *single* "root" element enclosing all remaining content.

The goal of the X-scan operator is to produce a set of bindings for each pattern in the `WHERE` clause. Hence, the x-scan operator is the bottommost operator in a query execution plan, and its results are later fed into other operations such as joins, grouping and aggregation. As was described above, the `WHERE` clause is a hierarchical description of path traversals; we can thus rewrite the XML-QL template in a different form using a more conventional dot-notation:

- E_l = `root."db"."lab"`
- $E_n = E_l$`."name"`
- $E_c = E_l$`._*."city"`

Note that expressions E_n and E_c are expressed in terms of E_l, since they are paths originating from a given l node[2]. This hierarchical relationship occurs very commonly in XML-QL. Sometimes there is an implicit rather than explicit set of dependencies — two XML-QL path expressions that are siblings with a common parent *must* actually both have a common parent path expression, even if an `ELEMENT_AS` keyword is not specified in the query, in order to preserve the correct structural and ordering relationship. If l were not specified in Figure 3, the query plan generator would need to create a temporary variable with the same regular path expression, and would have expressed n and c in terms of the temporary variable.

[2]Recall also that the underscore character `_`, used in E_c denotes wildcard so `_*` means zero or more edges of any type.

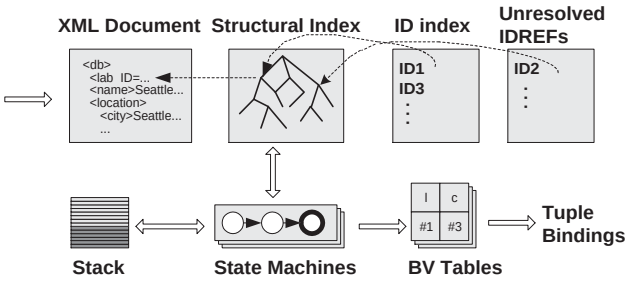


Figure 5: Data structures used by x-scan. The algorithm takes an XML document and generates an index of its structure, keeping track of `ID`s and filling in unresolved `IDREF` targets as they are encountered. Simultaneously, x-scan runs a series of state machines over the graph structure (using a stack to backtrack to previous states) and generates tables of bindings for variables.

3 The X-scan Operator

Given the text stream of an XML document and a set of regular path expressions as inputs, x-scan outputs a stream of tuples assigning binding values to each variable in the set of regular path expressions. The stream of binding values is generated incrementally, and hence x-scan is suitable for inclusion in a pipelined execution plan. The central mechanism underlying the operation of x-scan is a set of state machines that traverse the XML graph, attempting to satisfy the path expressions.

The data components of x-scan are illustrated in Figure 5. As the data streams into the system, we create several structures:

- the data gets parsed and stored locally,
- a structural index of the XML graph is created to facilitate fast traversal across IDREFs through the graph,
- an `ID` index records the `ID`s of all elements and their matching locations in the structural index, and
- a list of references to not-yet-seen element `ID`s is maintained.

In parallel with the construction of these data structures, a set of finite state machines (one per regular path expression/variable) perform a depth-first search over the structural index. When a machine reaches an accept state, a new value is added to the binding-value table associated with the machine. These values are then combined to produce the binding tuples for the query. Each of the state machines also maintains a stack of previously seen bindings along its current path, which is used in order to avoid cycles in traversing the data.

As this section elaborates below, several aspects conspire to make x-scan more complex than a simple application of state-machine searching applied to XML data. First, x-scan operates on possibly cyclic, graph structured data. Second, although x-scan generates tuples as the input XML is streaming into the system, it generates binding tuples in a way that preserves the XML order, when necessary. X-scan includes an optional timestamp component that allows it to prune duplicate bindings (which can be generated when nodes in the XML graph are reachable through multiple paths) incrementally.

Section 3.1 describes the construction of the state machines used by x-scan, and Section 3.2 describes the graph index structure it creates. Section 3.3 describes the operation of the state

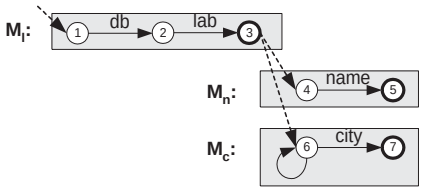


Figure 6: Three state machines (outlined in grey) generated for the path expressions in the XML-QL query of Figure 3. Solid arcs denote state transitions and are labeled with the token required for traversal; the self arc from state 6 is a wildcard and may followed for any token. Dashed arcs denote dependencies between machines, and bold circles signify accept states. Note that, for simplicity, we show non-deterministic finite state machines here, but that x-scan execution actually uses the equivalent deterministic machines.

machines over the data and the production of bindings. Section 3.4 describes how x-scan handles cycles safely, Section 3.6 discusses handling larger-than-memory data sets, and finally, Section 3.5 describes several efficiency enhancements to the algorithm.

3.1 The State Machines

As described in Section 2, we create one regular expression for every variable in the XML-QL query; we refer frequently to the *variable of* a path expression and its inverse, the *expression of* a variable.

The variables in an XML-QL query are typically expressed at different levels in a hierarchical template. We say that variable x is *dependent on* variable y if the expression of x refers to the expression of y, and we say that y is the *parent of* x. In our example, both n and c are dependent on l. Dependencies occur when a query binds one variable (*e.g.*, l) to a node along one path expression, and then binds another variable (*e.g.*, c) to a node that at the end of a specified path *from* the first variable. X-scan must first find a binding for l before searching for bindings for n and c.

Given a set of regular path expressions, we build a finite-state machine for each expression; Figure 6 shows the three machines, M_l, M_n, and M_c, for our example. State transitions in these machines correspond to edge traversals in the XML data graph. The end of the path expression yields an accept state in the machine, which outputs instances of the corresponding variable. The different state machines are related according to the dependencies of the corresponding variables: because c is dependent upon l, machine M_c is dependent on M_l; this means that M_c is only enabled once M_l reaches an accept state. In Figure 6 dependencies are shown as dashed lines.

3.2 Indexing the XML Graph

When x-scan is run on an XML source, it parses the XML and builds a graph-structured index of the data. This index allows x-scan to quickly traverse the XML structure once it has seen some portion of the document, and as a consequence, handle graph-structured data more efficiently. In addition, as we explain below, the construction of the structural index continues even when we need to suspend the state machines because of unresolved `IDREF`s.

