Indexing XML to Support Path Expressions

Dmitry Barashev and Boris Novikov

University of St. Petersburg, Russia
{db2,borisnov}@acm.org

Abstract. The extensible markup language (XML) is rapidly becoming a dominating technology in the area of data intensive applications. Although several implementations are already offered in commercial products, especially DBMSs, there are still open research issues related to efficiency of XML storage and retrieval.
This paper introduces and analyses new index structures suitable for support of regular expressions over character data combined with path expressions in XML queries. The performance of these structures is analyzed and compared with performance of alternative approaches.
In addition to usual criteria of I/O and CPU performance, a possibility to implemenent new index structure within existing DBMS engines is considered.

1 Introduction

During few years since the XML language standard was first introduced, it became one of most frequent buzzwords. Everything in information technology is now either supporting or moving to XML.

A huge amount of research and practical developments were done related to XML. In the database research community, the efforts were concentrated on definition and refinement of semistructured data models and languages, query evaluation techniques, wrapper generation, as well as research of storage structures for both native XML databases and representation of XML over relational database systems. XML was also supported by many of commercial DBMS vendors who designed comprehensive tools and components enabling developers to create XML-based applications.

However, the standards are still developing rapidly, and there are many research issues related to different aspects of XML from conceptual models to data structures. The technology did not reach it's maturity yet.

The specific feature of XML and XML query languages, in contrast with traditional relational databases, is support for paths and path expressions. A query may specify, in addition to conditions on attribute values, certain conditions on paths to be selected. These conditions may include matching path against some regular expression. Support of paths in indexing techniques proposed so far for semistructured data and XML is not perfect and even worse results are achieved in evaluation of path expressions with pattern matchings.

The goal of this paper is to define and evaluate data structures and search algorithms which provide support for path expressions with pattern matchings in queries to XML documents.

For large stored XML datasets, the direct evaluation of regular path expressions may be both I/O and CPU intensive. For this reason, the search algorithm should try to access as few index pages as possible (which is usual objective of any secondary storage index structure) and try to reduce the amount of data to be tested against reqular expression (that is, processed by finite automaton).

Some of XML indexing techniques proposed elsewhere, e.g. [3] may be adapted for path expression selections, however, their behavior and efficiency will vary very significantly, depending on the form of the path expression. One more goal is to achieve more uniform performance, regardless of the form of reqular expression.

The paper is organized as follows. In the next section we analyze previous work done on XML indexing, in both research and industrial contexts. A couple of indexing structures are introduced in section 3. In section 4 these methods are analyzed and compared with methods defined elsewhere. The major results are summarized in the conclusion.

2 Related Work

A number of research efforts have been made to introduce and investigate storage, index structures, and access methods suitable for efficient querying and retrieval of semistructured data collections and XML databases and documents.

Many researchers investigated different ways to store semistructured data and XML in relational databases [4, 5, 13]. These approaches do not require consideration of any special index structures, because they rely on the power of relational query processing engine. Actually, variations of this approach are also adopted in commercial DBMSs.

Several index structures were defined in *Lore* [6, 8]. They index values of atomic objects supporting type coercion (*Vindex*) and keyword search (*Tindex*) and labeled paths (*Pindex*).

An efficient index structure *Index Fabric* was proposed in [3]. This structure is well tuned for path selection and is capable of indexing huge amounts of data. However, this structure does not specifically support regular expressions on paths.

A problem of regular path expressions was addressed in [12]. This research introduces several indexes and algorithms tuned for efficient processing of three operations: find all elements/attributes having the given name, find parent and children of the given element, for the given two elements find out if one of them is the parent of another. These operations are very useful when regular expression over document structure elements is evaluated, but they do not deal with regular expressions over document data or over document structure elements alphabet.

Regular expressions searching on tries was investigated in [2]. This work provides a good algorithm of regular expressions searching on Patricia tries but it works in main memory and does not consider disk access costs.

Major industrial DBMS vendors offer extensive support for XML.

Oracle [11] supports two methods to store XML: *generated XML* and *native XML*. In the former case, the XML is not actually stored in the database, instead, it is generated on the fly from data stored in conventional tables. Hence, all power of conventional DBMS indexing (and query optimization) is applicable in this case. However, this indexing actually has nothing to do with XML. The latter option, native XML, means that documents are just stored as LOBs, and user-defined indices may be implemented using, for example, function-based index techniques. This is just the place where our proposals might be valuable.

The *IBM DB2* system supports two options for storing XML documents: *XML collection* and *XML column* [7] which are similar to Oracle's generated and native XML storage methods correspondingly.

The *Microsoft SQL Server 2000* [9] supports XML views and provides functionality for storage and retrieval of XML documents, but does not provide any specific index structures for XML.

3 Index Structures to Support Path Expressions

In this section we introduce data structures able to support queries of XML documents containing regular expressions on paths. We shall trail an algorithm of regular path expression evaluation on two different index structures starting from the most abstract and general considerations and further going deeper into details of algorithm.

