Optimizing The Lazy DFA Approach for XML Stream Processing

Danny Chen   Raymond K. Wong

School of Computer Science & Engineering
University of New South Wales
NSW 2052, Australia

wong@cse.unsw.edu.au

Abstract

Lazy DFA (Deterministic Finite Automata) approach has been recently proposed to for efficient XML stream data processing. This paper discusses the drawbacks of the approach, suggests several optimizations as solutions, and presents a detailed analysis for the processing model. The experiments show that our proposed approach is indeed effective and scalable.

Keywords: Lazy DFA, Stream Data, XML, XPath

1 Introduction

The problem of processing streaming XML data is gaining widespread attention from the research community, due to the lack of available tools that can efficiently manage large streams of data as well as the increasing popularity of XML information exchange applications. Traditional database processing techniques are not designed to handle the infinite size and unpredictability of the data streams. Often in data stream processing, it is necessary to filter parts of the stream to extract the relevant material for further processing. With XML streams we can take advantage of the structural characteristics and design filters that will incrementally return the results.

This paper presents an analysis of a processing model that uses Deterministic Finite Automata to evaluate XPath queries over XML streams. The analysis included an overview of the various stages in creating the lazy DFA from a set of XPath expressions. The stages included the definition of pattern trees, the construction of the Non Deterministic Finite Automata, the lazy DFA construction, creation of Stream Index, and processing of the XML events. Due to space limitation here, the detail of the lazy DFA approach can be found at [12,14].

Secondly, this paper highlights some of the major concerns regarding the efficiency of the approach. These issues included the NFA construction costs, the handling of predicate filters, the lazy DFA construction cost, and the SIX cost.

Copyright © 2004. Australian Computer Society, Inc. This paper appeared at *The Fifteenth Australasian Database Conference (ADC2004),* Dunedin, New Zealand. Conferences in Research and Practice in Information Technology, Vol. 27. Klaus-Dieter Schewe and Hugh Williams, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

We then propose several optimizations to alleviate problems like the large state counts of the NFA, and the processing overheads of the lazy DFA. These proposals include the use of a DTD parser, pre-filtering of the queries, storing NFA transitions in the DFA states, generalizing transitions, DFA state key generation, and the pull parser.

Finally, the experimental results confirm that the optimized, lazy DFA performs well against many different types of query inputs and is scalable to a large number of queries without major degradation of the processing times.

2 Related Work

There are numerous recent proposals in this field and here we summarize some of the major approaches. XFilter [1] was designed as a selective dissemination of information system for filtering XML documents. Their focus was on building a publish/subscribe system that could register user interests or "profiles" that get matched against incoming XML documents. However it is oriented towards handling moderately small discrete XML documents which makes it less effective against continuous streams of XML. XTrie [4] is a filtering system that was designed to support large scale filtering of streaming XML data. Its basic idea is to use the *trie* to detect occurrences of substring matches for each event that it receives. The time complexity of XTrie makes it slower than the lazy DFA approach. XTrie's time complexity to process one event tag was shown to be $O(PLHL_{max})$ where P is length of the longest path in the *trie,* L the maximum length of the linked list in the substring table, H is the maximum height of the substring tree, and L_{max} is the maximum number of levels in the XML document. X-Scan [16] was designed as an operator to be used in a pipeline fashion to evaluate XPath expressions against XML streams. In fact the architecture is very similar to that of our proposed query processor in that it also uses a variation of the lazy DFA, with the exception that x-scan uses the XML-QL as the query specification language and that x-scan can also handle cyclic XML graphs. The problem with the x-scan approach is that when the XML stream gets really large the structural index can grow infinitely. One recommendation made in the paper was to store parts of it onto disk and allow the x-scan to page disk to locate the indexes. This problem does not apply to the lazy DFA approach since it only handles tree structured XML.

WebFilter [10] is another publish/subscribe system that uses XPath as the query language to filter XML streams. However WebFilter takes a different approach to evaluate the queries and is based on the techniques described in [9]. Instead of getting events from a SAX parser, WebFilter treats an XML document as a set of attribute value pairs and similarly the XPath expressions as a collection of predicate pairs storing the attribute and value as well as some relational operator. An event satisfies a query if every predicate pair in the query matches some attribute value pair in the event. Although the experimental results for the system was not available it would be worthwhile to further investigate this approach and compare its performance against the lazy DFA approach. [11] presents a processing algorithm for streaming XML that is based on algebraic operators. The architecture is designed so that it can be used in "pull based" systems where queries are submitted to the server for processing, as well as "push based" systems, where the server distributes everything to the clients to process themselves. The key idea of the processing model is that instead of sending the entire stream across, they send fragments depending on the popularity or current needs of the client. This means multiple streams can actually be received at the client at any one time (which is different to our approach, which gets an entire stream from start to finish). [13] presents the use of views over XML streams to speed up the processing times. The idea is to augment the query processor, with the ability to utilise views embedded in the stream to aid in the evaluation of the XPath expressions. There are several problems that need to be addressed when using the views. Firstly there is the view selection problem, which is how to select the set of views to maximise the matches for each view. The other problem is when to create the views. Creating them offline requires assumptions about the things like the workloads, network topology, and types of XML streams. Whereas creating them online means the views are chosen according to the current status of the network. Another problem is how to evaluate the queries to find a match in the views as quickly as possible. Although processing streams with views is an attractive option, the problems of using views on streams are relatively hard to solve, Further research is needed in order to justify this approach. Finally, the most related work can be found at [12,14,25] in which the original idea of employing lazy DFA for XML stream data was proposed. Our work is inspired by their result, and tries to analyze and optimize their methods.