Each node in the index contains information about an element (its `ID` and an offset into the original XML data file so that the node's source can be accessed quickly) as well as pointers to all subelements, attributes, and `IDREF`s of the element. Essentially, the index structure looks like the graph of Figure 2 except that data values such as those in the leaf (`PCDATA`) nodes are not stored.

In addition, x-scan creates an index on IDs that it has encountered so far, mapping from ID to entry in the structural index. In addition, an index of all unresolved IDs is maintained, listing all referrers to each unseen ID.

3.3 The Operation of X-Scan

X-scan proceeds by building the structural index and running a set of *active* state machines in parallel. We now focus on the running of the state machines.

The set of active state machines is determined as follows. Initially, only the top-level machine (M_l in our example) is active. When a machine M reaches an accepting state, it produces a binding b for the variable associated with it. It then activates all of its dependent state machines, and they remain active while x-scan is scanning b or any element accessible by a path from b. In our example, the machines M_n and M_c remain active while we scan a given value of l.

Associated with each machine is a table for storing binding values. As a machine reaches an accept state, it writes into this table a tuple containing its bound node value as well as the value of its parent variable (thus providing a means of associating the variable and its parent)[3]. In our example, M_l's table would just store values of l, while n and c would store name and city values, respectively, paired with their corresponding l values. The final output of x-scan is the equi-join of the tables maintained by the three machines.

We illustrate the execution of x-scan on our example. Suppose M_1 is initialized to machine state 1, which takes the XML `root` as binding value. There is one outgoing edge, and because it is labeled `db` x-scan follows it, pushing M_l's old value on the stack and setting M_l to state 2 with value node `#1`. Next x-scan follows the first of four outgoing edges, pushing the old state value, and setting M_l to state 3 with value `baselab`. Since M_l is now in an accepting state, x-scan writes the value `baselab` into M_l's table, suspends M_l, and activates M_n and M_c. The next edge takes M_n from state 4 to 5 while M_c follows the self-arc back to state 6; both machines have `#2` as binding value. Since M_n is now in an accept state, x-scan writes $\langle$`#2`, `baselab`$\rangle$ into M_n's table; note that the current value of l is written along with that of n since l is n's parent. From this node, no (non-`PCDATA`) edges remain for exploration, so x-scan pops the stack and backs up the state machines, resetting M_n to state 4 and M_c to state 6. The next edge is labeled `location` which M_n can't traverse, so it deactivates, while x-scan advances M_c through state `#3` and then into accepting state `#4`. At this point x-scan writes $\langle$`#4`, `baselab`$\rangle$ into M_c's table. X-scan is now able to output its first tuple of bindings: $\langle l/$`baselab`$, n/$`#2`$, c/$`#4`$\rangle$.

X-scan keeps running M_c but no more cities are found, and so eventually it pops back up to `baselab`. X-scan tries running M_c along the `IDREF` to `smith1`, but still no cities are found. So x-scan deactivates M_n and M_c, and control returns to their parent M_l. X-scan pops up to node `#1` and a similar process yields another binding tuple $\langle l/$`lab2`$, n/$`#6`$, c/$`#7`$\rangle$ once M_l finds `lab2`. □

Handling Forward References: On occasion x-scan will encounter an `IDREF` edge which points "ahead" to a node which has not yet been parsed. This situation is easily detected since the `ID` index records all element `ID`s, and the target will not be in the index.

If preserving document order is not important, then x-scan can proceed to process elements out of order, but then the XML query processor will need to do some complex bookkeeping at later stages in order to produce output whose structure (even beyond simply the order) properly

[3]The implementation stores pointers to XML nodes as the values in these tables; this allows x-scan to preserve order in later stages. However, for expository simplicity in the example narrative below, we write as if the node IDs were stored as the values.

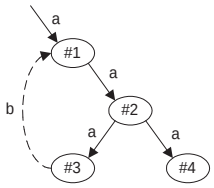


Figure 7: Graph representation for XML data fragment containing a cycle. The dashed edge represents an `IDREF`.

corresponds to the input document. We explain the case of order preservation, which is conceptually simpler and comes at little extra cost.

When x-scan hits a forward reference to an (unseen) element, it pauses all state machines and adds an entry to the list of unresolved `IDREF` symbols, specifying the desired `ID` value and the referrer's address. However, x-scan continues reading the XML source and building the structural index. Once the target element is parsed, x-scan fills its address into each referring `IDREF` in the structural index, removes the entry from the list of unresolved `IDREF`s, and awakens the state machines and proceeds. It is important to note that by continuing to build the structural index, x-scan can process the parsed-but-not-yet-traversed portion of the data much more quickly.

3.4 Handling Cycles Safely

When the input XML document contains cycles, care must be used to ensure that x-scan returns all possible binding tuples without getting trapped in an infinite loop. Consider the XML data of Figure 7, and suppose that the query involves the following path expression:

- E_x = `root._*."b"."a"`

In other words, the query is searching for paths of any length where the last two edges are `b` followed by `a`. A quick inspection of Figure 7 shows that there *is* a match binding x to element `#2`, but the only way to find this match means searching down through element `#1` following `a` to element `#2` continuing on to `3`, and following the `IDREF` back to elements `#1` and `#2` again. If x-scan had refused to follow the cycle and visit these elements again, then it would have missed answers to the query.

On the other hand, if x-scan follows cyclic paths with abandon, it could get trapped in an infinite loop. Consider the behavior of the following path expression on the same XML input:

- E_y = `root._*."z"`

Here, x-scan is directed to look for a path of any length, ending in the token `z`. Quick inspection shows that there aren't any `z`'s but we must ensure that x-scan doesn't run around the cycle endlessly looking for one.

The solution is based on checking the stack associated with the state machine. The stack contains pairs of the form (binding, state), describing which bindings have been associated with states of the machine along the current path. When a machine enters a state, it checks to see that this state has not been bound to the same binding along the current path. Since x-scan uses deterministic finite state machines, we know that returning to a previous state will not add any new possible actions.

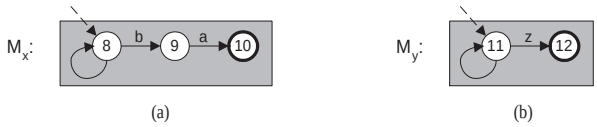


Figure 8: State machines for Kleene star queries on cyclic graphs.

```
WHERE <db>
        <lab manager="smith1">
             <name>$n</>
             <_*><city>$c</></>
        </> ELEMENT_AS $l
      </>
      IN "fig1.xml"
CONSTRUCT <result>
             <center><name>$n</>
             <location>$c</></>
          </>
```

Figure 9: XML-QL query with a selection predicate. We only return bindings when there is a `manager` reference with value `smith1`.