Indexing structures considered in our work are *B-tree* and it Patricia trie [10]. Both of them are well known in database world and B-trees are widely used in industrial database management systems.

We use a widely known representation of XML document as a set P of dot-separated paths where each item is a label of some XML element. We assume that labels are represented as words in certain alphabet of fixed size (e.g. ASCII or Unicode) denoted hereafter Σ. Finiteness of Σ is an important precondition for our indexing structures and automata.

Given a regular path expression and XML document represented as a set of all paths, our goal is to find all paths satisfying the regular expression. For large documents, any full scan algorithm would be unacceptable both in terms of CPU and I/O, so it is necessary to reduce the number of paths to be tested against regular expression and reduce the cost of this testing.

3.1 Representation of Paths Set

As first step in refinement of the data structure, we sort set P lexicographically. This is actually done in all methods dealing with XML paths, regardless of their intention to support path expressions. The ordered set P is represented in our index structures as follows.

Paths in B-tree are stored as tree keys. For purposes of both disk space and CPU time optimization we use left truncation for paths. Although the truncation is not considered as efficient in general case, it is expected to be very helpful for paths.

Patricia trie based index consists of two tries. The first one called *Top-Down Patricia Trie* is constructed from P as suggested in [3] without additional layers. However, information stored in Top-Down trie is not sufficient for our purposes because of suffixes which are not covered by the trie.

Let S be the set of all suffixes. To make this information available for searching we build a *Bottom-Up Patricia trie* from elements of S sorted in reverse lexicographical order (i.e. starting from the last to first character). An important difference between Top-Down and Bottom-Up tries is that every leaf vertex of Top-Down trie points to one data value, while leaf vertex of Bottom-Up trie may point to several data values having the same suffix (see Fig. 1)

We use a special numeration of Patricia trie vertices. The idea and intentions of this numeration are similar to numeration suggested in [12]. However, we exploit finiteness of Σ and our numeration does not need renumbering at all. If we have chosen some order in alphabet Σ such that every character α has got a unique number $num(\alpha)$ in range $0...|\Sigma|-1$ then numeration rule is the following: vertex of height h with incoming edge labeled α gets a number $\frac{num(\alpha)}{|\Sigma|^h}$.

We'll use a notion of (abstract) *index structure element*. For B-tree those elements are disk blocks and for Patricia trie the elements are vertices of the trie. The key difference between B-tree and Patricia trie with respect to index structure elements is that a maximal amount of children of B-tree index element is bounded by a value which does not depend on the Σ alphabet size while for Patricia trie it equals to $|\Sigma|$ and if Σ is infinite then index structure element may have an infinite amount of children.

We define also an *index structure path* as a sequence $\{e_i\}_{i=1}^n$ of index structure elements starting in the top element of index structure and finishing in the leaf element such that e_i is a child of e_{i-1} for $i = 2 \dots n$. We say that index structure path *points to* (a set of) index keys located in the end element of the path.

3.2 First Refinement: Determine Search Directions

We assume hereafter that we search for matches of pattern Q. As a first step to find a match we find a set of *"search directions"*. On the abstract level search direction is an index structure element such that a set of keys pointed by index structure paths passing through this element may contain a matching key. We give a guarantee that no other keys except those produced by the set of search directions contain matching paths.

As applied to B-tree search direction is just a leaf block. To locate search directions we use second-level index that contains labels as keys and the entries of this index refer to leaves of B-tree, which store a path containing the label (in any position). We extract fixed label names from Q, if any and then for each label use second-level index to build a list of blocks with potentially relevant paths. Then we compute the intersection of these block lists (actually, sometimes union is also needed) and finally get a set of search directions.

For Patricia trie we also use second level index. This index maps symbol $\alpha \in \Sigma$ into set R_α of vertices having an incoming edge labeled with α. We find these sets for symbols from Q. Each vertex from the found sets is a search direction. We certainly do not need all of them and we use automata running on tries to truncate search directions set.

We have to build two sets of search directions: one for Top-Down trie and one for Bottom-Up trie. An example of Top-Down and Bottom-Up search directions set for symbols "c", "F" and "G" is shown on Fig. 1 where search directions are marked by the bold line.