3 Issues for the Lazy DFA approach

3.1 NFA construction costs

The DFA query processor design uses the method of state machines to quickly determine matching queries. However this comes at the cost of heavy memory usage due to the large number of states created when converting the collection of queries into a NFA. Since there is virtually no upper bound on the memory usage, the processors performance quickly degrades as it begins to page the hard disk when it is no longer possible to do in memory processing of the state transitions. In order to properly assess the limitations of the approach, there needs to be a space and time analysis at each stage of the algorithm (while most research are biased to the overall time analysis).

[12] omitted the cost of NFA construction from their measurements. However in our experimental results we will demonstrate the NFA construction adds significantly to the overall computational cost, and the majority of the space cost during processing.

Consider an XPath expression with k location steps then the query processor needs to check all k location steps and create an NFA state at each step, giving a total of k NFA states. The time and space complexity for one query is $O(k)$. Now if Q is the set of queries such that $q_i \in Q$ and k_i is the number of location steps for q_i, the total time complexity is $O(\sum k_i)$ since the query processor needs to process all location steps. For a system that is suppose to handle millions of queries then this could be really expensive to compute. Unlike the DFA, which can be computed during processing, it is necessary to have the NFA in memory, in order for the query processor to quickly determine which XPath expression is valid for the received SAX event.

3.2 Predicates in XPath expressions

Another source of time complexity overlooked in the other implementations (except [14]) is that of predicates. Predicates can come in the form of an element position, an attribute comparison, or a branching path. Each type of predicate has a different impact on the space and time complexities of the query processor so we need to consider each in turn. In particular, for the element position predicate, the worst-case scenario is when each location step in the XPath expressions has a position predicate associated with it. This means that in addition to processing the element name the query processor also needs to extract the position value. This extra step is thus done n times which is in addition to the time complexity of $O(n)$ shown previously for processing n location steps, giving a total time complexity of $O(2n)$.

In terms of space complexity we first need to identify location steps with position predicates as being separate location steps. Recall that the space complexity mentioned previously for linear XPaths was at most $O(n)$ where n was the total number of location steps. If there also n position predicates, the total space cost is at most $O(2n)$. The interesting aspect of position predicates is that given a limited number of unique element names, it is possible to write an unlimited number of unique and valid XPath expressions. To illustrate consider the two elements a and b such that

the element b can appear any number of times as a child element of a i.e.

<a>
 …
 …
 …
…

(This appears in the DTD as <element a (b*)> or <element a (b+))

We can thus specify queries that selects any of the b elements nested in a such as

$Q1 = /a/b[1]

$Q2 = /a/b[2]

$Q3 = /a/b[3]

…

Therefore the NFA will have a separate transition for each of the b[i] location steps, where $i > 0$. These transitions all lead to a different terminal NFA state. As you can see even with just two elements and one type of nesting you can specify valid queries using position predicates that leads to any number of NFA states. One possible solution to the unbounded number of NFA states is to have all those b[i] transitions lead to one state. Of course this one state needs to store all mappings for each b[i] transition, its variables $\$Q_i$, and its SaxEvents flag. This is because the single state now represents all the terminal states, and needs to distinguish between each one to return the correct variable event. This brings the number of NFA states back to at most $O(n)$ as before, however there is the extra space cost of storing the map.

Given this assumption we can count these predicates as extra location steps and are treated like any normal location step. Thus the time complexity for processing XPath queries with predicate branches is now $O(n + m)$ where n is the total number of location steps in the set of queries as before and m is the total number of location steps from the predicates. The query processor still needs to check the predicate branch even if the branch specifies a path that has already been created as part of the NFA.

The space complexity on the other hand is dependent on whether these predicate branches specify existing paths. This gives a space complexity of at most $O(n)$ where n is the total number of location steps, including the location steps in the predicate branches. Just as before the space complexity is more likely to be $O(m)$ for m unique location steps in the set of queries.

3.3 Lazy DFA construction costs

There are several factors, which contribute to the DFA construction cost. These include the SAX parser, the transition lookup, the NFA collection, and the DFA lookup.

Firstly the SAX parser's performance can have a significant effect on the overall performance of the query processor, and is the motivation behind the SIX [12]. With a SAX parser we do not need to keep in memory the element tree for the document since the query processor can process each event incrementally as they arrive. Although this makes the query processor slightly more complicated, the efficiency gained in terms of space and time is critically important.

Each DFA state consists of a hash table, mapping the transition from the state to the next DFA state. For each SAX event, the query processor needs to do a hash table lookup to see if there is a transition from the current state. In the eager DFA all valid states are already known so a lookup will indicate immediately whether the next state exists. If there are n SAX events and the cost of each lookup is order $O(1)$ then the eager DFA's total time complexity for transition lookups is $O(n)$. However this time complexity is the best-case scenario, and is dependent on whether the hashing function used can create one key per entry in the hash table.