Consider how this solution handles the last two examples. The two path expressions yield the state machines shown in Figure 8(a) and (b). When M_x first reaches element `#1`, it binds the node to state 8. Next it follows the self-loop so state 8 binds to `#2`; again it follows the self-loop so state 8 binds to `#3`. But when it follows the `b` edge it traverses into state 9, so this does not count as repetition because state 9 has never bound to element `#1` before. Now when x-scan traverses the `a` edge it binds state 10 to element `#2` and again there is no repetition, so x-scan successfully leads M_x to an accept.

Contrast this with x-scan's behavior on M_y. When x-scan first reaches element `#1`, it binds the element to state 11. X-scan follows M_y's self loop as it traverses to `#2`, which forms the new binding for state 11. Next state 11 gets bound to element `#3`. Then, as x-scan follows the `IDREF` back to `#1`, it attempts to bind M_y's state 11 to `#1` once again, and the duplication check rejects the binding; instead, x-scan forces M_y to backtrack.

We note that this simple duplication check suffices even for more complex path expressions involving multiple, dependent machines. All that is required is for each machine to refuse to bind any state to a particular node more than once along a path.

3.5 Performance Enhancements

The x-scan implementation includes several optimizations that improve performance: selection push-down, and incremental duplicate elimination.

3.5.1 Selection Push-Down

X-scan can perform a fairly substantial amount of work in evaluating path expressions, so, wherever possible, it is important to prevent the operator from spending time evaluating paths that are not useful in the query's output. We thus allow the query optimizer to push selection operators down into the x-scan operation.

Suppose, for instance, that the query of Figure 3 is modified slightly, as in Figure 9. Note the presence of the constraint that the `lab` must have a `manager` attribute (in this case, an `IDREF`, although we are treating it as an attribute rather than a reference edge) with value `smith1`. For this query, the query plan generator must create an additional temporary variable *temp*1 and a regular path expression:

- $E_{temp1} = E_l$`.@"manager"`

where the `@` prefix indicates that `manager` is an attribute rather than an element. The query plan generator also adds a selection predicate E_{temp1} = `"smith1"`.

During x-scan's evaluation of the graph in Figure 1, it will initially bind the `baselab` node to l, activating the machines for n, c, and *temp*1. X-scan evaluates all node attribute edges before subelement edges, so the `ID` and `manager` attributes will be tested against the state machines. In this case, the `manager` attribute exists and indeed has value `smith1`, so x-scan will continue down this portion of the document and bind values for n and c.

For the second `lab`, however, things are slightly different. The `lab2` node has only an `ID` attribute; as x-scan iterates through all attributes of `lab2`, it finds no `manager` attribute to follow for *temp*1. The *temp*1 path expression cannot be satisfied, so x-scan can "short-circuit" on this subgraph, discarding the value for l and ignoring its children.

Note that a pushed-down selection operator on a subelement (rather than an attribute) might not always allow x-scan's state machine evaluation to short-circuit. The reason is simple: x-scan evaluates each subelement successively in document order, and it will not be able to determine whether a particular subelement does or does not exist until it has processed *all* subelements.

3.5.2 Incremental Duplicate Elimination

When the XML data graph contains `IDREF`s, x-scan may visit an element multiple times through different paths. An unfortunate result of this is that it might generate duplicate binding tuples, which does not follow XML-QL's semantics. To see how duplicate bindings can occur, consider x-scan's behavior on the paper's first sample XML data (Figure 2) with the following path expressions:

- E_z = `root._*.("lab" | "source")`
- $E_n = E_z$`."name"`
- $E_c = E_z$`."city"`

Since E_z will find multiple paths to the element `lab2`, x-scan will produce the following binding tuple twice: $\langle z/$`lab2`$, n/$`#6`$, c/$`#7`$\rangle$.

There are two methods of solving this particular problem, and one must be selected by the query optimizer based on cost or other heuristics. The first method is obvious (but often highly effective): post-process the output tuples, removing duplicates. This can be done with either a sorting or hashing scheme. This approach does not typically require that we keep an entire history of tuples, as we might with a relational table, because the tuples are produced with a grouping based on the hierarchy of the regular expressions. In particular, the above query will produce all of the tuples for a given z value before producing the tuples for successive values of z, so the post-processing stage can flush its history on each new value of z.

There are cases where doing duplicate removal *within* x-scan is beneficial. If a particular path expression binds to a particular node multiple times, and it has expensive dependent path

expressions (*e.g.* path expressions with Kleene-star components), we might want to avoid generating duplicates. In order to do this without requiring large in-memory histories, x-scan annotates the structural index to track when a node was last visited. For each variable for which x-scan is to perform duplicate elimination, it reserves space in the structural index for a timestamp; it also gives every state machine an internal "clock."

Each time x-scan binds a variable to a node, it annotates that node's index entry with the variable's clock time. It then advances the clocks of any dependent variables by one tick. Variables can only bind to nodes with timestamps older than their internal clocks. The result is that for each binding of a "parent" variable, we will only see at most one binding per dependent variable to a given node. This mirrors XML-QL semantics, which allow multiple variables to bind to the same node, but do not allow duplicate tuples to be produced.

3.6 Handling Large XML Documents

In processing a large XML data stream, main memory may not be large enough to handle all of the index structures; this section explains how the x-scan implementation supports larger-than-memory execution.

The approach to handling very large XML documents is to allow paging of the XML source document and of the structural index. Index entries include a field referencing their corresponding elements in the source document, and a series of subelement and `IDREF` edge "links" to other entries within the index itself. With both of these structures, a conventional buffer manager using LRU or some similar policy is sufficient.

There are three auxiliary data structures that are perhaps most naturally kept in memory, namely the `ID` lookup index, the list of unresolved `IDREF` targets, and the state machine stack. The `ID` lookup index is undoubtedly most efficient as a hash table from `ID`s to addresses. However, if this data structure runs out of memory, we may wish to switch to a paged data structure, either a B+-tree or a multilevel hash table. The B+-tree has the property that it is sorted, but it is unclear that this ordering will typically match the order of appearance of `IDREF`s; thus a paged hash-based structure may be a good alternative. A similar approach can be taken with the list of unresolved `IDREF` targets, although such an approach would be more costly since x-scan need to consult this list whenever it finds a new `ID`. Fortunately, this data structure is much less likely to exceed memory, since items are removed as they are resolved.

The number of states in the state machine stack is bounded by the product of the number of variables and the longest non-repeating path. This is a worst-case number in which all state machines are simultaneously active and they all match the edges in our path; typically this is not the case, and we do not need to store the state of an inactive machine. Even if this stack does get very large, it can be very naturally paged to disk, as we can simply swap out the oldest entries to make more room, and re-fetch them as entries get popped off.

4 Experimental Results

Our X-scan implementation uses the IBM XML4C parser version 3.0.1 (based on the Apache Xerces-C library) to parse XML documents. We use the SAX [SAX98] parser API, which provides callbacks to our code as elements are read and allows us to evaluate streaming XML data without first having to build an entire in-memory parse tree.

We have implemented x-scan within the Tukwila [IFF+99] data integration system, which we are extending to support XML queries. Tukwila supports large data sources via paging, and

our implementation of x-scan leverages these capabilities to support larger-than-memory XML documents and structural indices. In our current version, the number of elements with IDs is constrained by an in-memory hash table; in the future, we plan to replace the hash table with a B+-tree to fully support out-of-memory execution.

4.1 Comparison to Current Systems

To the best of our knowledge, x-scan is the first algorithm developed for computing regular path expressions in a data integration context. As such, there is no "fair" system to compete against — however, in order to get an idea for how it fares against previous work, we ran a series of experiments against current XML repository systems. We examined the performance of x-scan, which processes the data incrementally as it parses, versus a conventional store-then-query approach. This experiment was performed with locally stored XML files, and thus it does not show the additional performance benefits of x-scan's ability to incrementally evaluate path expressions as data is slowly streaming into the system; the other systems cannot begin producing results until the XML document has been fully read from the network and then loaded into their proprietary storage formats. On the other hand, x-scan is merely the first component of a query processing system that is under construction, so its numbers do not include the (typically small, especially for the simple queries we used) overhead required by the competing systems to parse and optimize input queries.

We compared the performance of x-scan, Stanford's Lore [GMW99] semi-structured/XML database system, and a commercial OO-based XML repository, across a number of different document sizes and query complexities. Note that the capabilities of the three systems are somewhat different. The commercial XML repository is based on the XQL query language, which is tree-structured in nature, and its capabilities for traversing `IDREF`s are not efficient. Lore supports a graph structured data model with its Lorel query language; however, a Lorel query on an XML document may result in non-XML-compliant output if the result is not strictly a tree. Lore supports an indexing structure called a DataGuide [GW97] that can speed path expression evaluation, but index creation failed on our data sets[4], so we were unable to take advantage of this optimization. Our current x-scan implementation does not support selection predicates, so all queries are simple path expression evaluations over the entire data set.

All x-scan and commercial repository queries were performed on a single-processor 450MHz Pentium II machine running Windows NT with 256MB of memory. The Lore queries were run on a similarly configured 450MHz Pentium II running Linux, using Diet Lore 5.0. All queries were run 7 times and their results were averaged.

We obtained a number of XML documents from the web, including religious texts, Shakespeare's plays, the Mondial geographical encyclopedia, and database publication information from DBLP concerning the VLDB conference. Most of these documents were strictly tree-structured, except for Mondial (which has numerous references) and VLDB (which has references from papers to their proceedings). Table 1 summarizes the queries and data sources used.

Figure 10 displays the results. The x-scan bars are separated into two components, lower portion showing the overhead of the parser and the Tukwila XML document paging system, and the upper showing the additional cost of evaluating the query path expressions using x-scan. In the first 5 data sets, which are all tree-structured, the overhead of parsing dominates the costs of performing node bindings. For the graph-structured data sets, Mondial and VLDB, we see the x-scan costs increase as the path expressions must now be evaluated repeatedly across referenced portions of the graph.

[4]Note that the use of DataGuides is unlikely to speed up Lore's overall performance, as the savings in query processing time would probably be negated by the index creation time.

Query	*Data size*	*Description*
Henry_VI-q1	646 KB	Shakespeare's Henry VI title, personae, speakers
Henry_VI-q2	646 KB	Shakespeare's Henry VI title, personae, lines
Quran	898 KB	Sura titles, epigraphs, verses from Quran
NT	1023 KB	Book and chapter titles from New Testament
Mormon	1510 KB	Book of Mormon preface headings, J. Smith's signed witnesses
Mondial	1332 KB	Mondial encyclopedia countries, cities, cities' ref'd loc. names
VLDB	1558 KB	VLDB paper authors, titles, proceedings' ISBN numbers

Table 1: Queries and data sources used in the experiment comparing x-scan to Lore and a commercial system (Figure 10). See the Appendix for the actual queries and regular path expressions.

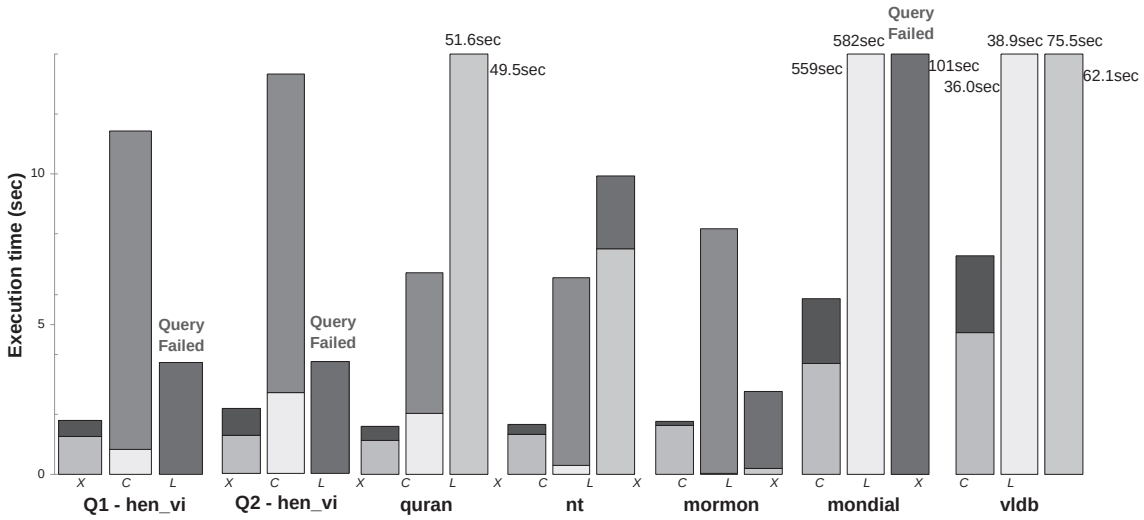


Figure 10: Comparison of query performance. For X-scan, (X) the light bar represents parsing and storage overhead, and the dark bar is state machine and binding costs. For Commercial system (C) and Lore (L), the light bar represents query costs, and upper bar is the cost of loading the document into the repository.