In the lazy DFA, these hash tables are partially filled so each lookup will determine if it has found an existing DFA state, or that the query processor needs to create a new DFA state. Once again the cost of each hash table lookup is at best $O(1)$ and is dependent on whether the hashing function used, can create one entry for each transition. If a DFA state contains k NFA states then the time complexity of performing all the transition lookups is at best $O(k)$. In addition to this lookup cost there is the cost to add the valid NFA states to the newly created DFA state. This operation occurs after finding an entry in the hash table of the NFA state matching the current event. From this hash table we get the pointer to the next NFA state and add it to the new DFA state. The DFA state maintains a sorted set of the NFA state pointers and insertion times into this set is dependent on the set implementation used.

The main source of memory usage for the DFA is keeping the pointers to the NFA states. In general the DFA has exponentially many states because we can take the powerset of the NFA states and construct the eager DFA. However it has been shown that certain XPath expressions will lead to an exponential expansion of the states for a DFA while others will not. The theorems described in [12] reinforce the fact the eager DFA is far too space intensive and is the motivation for the use of the lazy DFA for processing large scale query sets.

Another issue to consider when using the lazy DFA approach is how to maintain the pointers to the DFA states already created. As mentioned earlier each SAX event involves a lookup in the current DFA state to find a transition, and if it fails to find one then it proceeds to collect the NFA states to create a new DFA state. Two

DFA states are considered the same state if their sets of NFA states are the same. This can occur due to transitions like the //, or * which will match any SAX event and so events following this transition might revisit the same state. The query processor needs to check to see if the newly formed state is an existing state or a new unvisited state. This can be accomplished by searching through a container of previous DFA states.

There are several problems with this approach, the obvious being the unbounded size on this container, as stated previously about the exponential expansion of the DFA. This means the traversal time will depend on the type of container used. A natural choice for the container would be a hash table, which can provide fast retrieval times. However this leads to another problem, which is what do we use as a unique key to represent the DFA states. Since the set of NFA states defines the DFA state then a signature could be calculated based on the NFA state identifiers. This would incur the extra cost of computing this signature and this computation grows linearly with the number of NFA states in the set.

3.4 Cost of using the SIX

Even though the SIX construction takes place at the source of the XML stream we need to measure its construction cost to gain a better picture of the overall requirements of the query processor. In most instances, computing the entire SIX is possible as long as the elements in the XML data is smaller than 2^{32} bytes. It would thus take $O(n)$ time to generate the SIX for an XML document with n elements. However it has been shown in [12] that it is possible to not generate SIX entries for small elements without affecting the integrity and performance of the query processor.

On the query processor end, the SIX Manager has the responsibility of synchronising the stream of SIX entries with the stream of XML elements. The cost of this operation can be divided into several areas. Firstly there is the cost of reading from the input stream into a buffer, each SIX entry that corresponds to the current element event received from the SAX parser. The read from input cost grows linearly with the number of SIX entries.

Next there is the cost of pushing the SIX entry onto the stack every time the query processor receives the SAX event for the start of an element. Given n elements this cost is at most $O(n)$ and occurs when the XPath expression begins with a //. This is because the processor needs to check each element and cannot skip over any elements.

There is also the cost of popping SIX entries off the stack, which occurs when the query processor receives the SAX event for the end of an element. The pop is also performed when the query processor failed to find a matching transition and so needs to skip over the remainder of the element. For the same reasons as the push operation the cost for a pop is at most $O(n)$.

The size of this stack is dependent on both the XML data and the XPath expressions. For example consider an XPath expression without any predicate filters or //, then the size of the stack will be at most the length of the XPath expression. The entries in the stack will correspond to the elements specified at each of the location steps in the XPath expression. For a set of XPath expressions without any predicate filters or //, the size of this stack will be at most equal to the length of the longest XPath expression in the set. For XPath expressions with //, the size of the stack will depend on where in the expression // occurs and the nesting of the actual elements in the XML data.

4 Optimizations

4.1 Pre-filter

The major space complexity of the query processor comes from the size of this NFA. We propose a pre-filtering method that is similar to the one proposed in [1], with the difference that our pre-filtering method is adjustable such that we can pre-filter based on only the start elements, the elements upto a given depth, or all elements. For instance, if we set it to the start elements, it works very well when the set of XPath expressions contain many different starting element nodes for the first location step. It is also possible to have a set of queries such that all of them start with the correct root node for the current XML document. In this case there is no benefit gained since no queries will be filtered out. However the pre-filtering requires no extra storage and the extra checks performed on each XPath is insignificant which can be seen in the experiments in the next section. Thus the pre-filter method proves to be a simple and effective improvement to the lazy DFA approach.

4.2 DTD for pre-filtering

We can check each location step for validity against the DTD. Essentially this is an extension to the pre-filter technique since we're eliminating any query that would not match the current document. A DTD parser such as [8] builds an internal representation of the DTD tree structure. We can integrate the DTD checks into the existing NFA builder by simply buffering the tokens received for each query. In the original design of the NFA builder each token was immediately processed as soon as it was received. This time we just collect all the tokens for the current XPath expression and place them into a vector. Once all the tokens for one XPath expression has been received, we can begin to check them against the DTD.

The extra costs of the DTD check, comes from the buffering and lookups for each token. In the worst-case scenario each token is looped through twice, once to

check against the DTD, and another to build the NFA states. The size of the vector will be at most the length of the longest query.