For the existing systems, we differentiate the actual XML query cost from the cost of loading the document into the repository. In a non-data integration context, the cost of a load can be amortized across multiple queries, but in the data integration context this is not possible because we reread data on every query. Both Lore and the commercial system gave very quick query responses to the Mormon query, which only asked for a very small portion of the overall XML document; but their load costs were higher than the execution times for x-scan. For the other queries in the first 5 data sets, we find that Lore generally has significantly better load times than the commercial system, but the commercial system has faster query times, and performs better overall. Lore was unable to complete either query on the Henry_VI text within our time limit of 1000 sec.

The graph-structured Mondial data set was also a problem for Lore, which failed in querying it. We attempted to simulate the traversal of `IDREF`s in this query with the commercial system by using XQL's `id` lookup facilities, but performance suffered greatly. For the VLDB data set, we simplified the query for Lore and the commercial product, simply asking for the value of the papers' `IDREF`s, rather than looking up this value to get the ISBN (which we retrieved in the corresponding x-scan query). Running times were still much higher than those for x-scan.

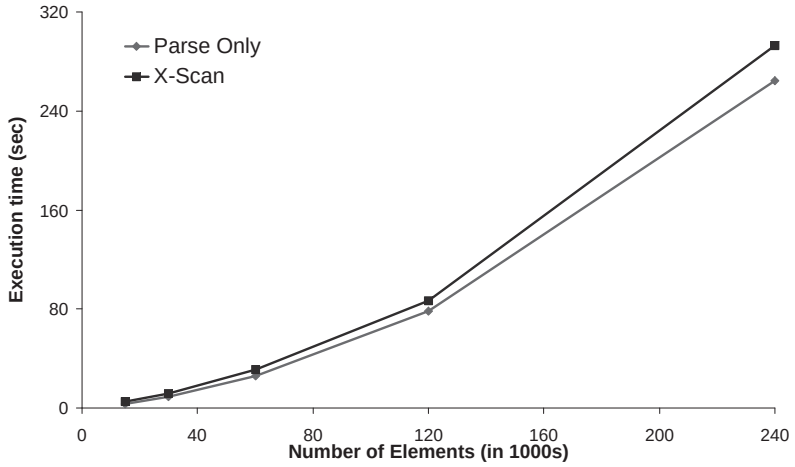


Figure 11: Scalability results for a query over an XML tree. X-scan has minimal overhead over the parse, and grows only approximately 8% faster, even when the document exceeds memory.

We can conclude from this section that neither Lore nor the commercial system scale up well to queries across multi-megabyte data files, particularly files that contain graph structure. X-scan outperforms them in all cases, and also scales better (particularly for tree-structured documents).

4.2 Scalability

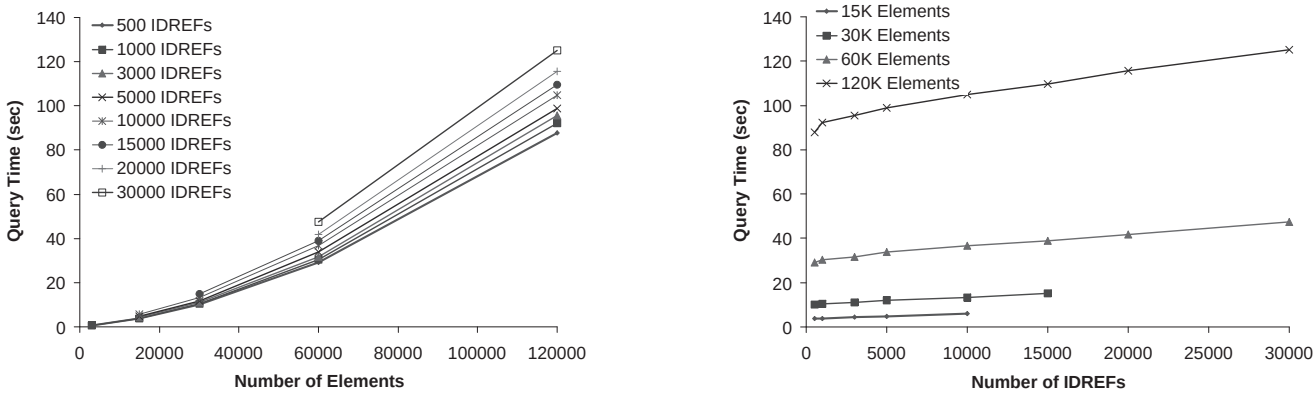
In order to better gauge the scalability of x-scan, we ran our system on a series of synthetic XML data files created by a random XML graph generator. The random graph generator starts with a small XML tree-shaped "template" and begins replicating this to form an irregular XML tree; with 75% probability it adds this template as a subtree of the graph root, and with 25% probability it picks a random node as a parent element. The result is an XML document consisting of subtrees of varying depth, with a large number of children off the root. Next, the graph generator begins randomly adding a specified number of `IDREF` edges between nodes to transform the tree into a graph.

The final graph consists of a root node with a series of `outer` subelement edges emanating from it. At the ends of these edges, there are nodes with some random number of `child` edges (both subelement and `IDREF`), emanating. The `child` edges' destination nodes may source additional `child` edges, and they may be the origin for `sub` edges that point to character data. Most of our queries in this section will be "searching" for these `sub` edges' destination nodes.

Since it is possible for the random graph generator to produce graphs that are unusually favorable or unfavorable to the experiments, we average three different runs across each of three different random graphs of the same generation parameters. (In practice, we found very little difference in performance across the different randomly generated graphs of the same specifications.)

4.2.1 Performance on trees

Since x-scan is applicable to both tree-structured and graph-structured data, we shall first examine how it performs on documents *without* `ID`s and `IDREF`s. For reasons of consistency with later experiments, we actually used the same graph-structured XML files output by the random graph



(a) Time vs. total elements

(b) Time vs. `IDREF` edges

Figure 12: Scalability results for query of Figure 11, but constructing graph index of document.

generator, but replaced their default DTD with one that defines all attributes to be character data rather than `ID`s and `IDREF`s. We also directed x-scan to not generate a structural index of each XML document's data graph, since such an index provides no benefits for tree-based regular path expression evaluation[5]. In this experiment, x-scan simply uses the state machines to generate bindings, which it returns as pipelined tuples; the structural index and the `ID`/reference resolution components are disabled.