Nevertheless we save on the wasted space in the NFA since it doesn't contain any invalid queries. We also get reduced processing time when the NFA is used to create the lazy DFA since it won' t have to visit states in the NFA that lead to no terminal states. Besides we can perform the NFA construction off-line prior to any processing of the XML streams.

4.3 Storing NFA Transitions in DFA

Unwanted events produce DFA states that contain no NFA state pointers, which the query processor ends up deleting. Thus the goal of this enhancement is to avoid the cost of doing the NFA lookups for unwanted events. The technique that we implemented involves storing part of the NFA transitions within the DFA states.

Imagine that we have a current DFA state, *d1*, which contains pointers to the NFA states, *n1,* and *n2*. Figure 4.1 shows the current status of their transition tables.

(a)

Transition	Next State
price	*d2*

(b)

Transition	Next State
stock	*n4*

(c)

Transition	Next State
amount	*n5*

Figure 4.1: Transition tables for (a) DFA state *d1* (b) NFA state *n1* (c) NFA state *n2*

Consider the situation where the query processor is currently at state *d1* and it receives the event, product. The query processor would look up the hash table for, *d1,* and see that there is no product transition. Normally the query processor would proceed to create a new DFA state and check the transition tables for *n1*, and *n2*. It would discover that there is no product transition in either and discard the new DFA state. Instead we store in *d1* another data structure that contains the transitions of both *n1*, and *n2*.

Discrete Transitions
stock
amount

Figure 4.2: The discrete transition table for DFA state *d1*

The discrete transition table stores the transitions from the NFA states associated with the DFA state, except for the ε, //, or * transitions. This table is implemented as a set and remains fixed once it has been created for the DFA state. So after failing to find product in the hash table of *d1*, the query processor checks the discrete transition table shown in Figure 4.2. The product transition does not appear in the discrete transition table, which means product is an unwanted event and the query processor can skip over the rest of the product element.

Maintaining the discrete transition table will improve the processing time at the expense of using more storage. The size of each discrete transition table is actually at most *4n,* where *n* is the number of NFA states in the DFA state. The 4 comes from the fact that an NFA state can have at most 4 different transitions, that is a normal transition, the ε transition, the // transition, and the * transition. Despite the size concerns it improves the performance of the query processor dramatically especially when the XML stream is made up of a large number of elements that are irrelevant to the queries.

4.4 Generalisation of Transitions

Following on from the previous proposal, we propose another solution, which will avoid the costs of inserting the NFA pointers into the DFA state. To illustrate consider the NFA states which contain the transitions ε, //, or * which I term as "*other*" transitions. Any event received will successfully match against these "other" transitions.

Extending the previous example, if *n1* had an "other" transition then the event product should produce a transition to a new state. However by simply checking the discrete transition table in *d1* it would not reveal this and the query processor would have to perform the NFA transition lookups to find the "other" transition in *n1*.

By grouping the "other" transitions together and using a single separate state pointer to represent the next state we can reduce the size of the transition table for the DFA state. Continuing with the same example, the product and name events would both match the "other" transition found in *n1* since neither appear in the discrete transition table of *d1*. But we don't want to update the transition table of *d1* to include separate entries for product and name, since both lead to the same state.

So the next time the query processor receives an event not found in the DFA states transition table or its discrete transition table it checks this "other" state pointer. If it's not null then there is an "other" transition

in one of the NFA states and the pointed to object becomes the current DFA state, otherwise this is an unwanted event and can be skipped.

There is one more requirement for this to work and that is to keep a flag that indicates whether any of the NFA states contains an "other" transition. This flag is needed for the case when we haven't assigned any state object to the "other" state pointer but one of the NFA state contains an "other" transition. So by checking this flag the processor won't mistakenly label the event as unwanted when there is actually a "other" transition. If the flag is true then the query processor can proceed to create a new DFA state and collect the NFA states for the event, otherwise the event is unwanted and can be skipped. As you can see by generalising "other" transitions in the DFA state, the query processor can avoid doing unnecessary processing for events that end up at the same state. The other benefit of this enhancement is the smaller transition tables thus saving space.

4.5 DFA Storage and DFA keys

Retaining the collection of DFA states and calculating new ones presents some difficulties that further complicate the task of the query processor. One of the problems is the comparison of new and previous DFA states. Two DFA states are considered equivalent if their sets of NFA states are equivalent. This comparison needs to take place after every new DFA state is created and all valid NFA states have been added into the DFA state. The comparison will determine whether the query processor is entering an existing state or a previously unvisited state.

For each new DFA state created its key is computed and a mapping table is employed. If the key is not found then this DFA state is an unvisited state and a new entry, consisting of a copy of the key and pointer to the state, is inserted into the map. This new DFA state now becomes the current DFA state, the key is deleted and the previous current state is pushed onto the stack.

If the key is found then this DFA state is an existing state so no new entry into the map is made for the DFA state. The DFA state is deleted along with its key, the state retrieved from the map using the key becomes the current state, and the previous current state is pushed onto the stack.

The only major concern with using the map and the key set is the extra space used to store the keys. The size of each key is dependent on the number of NFA states. There is also the cost of copying the key into the map and deleting keys. However my experimental results show that the number of DFA states is small so there would not be that many keys stored.