Our query was a simple path expression that returned all `outer` subelements as values for the first variable, plus all nodes at distances 1 and 2 from those nodes in the second variable. The graph in Figure 11 shows the results on the different data sets. We can see that x-scan in this case only adds a small amount of overhead versus simply parsing the data file. In every experiment, the x-scan overhead grows at a rate approximately 8% faster than the parser alone, and the actual overhead remains minimal for even the largest of the data sets (which was approximately 14.6MB). Note that for the 240,000-element query, the XML document exceeded x-scan's memory allocation, and was paged to and from disk during operation.

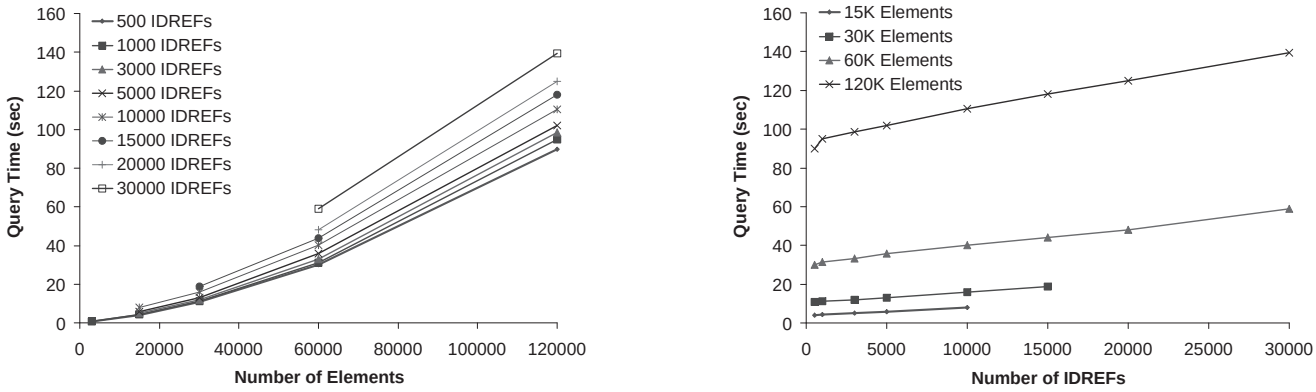
4.2.2 Cost of graph indexing

Our next experiment was to take the same data set and query as in the previous section, but to use the graph DTD. From this we can determine the impact of building the structural index and of resolving references. Figure 12 (a) illustrates query performance vs. number of elements in the document, and (b) shows performance vs. number of `IDREF` edges added.

As one would expect, the running times have moderately increased because of the index generation overhead. Additionally, the amount of time to process a query grows at a slightly superlinear rate in the number of elements, as shown in part (a) of the Figure. (This was also true of both the parser and of x-scan in the previous section, but the rate is slightly more pronounced here.) We attribute this to the additional number of document and index page accesses required for performing "bookkeeping" and storage on increasingly larger XML documents. Even for a 7.5MB XML document, however, our total execution time is approximately 2 minutes, and the operator actually outputs pipelined tuples as it executes.

Part (b) of the Figure demonstrates that query execution time increases linearly with the number

[5]By comparing the result of this experiment with that of the next, we can calculate the cost of building this index.



(a) Time vs. total elements

(b) Time vs. IDREF edges

Figure 13: Scalability results for graph-traversing query requesting outer nodes, their child nodes, and all sub nodes within 1 or 2 edge traversals of the outer nodes.

of IDREF edges. This query does not traverse any IDREF edges, so all costs incurred are for indexing the references.

4.2.3 Graph-traversing query

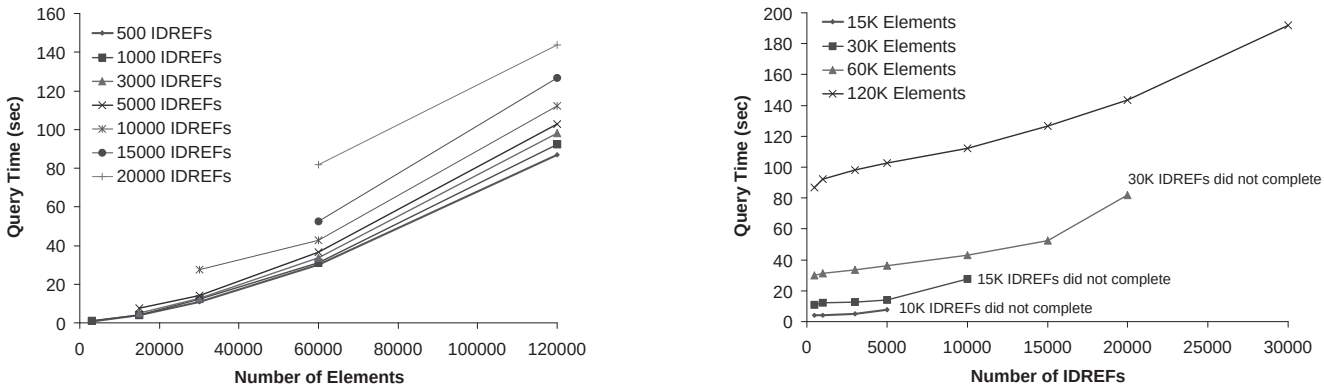
Since much of the complexity of the x-scan algorithm concerns efficient path expression matching not just against trees, but against full graphs with IDREF edges, the next experiment tests the effectiveness of our structural index when called to evaluate such reference edges.

The query we used in this experiment had three variables: the first bound to the outer nodes, the second to child nodes of outer and to the child nodes' sub children, and the third variable to sub nodes either one or two child edges away from the outer nodes. This query returns most of the nodes within radius 2 of the outer nodes.

The results, shown in Figure 13, have nearly identical shapes to the subelement-only query graphs from the previous section. Close examination reveals that the plots in Figure 13(a) run parallel to those in Figure 12, with a slightly higher value at each point. This is small offset is the overhead in traversing the additional references and binding to an additional (third) variable. A comparison of the growth respect to number of IDREFs, in part (b) of each figure, shows that the two queries behave similarly, but as the number of IDREFs increases, the graph-traversing query begins to grow at a slightly faster rate. This is to be expected because the tree-only query did not actually traverse IDREFs.