4.6 Position Predicates

Support for XPath predicates adds another level of complexity to the query processor. Because of space limitation, we focus on the position predicate in his paper. Consider the following XPath queries:

$Q1 = /a/b

$Q2 = /a/b[2]

In $Q1, both a, and b are general element nodes, and the query would match all occurrences of b elements nested within the root element a. For $Q2 only a is a general element node and the position predicate on b forces the query processor to only consider the second b child element of the root element a. Converting this to the NFA it would like the following.

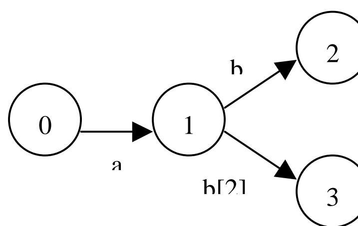


Figure 4.3: NFA with position predicates

As you can see in Figure 4.3, we now have a separate transition for the b element, and the b[2] element. In our implementation, state *1* in Figure 4.3, would now feature the transition table along with a new data structure called the position table as seen in Figure 4.4.

(a)

Transition	Current Position
b	0

(b)

Transition	Next State
b	2
b[2]	3

Figure 4.4: (a) Position table (b) transition table for NFA state 1

Only states that feature a transition with a position predicate will have an entry in the position table. In

Figure 4.3, state *0*, state *2*, and state *3* would have no entries in their position tables. The unboundedness property would only apply to the transition table and not the position table, which only holds one entry per element with a position predicate.

During the processing of start element events, the query processor will first iterate through each of the current NFA states and increment the position tables that match this event. Note that the position values for the common transitions would be the same in each of the current NFA position tables. The query processor will then proceed to perform the DFA transition lookup, and NFA collection as described before. However this time it will be done for both the general element name and the element name with the current position attached.

Using the example shown in Figure 4.3, consider a DFA state that currently contains state 1, i.e. after getting the start of element a. Now if the next event is the start of element b then the query processor will update the position table of state *1* in Figure 5.4, so that it would have a value of 1. The query processor then proceeds to do a DFA transition lookup with the transition b[1]. Failing to find it in the transition table the query processor performs the NFA lookups for both the b transition and the b[1] transition. It discovers the b transition to state *2*, which gets inserted into the new DFA state. The next time the query processor receives another b element that is a child of the currently opened a element, the position value would be 2 which means during the NFA transition lookup it would find b leading to state *2* and b[2] leading to state *3*.

Additionally when the query processor receives end element events, it needs to reset all entries in the position table to 0 for the current state. In this example when element a ends, the position value of transition b for state *1* is reset back to 0.

Nonetheless it is evident that handling position predicates is expensive. There is the increase in space cost due to the creation of extra NFA states. Then there is the increase in processing time to execute the incrementing and resetting of the position values in the position tables. This computation can be avoided by first checking to see if there is actually a position entry in the position tables. However this itself incurs the cost of checking each of the NFA states.

4.7 SIX

A query processor can take advantage of the SIX if it is available for an XML document. The SIX is typically created at the same source as the XML document so the publishers of the XML need to deliver both the XML stream and SIX stream at the same time. In the case where there was no SIX available for the XML document the query processor resorts to simply ignoring SAX events that fall within the context of the element that was to be skipped. This is straightforward since all that needs to be done is to remember the skipped element name. Then for all element events that are received before the end element event of the skipped element, we bypass all the processing for them. Once the end element event corresponding to the skipped element is found we clear the skipped element name and continue processing the events as normal.

4.8 Pull Parser

The Pull Parser [22] is an ideal XML parser for our query processor because it provides, as part of its API, a function that performs internally, the skipping over elements without the need to manipulate the byte offsets on the input buffer stream. A pull parser is different from a SAX parser in that the handler initiates the calls to extract items from the XML document. That is the pull parser acts like a tokeniser such that the onus is on the handler to make next() calls to retrieve the next XML event. The SAX parser is more of a "push" parser in that the handlers do not make explicitly requests for each XML event. They merely wait for the parser to deliver them via calls to their event handling functions, when they are produced.

The other key factor that makes the pull parser ideal is that it produces XML events incrementally similar to a SAX parser. That means the pull parser doesn't need to read in the entire document before producing events. Although there may be some buffering of the XML to process parts of it, it doesn't build an internal representation of the XML like a DOM parser. This makes the pull parser just as efficient as a SAX parser.

We have implemented the query processor, which utilises the pull parser. The only difference is that now the registered handler makes calls to the parser to retrieve the next event. It checks to see which type of event it is and delegates it to the processor just like the handler for the SAX parser version. The query processor will process the event just as before and determines whether we can skip this element.

Instead of running a SIX Manager to maintain the stream of SIX entries for skipping, all the query processor needs to do is tell the handler to call skip. The pull parser will skip over the rest of this element which means the next time the handler calls next() it will return the event for the element following the end of the skipped element.

Internally the skip function is actually accomplished with repeated calls to next(). Although this is not as efficient as directly manipulating the input stream we still save on all the calls to event handler functions if we used a SAX parser.