4.2.4 Kleene-star

Our final experiment measures the costs of evaluating a query that uses a Kleene-star operator to return all sub nodes in the graph. We would expect that this query would be more subject to variation on different random graphs, as certain graphs may have "hub"-like nodes that have many out-edges and multiple in-edges. For such nodes, the path expression evaluation algorithm will re-evaluate the entire subgraph for each incoming edge. If several of these hub nodes are chained together, the number of repeated traversals can grow exponentially. Moreover, a high ratio of IDREF edges to elements in the graph greatly increases the likelihood of such chains appearing.



(a) Time vs. total elements

(b) Time vs. `IDREF` edges

Figure 14: Scalability results for Kleene-star query requesting all `sub` elements reachable from any number of `child` edges.

The graphs in Figure 14 show x-scan performance. In part (a) we see a familiar pattern for execution time versus number of elements, although the actual completion times are slightly longer. Part (b) shows the more interesting results, comparing running times versus number of `IDREF`s present in the document. The growth is now superlinear, generally increasing at successively faster rates as we approach a point in which the number of `IDREF`s reaches 50% of the total number of elements. (We note that at this value, an n-node graph actually has $3n/2 - 1$ edges, since all nodes are subelements.) At the 50% point, the x-scan running times increase to some indeterminately high value; in none of our experiments did such a query manage to complete within an hour.

We believe that XML data with such a high concentration of edges is unlikely to occur often in practice. However, we believe we have a solution that will make processing of such data graphs more tractable. In particular, the problem is that x-scan spends massive amounts of time duplicating previous traversals to get binding values. For this case, we propose "memo-izing" the bindings produced by following out-edges from a particular node, annotating the structural index with pointers to such memo-ized values. Now if x-scan reaches a previously visited node and is in a previously encountered state machine configuration, we can simply read and return the memo-ized results. We have a trade-off in the extra disk accesses required to read and write memo-ized values, but in highly-connected graphs, this will produce a net gain.

5 Related Work

As XML has emerged as a medium for representing data as well as documents, and as query languages for XML have been proposed, a number of approaches have been proposed for evaluating XML queries. Most of these involve mapping XML to an existing database model and utilizing conventional query engine to do the core work. XML mapping strategies for relational [FK99, SGT+99, DFS99], object-oriented [vZAW99, LAW98], and semi-structured [GMW99] databases have all been implemented. The system's particular storage mapping may actually simplify certain path expressions, *e.g.* if a set of path expressions includes multiple data items that are mapped to the same tuple in a table. However, in the general case, indexing techniques such as join indices [Val87], access support relations [KM90], DataGuides [GW97], and t-indices [MS99] must be used to speed the processing of path expressions. These index structures describe the nodes reachable by certain classes of path expressions. The t-index, and to some extent the DataGuide, are particularly

powerful structures that allow efficient computation of a wide range of regular path expression types. However, the actual index generation tends to be fairly complex and time consuming: DataGuides can be exponential in the size of the data and t-indices, while not exponential, are also costly to generate, especially for more complex path expression types.

X-scan differs in three key ways from these techniques. First, x-scan's structural index essentially converts the XML document into a semi-structured format that can be more efficiently traversed without parsing; but it preserves the ordering and locality of the XML document, rather than splitting it into separate tables or objects that must later be re-combined.

Second, x-scan path expression evaluation is done through finite state machines based on the *query*. By contrast, semi-structured index techniques such as DataGuides and t-indices are essentially finite automata describing paths through the *data*, with each state pointing to the set of nodes reachable through a particular path. The benefits of the t-index or DataGuide are that it is a reusable structure, which can be leveraged across multiple queries with different regular path expressions. However, in a data integration context, we re-read data from the source, so reuse does not occur — thus it is more appropriate to build a structure specific to the given query.

Finally, x-scan is a pipelining operator intended for streaming data, whereas other approaches require a costly translation and indexing stage before the query can be executed. This pipelining capability is key in an interactive ad-hoc query system, particularly if the data must be obtained from a slow source [UFA98, IFF+99, AH00].

The x-scan pattern matching approach is similar to the concept behind the Knuth-Morris-Pratt substring-matching algorithm, which creates a finite state machine out of one string and matches it against the other string. However, x-scan must be more sophisticated in order to handle matching of tree-structured regular expression templates across graphs: (1) it supports both "forward" and "reverse" traversals as we encounter open- and close-tags in XML, (2) it handles cycles in a way that prevents infinite loops, (3) it uses multiple dependent machines in conjunction, (4) it supports arbitrary wildcards, disjunction, and Kleene-closure operations in paths, and (5) it has the ability to avoid generating duplicate bindings for nodes reachable by several paths.

The basic goal of x-scan, of converting from semistructured data to tuples in pipelined fashion, is very similar to the *scan* operator proposed by Cluet and Moerkotte in [CM97]. However, x-scan differs in that it handles (cyclical) graphs as well as trees, it is for XML data rather than native object or semistructured data, and it includes an algorithm and implementation.

6 Conclusions and Future Work

In this paper we have presented the x-scan algorithm, a new primitive for XML query processing, that evaluates regular path expressions to produce bindings. X-scan is scalable to larger XML documents than previous approaches and provides important advantages for data integration, with the following contributions:

- X-scan is pipelined and produces bindings as data is being streamed into the system, rather than requiring an initial stage to store and index the data.
- X-scan handles graph-structured data, including cyclical data, by resolving and traversing `IDREF` edges, and it does this following document order and eliminating duplicate bindings.
- X-scan generates an index of the structure of the XML document, while preserving the original XML structure.

- X-scan uses a set of dependent finite state machines to efficiently compute variable bindings as edges are traversed. In contrast to semi-structured indexing techniques, x-scan constructs finite automata for the paths in the *query*, rather than for the paths in the *data*.
- X-scan is very efficient, typically imposing only an 8% overhead on top of the time required to parse the XML document. X-scan scales to handle large XML sources and compares favorably to Lore and a commerical XML repository, sometimes even when the cost of loading data into those systems is ignored.

In the short term, we plan to add several refinements to our x-scan implementation and investigate their effects. As was previously mentioned, we will be adding full support for out-of-memory execution and for selection push-down, and we will also add the ability to memoize intermediate results and avoid redundant computation in highly connected graphs. Additionally, the current algorithm separates the parsing and state-machine traversal components into different stages whose execution must be interleaved. We envision the final implementation putting these stages in separate threads, and to run both in parallel in a producer-consumer arrangement so x-scan can parse and return results completely in parallel.

Additionally, we believe that the two key contributions of x-scan — state machine-based evaluation of regular path expressions and "on-the-fly" indexing of XML — are general techniques that have application beyond our current domain of focus. For instance, XML-QL queries may be composed over other XML-QL views ("functions"); this adds greater expressive power than a single XML-QL query, and thus may require computation of intermediate view results that are fed into the next query or view. As the input to the second query or view, we can use a variation of x-scan that works on graph data rather than "pure" XML. The graph structure index may be useful in a number of other operations. This output can be used as the input to a t-index generator, which we believe will speed the creation of indexing structures for general path expressions on stored XML data. The index may also be useful in constructing the results of an XML-QL query: the semantics of XML-QL specify that if a node is copied from the input data graph to the output graph, we must also copy all nodes that are transitively connected to this node — an XML-QL node potentially represents an entire subgraph. The graph index allows us to quickly find the required XML fragments and copy them over; we might even mark these in the index as having been copied. Finally, we are also considering the use of the structural index to support efficient updates on XML data.

Currently, x-scan represents a new operator which will be at the core of the new version of the Tukwila data integration system [IFF+99]. However, this is just a first step. As we proceed, we plan to investigate the potential uses described above — to develop a family of algorithms derived from or assisted by x-scan.

References

[AH00] Ron Avnur and Joseph M. Hellerstein. Continuous query optimization. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Dallas, TX, 2000.

[CCD+98] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and reshaping XML documents. W3C Query Language Workshop, http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html, December 1998.

[CM97] Sophie Cluet and Guido Moerkotte. Query processing in the schemaless and semistructured context. Unpublished manuscript, 1997.

[DFF+99] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings of the International Word Wide Web Conference, Toronto, CA*, 1999.

[DFS99] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 431–442, 1999.

[FK99] Daniela Florescu and Donald Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, March 1999.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web (WebDB), Philadelphia, PA*, pages 25–30, 1999.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445, 1997.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 299–310, 1999.

[KM90] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 364–374, 1990.

[LAW98] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating structured and semistructured data. Technical report, Stanford University, October 1998.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999*, pages 277–295, 1999.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, December 1998.

[SAX98] SAX 1.0: The simple API for XML. http://www.megginson.com/SAX/index.html, May 1998.

[SGT+99] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland*, pages 302–304, 1999.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, 1998.

[Val87] Patrick Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

[vZAW99] Roelof van Zwol, Peter M.G. Apers, and Annita N. Wilschut. Modelling and querying semistructured data with MOA. In *Proceedings of the Workshop on Semi-Structured Data and Non-Standard Data Formats, Jerusalem, Israel*, 1999.

[XML98] Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, 10 February 1998. World Wide Web Consortium (W3C) Recommendation.

Appendix: Queries Used in Experiments

The tables on the next page present the different queries used in the experimental section of this paper.

Name	*System*	*Query*
Henry_VI-q1	x-scan	p = root."PLAY", t = p."TITLE", per = p."PERSONAE"."PERSONA", a = p."ACT", as = a."SCENE"."SPEECH"."SPEAKER"
	commercial	/play/title | /PLAY/PERSONAE/PERSONA | /PLAY/ACT/SCENE/SPEECH/SPEAKER
	Lore	select t, p, s from PLAY pl, pl.TITLE t, pl.PERSONAE.PERSONA p, pl.ACT.SCENE.SPEECH.SPEAKER s
Henry_VI-q2	x-scan	p = root."PLAY", t = p."TITLE", per = p."PERSONAE"."PERSONA", a = p."ACT", l = a."SCENE"."SPEECH"."LINE"
	commercial	/play/title | /PLAY/PERSONAE/PERSONA | /PLAY/ACT/SCENE/SPEECH/LINE
	Lore	select t, p, l from PLAY pl, pl.TITLE t, pl.PERSONAE.PERSONA p, pl.ACT.SCENE.SPEECH.LINE l
Quran	x-scan	r = root."tstmt"."suracoll", s = r."sura", e = s."epigraph", t = s."bktlong", v = s."v"
	commercial	/tstmt/suracoll/sura/epigraph | /tstmt/suracoll/sura/bktlong | /tstmt/suracoll/sura/v
	Lore	select e,t,v from tstmt.suracoll r, r.sura s, s.epigraph e, s.bktlong t, s.v v
NT	x-scan	b = root.."tstmt"."bookcoll", bk = b."book", t = bk."bktlong", c = bk."chapter"."title"
	commercial	/tstmt/bookcoll/book/bktlong | /tstmt/bookcoll/book/chapter/chtitle
	Lore	select t, c from tstmt.bookcoll.book b, b.bktlong t, b.chapter c
Mormon	x-scan	pref = root."tstmt"._, t = pref."ptitle", w = pref."witlist", per = w."witness"
	commercial	/tstmt/*/ptitle | /tsmt/t/*/witlist | /tstmt/*/witlist/witness
	Lore	select t,w from tstmt.% pref, pref.ptitle t, pref.witlist.witness w
Mondial	x-scan	c = root."mondial"."country", n = c."name", cit = c."city", cn = cit."name", at = cit."located"."ref"."name"
	commercial	/mondial/country/name | /mondial/country/city/name | id(/mondial/country/city/located/@ref)/name
	Lore	select ctry, cit, loc from mondial.country co, co.name ctry, co.city ci, ci.name cit, ci.located.ref.name loc
VLDB	x-scan	i = root."conf"."inproceedings", a = i."author", t = i."title", p = i."crossref"."IDREF"."isbn"
	commercial	/conf/inproceedings/author | /conf/inproceedings/title | /conf/inproceedings/crossref/@IDREF
	Lore	select a, t, r from conf.inproceedings i, i.author a, i.title t, i.crossref.IDREF r

Table 2: Queries used in the experiment comparing systems (Figure 10). X-scan uses a series of path expressions, the commercial system uses XPath/XQL, and Lore uses Lorel.

Section	*Query*
4.2.1	o = root."doc"."outer", s = o~("child"|"child"."sub")
4.2.2	o = root."doc"."outer", s = o~("child"|"child"."sub")
4.2.3	o = root."doc"."outer", s = o.("child"|"child"."sub"), t = o.("child"."sub"|"child"."child"."sub")
4.2.4	o = root."doc"."outer", s = o."child"*."sub"

Table 3: Queries used in the scalability experiments. Note that the tilde (∼) character is a path segment separator much like the dot operator, but it specifies that the next edge is can only be a subelement (as opposed to an `IDREF`).