5 Experiments

All experiments were conducted on an Intel Pentium III 666Mhz processor with 384MB of memory running Red Hat Linux 7.3. The query processor was run

against real NASA XML datasets [18] created by concatenating 2,436 XML documents into one single file. The file was 25MB in size and contained 476,646 elements. The size of its SIX file, containing an entry for all elements, was approximately 3.8MB. For comparison purposes the query processor was also run against synthetic NASA XML datasets created using the dataset_048.dtd [19], which is also used by the real XML data. This DTD contains 108 elements with a single non-simple cycle.

The synthetic XML data was generated using IBM's XML generator tool [7]. The synthetic XML document used is 28.8MB in size with 345,272 elements. It is generated with a maximum depth of 10 and maximum repeat of 15. The size of its SIX file is 2.8MB. The sets of input queries used contain 50% randomness.

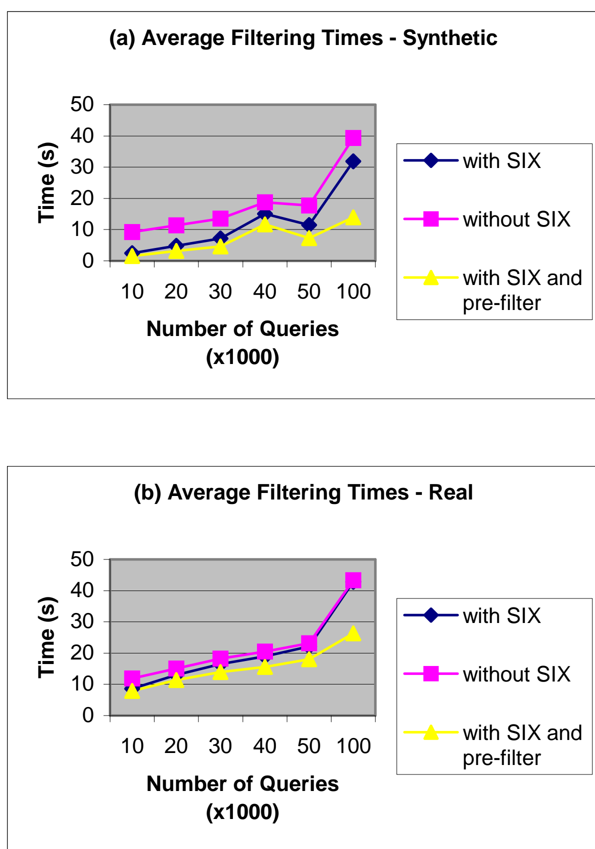


Figure 5.1: Average Filtering Times for (a) Synthetic (b) Real data

The graphs in Figure 5.1, shows the average filtering times for the synthetic and real XML data as the number of queries is increased. As expected the filtering times for both synthetic and real XML grows linearly as the number of queries is increased. It is also evident that pre-filtering of the queries and use of the SIX generally performs better than using SIX alone, which in turn performs better than without SIX. For the case of 100,000 queries only the pre-filtering method actually completed without paging, which is due to the fact that a significant number of irrelevant queries have been filtered out prior to processing.

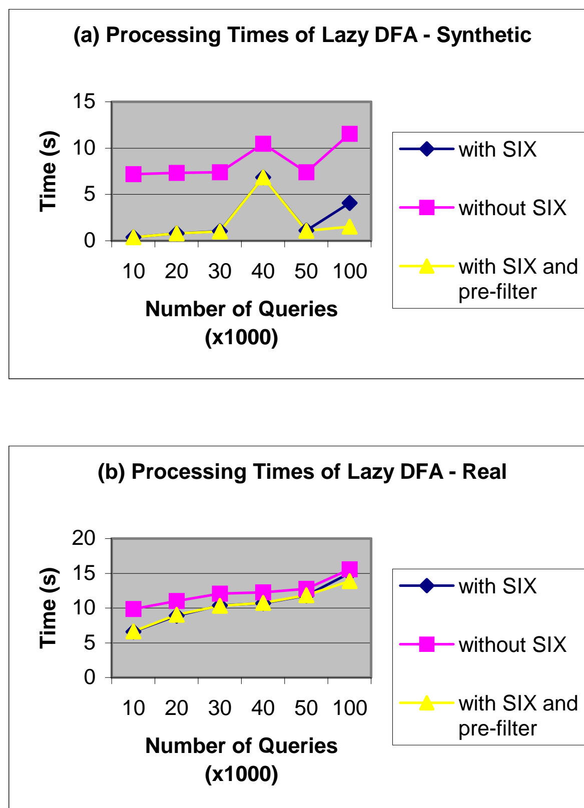
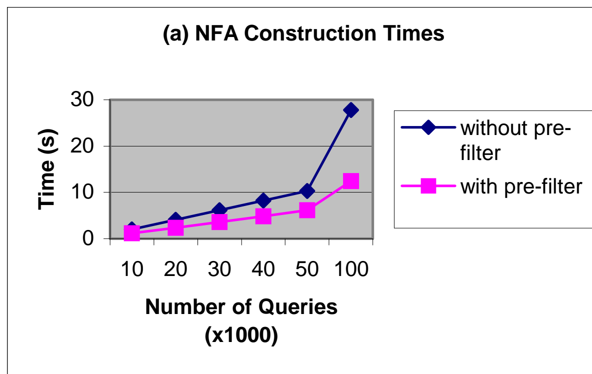


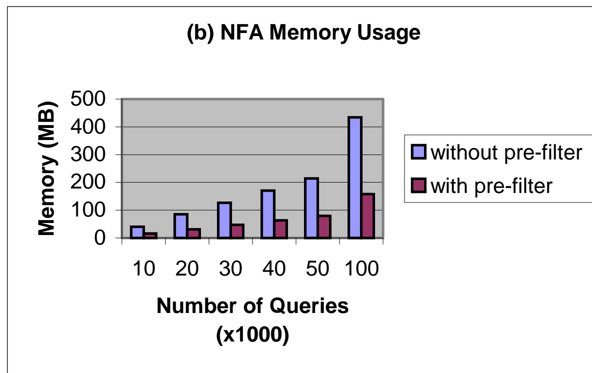
Figure 5.2: Processing Times of Lazy DFA for (a) Synthetic (b) Real data

Figure 5.2 (b) shows that for real XML data the SIX performs marginally better than without SIX. This is a consequence of the real XML data contains many small elements, which causes the processor to do more SIX operations such as reading entries and maintaining the SIX stack. Compare this to the synthetic version, in Figure 5.2 (a), where without SIX is significantly slower than the others. This is because the elements in the synthetic XML tend to be large, which allows the SIX to skip over more. Furthermore we can see that the lazy DFA is capable of processing up to 100,000 queries without a dramatic increase in the processing time. Note also that the pre-filter method doesn't improve the lazy DFA's performance because the pre-filter only helps in the reduction of the NFA construction cost, which occurs before any lazy DFA processing.

The motivation for the pre-filter method is to minimise the time it takes to create all the NFA states and ultimately save on the memory usage. The pre-filter

method is able to eliminate queries that don't start with the right root node. A comparison of the NFA construction is shown in the graphs of Figure 5.3.





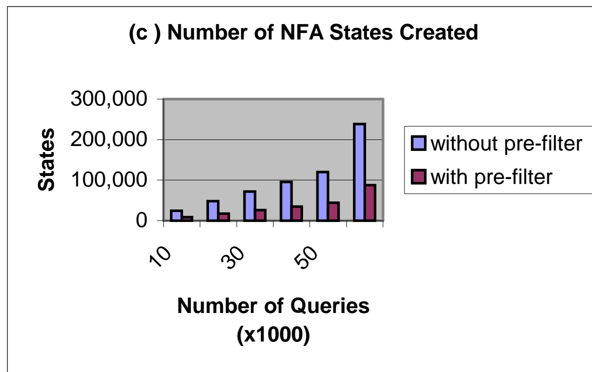
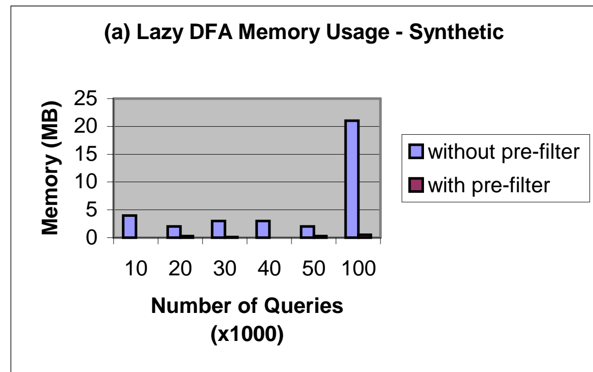


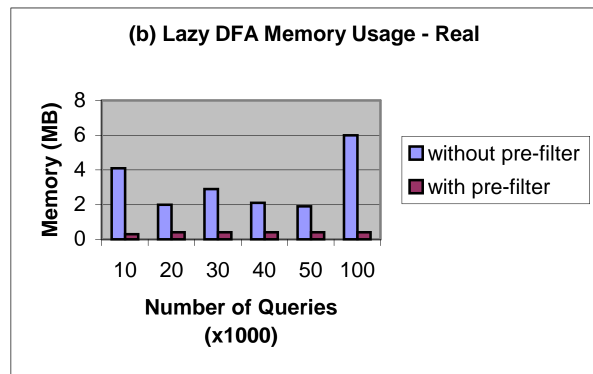
Figure 5.3: Results for NFA (a) construction times (b) memory usage (c) states

Figures 5.3 (a) and (b) shows the NFA construction times and memory usage, growing linearly with the number of queries. By pre-filtering the queries we can reduce the time spent creating the NFA. When pre-filtering was used, the NFA construction for 100,000 queries was completed without paging the disk.

As you can see in Figure 5.3 (c), the number of NFA states is proportional to the number of queries, which is consistent with the results of the memory usage by the NFA. Once again we can see the effectiveness of the pre-filtering method to help reduce the size of the NFA. This experiment has shown that the NFA is the main contributor of the processors memory usage.

The lazy DFA memory usage on the other hand is considerably less. This is because only the required states are created in the lazy DFA. In the case of linear XPath queries the lazy DFA's memory remains low because the number of states in the DFA was said to be dependent only on the occurrences of *, and // as given by the theorem. This can be seen in the graphs of Figure 5.4.





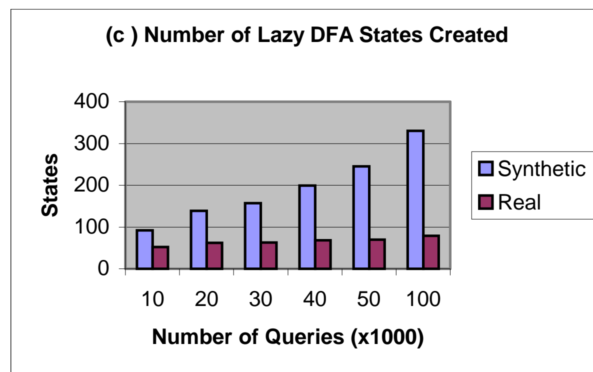


Figure 5.4: Lazy DFA memory usage for (a) Synthetic (b) Real, and (c) states created

Figure 5.4 (a) and (b) shows the memory used by the lazy DFA is less than 5MB, except for 100,000 queries which was affected by the memory overflow from the NFA. The memory usage of the lazy DFA tends to be independent of the number of queries and is usually determined by the coverage of the queries over the XML document. Note for synthetic XML, shown in (a), the pre-filtering is able to further reduce the memory usage in all cases to below 1MB. This is because pre-filtering reduces number of NFA states, which causes the lazy DFA to store less of the NFA transition tables.

Figure 5.4 (c) shows that the number of states in the lazy DFA remains consistently low for the real XML data compared to the synthetic XML data, which tends to grow linearly with the number of queries. This is due to the generator used to create the synthetic XML data, which will attempt to produce all possible nesting of elements as given in the DTD. In practice there will only be limited nesting of elements, such as in the real XML data, resulting in only a small number of lazy DFA states.

6 Conclusions

This paper discussed the issues that affect the efficiency of the lazy DFA approach for processing XML stream data. These issues included the NFA construction costs, the handling of predicate filters, the lazy DFA construction cost, and the SIX cost. Optimizations have been proposed to address each of these issues. Finally experiments and analysis showed that our improved lazy DFA approach performs significantly better (in terms of both the time/space efficiency and scalability) than the original DFA approach.

7 References

[1] M. Altinel, and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In Proceedings of VLDB, pages 53-64, Cairo, Egypt, September 2000.

[2] Apache. Xerces C++ Parser. http://xml.apache.org, 2002.

[3] S. Babu, and J. Widom. Continuous Queries Over Data Streams. SIGMOD Record, 30(3):109-120, September 2001.

[4] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In Proceedings of the International Conference on Data Engineering, 2002.

[5] J. Chen, D. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pages 379-390, Dallas, TX, May 2000.

[6] Y. Chen, S. B. Davidson, Y. Zheng. Validating Constraints in XML. Technical report, University of Pennsylvania, 2002. Technical Report MS-CIS-02-03.

[7] A. Diaz, and D. Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, 2001.

[8] DTDParser. http://www.wutka.com/dtdparser.html

[9] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscribe Systems. In Proceedings of ACM SIGMOD, pages 115-126, Santa Barbara, California, May 2001.

[10] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, D. Shasha. Webfilter: A High-throughput XML-based Publish and Subscribe System. In Proceedings of The {VLDB} Journal, pages 723-724, 2001.

[11] L. Fegaras, D. Levine, S. Bose, V. Chaluvadi. Query Processing of Streamed XML Data. Submitted, November 2001.

[12] T. Green, G. Miklau, M. Onizuka, D. Suciu. Processing XML Streams with Deterministic Automata. Submitted to The 9th International Conference on Database Theory, Siena, Italy, 8-10 January 2003.

[13] A. Gupta, A. Y. Halevy, D. Suciu. View Selection for Stream Processing. Submitted to Fifth International Workshop on the Web and Databases (WebDB 2002), Madison, Wisconsin, June 2002.

[14] Ashish Gupta, Dan Suciu. Stream Processing of XPath Queries with Predicates. In Proceeding of ACM SIGMOD Conference on Management of Data, 2003

[15] J. Hopcroft, and J. Ullman. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, pages 13-46, 1979.

[16] Z. Ives, A. Levy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical report, University of Washington, 2000. Technical Report UW-CSE-200-05-02.

[17] T. Milo, and D. Suciu. Index Structures for Path Expressions. In ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, pages 277-295, 1999.

[18] NASA astronomical data center XML file. http://xml.gsfc.nasa.gov/guides

[19] NASA DTD dataset_048.dtd. http://xml.gsfc.nasa.gov/DTD/dataset_048.dtd

[20] NITF DTD nitf-2-5.dtd. http://www.nitf.org/nitf-documentation/nitf-2-5.dtd

[21] B. Nguyen, S. Abiteboul, and G. Cobena. Monitoring XML Data on the Web. In Proceedings of the 2001 ACM SIGMOD International Conference On Management of Data, pages 437-448, May 2001.

[22] A. Slominski. XML Pull Parser version 2.1.8. http://www.extreme.indiana.edu/xgws/xsoap/xpp/, 2002

[23] A. Snoeren, K. Conley, and D. Gifford. Mesh-based Content Routing Using XML. In Proceedings of the 18th Symposium on Operating Systems Principles, 2001.

[24] E. Viglas, and J. Naughton. Rate-based Query Optimization for Streaming Information Sources. In Proceedings of SIGMOD, 2002.

[25] XMLTK. The XML toolkit. http://www.cs.washington.edu/homes/suciu/XMLTK/, University of Washington, 2002.