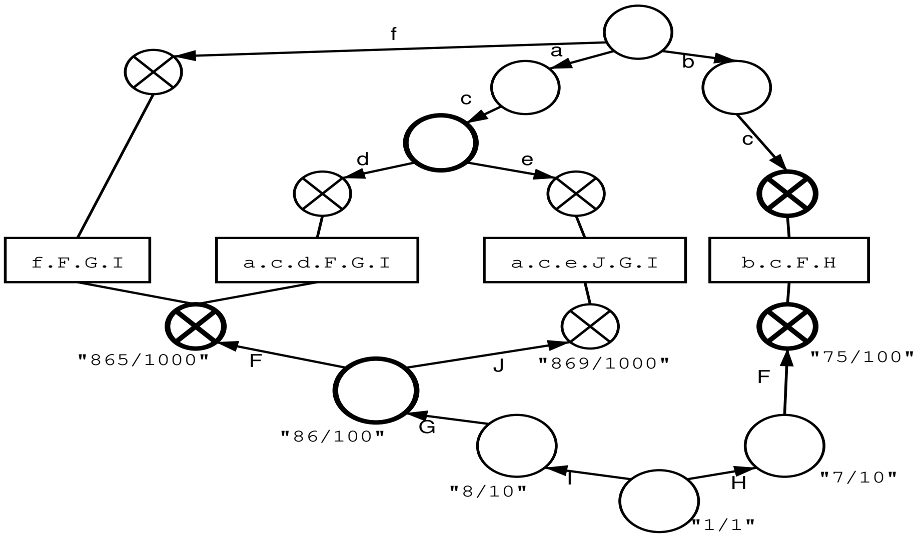

Fig. 1. Top-Down and Bottom-Up Patricia tries

3.3 Exploring a Search Direction

B-tree We build a NFA from our query Q and for each block from search directions intersection sequentially process the paths in the block, creating a stack of automaton states. When we move to the next path we rely on this stack and on prefix truncation to choose a proper initial state of automaton.

Patricia Trie We build a finite automaton on Patricia trie to find matches of Q. This automaton is constructed from two automata one of which works on Top-Down trie and second works on Bottom-Up trie.

Bottom-Up trie automaton is constructed from pattern Q by Thompson algorithm [1] with an additional modification allowing those suffixes that match the end part of Q to be stored for later usage. For this purpose we create an additional state S^{write} and transitions $S \longrightarrow S^{write}$ and $S^{write} \longrightarrow S$ for some states produced by Thompson algorithm. Automaton follows this transition when it reaches a leaf of Bottom-Up trie.

Input of Bottom-Up automaton is a sequence of search directions sets $\{R_\alpha\}$ built for every symbol $\alpha \in \Sigma$ found in Q in reverse order.

Bottom-Up automaton keeps a set V of Bottom-Up trie vertices. Set V contains search directions where automaton still has a chance to find a matching suffix. In initial state set $V = Root_{BU}$ where $Root_{BU}$ is the root of Bottom-Up trie.

When automaton is in the state S it calculates an intersection $V_L := V \cap Leaves_{BU}$ where $Leaves_{BU}$ is a set of Bottom-Up trie leaves. If $V_L \neq \emptyset$ then automaton follows a transition $S \longrightarrow S^{write}$ and writes to output a pair (Q_{suf}, V_L) where Q_{suf} is a suffix of pattern matched so far by strings stored in Bottom-Up trie.

If the only transition from the current state is labeled with α then automaton takes search directions set R_α and assigns new value to $V := Tree(V) \cap R_\alpha$ where $Tree(V)$ is a union of subtries rooted by $v \in V$.

If there are two outgoing ϵ-transitions (that means there is either choice $\alpha|\beta$ or Kleene star $\alpha*$) then automaton pushes current state into stack and sequentially follows the transitions.

Lets illustrate Bottom-Up automaton work on sample trie shown on Fig. 1 and pattern "c*FG". Bottom-Up automaton for this expression is shown on Fig. 3.3. Input of automaton consists of sets R_G, R_F, R_c.

Automaton starts in state 1. $V = \{\frac{1}{1}\}$ and the next input is $R_G = \{\frac{86}{100}\}$. Automaton calculates $Tree(V) \cap R_G$ and gets $V = \{\frac{86}{100}\}$. The only transition here is $S_1 \longrightarrow S_2$.

In state 2 automaton calculates $V_L := V \cap Leaves_{BU}$. $V_L = \emptyset$ so automaton takes the next input $R_F = \{\frac{865}{1000}, \frac{75}{100}\}$. An intersection $Tree(V) \cap R_F$ gives $V = \{\frac{865}{1000}\}$ and automaton follows the transition $S_2 \longrightarrow S_3$.

In state 3 automaton gets non-empty $V_L = \{\frac{865}{1000}\}$. Currently Q_{suf} ="FG" and automaton writes a pair $(``FG'', \frac{865}{1000})$ to output.

Similarly automaton writes to output a pair $(``c * FG'', \frac{856}{1000})$ from state 6.

When Bottom-Up automaton finishes we have the following output:

$$(``FG'', \frac{856}{1000}) \qquad (``c * FG'', \frac{856}{1000})$$

Note that the second element of output matches the whole pattern. Thus, if Top-Down automaton during its work enters path having suffix passing through $\frac{856}{1000}$ vertex then it will know that path already matches and it is not necessary to examine it. In our example such kind of paths are "fFGI" and "acdFGI". Nevertheless, we need to run Top-Down automaton to find matchings entirely located in Top-Down trie.

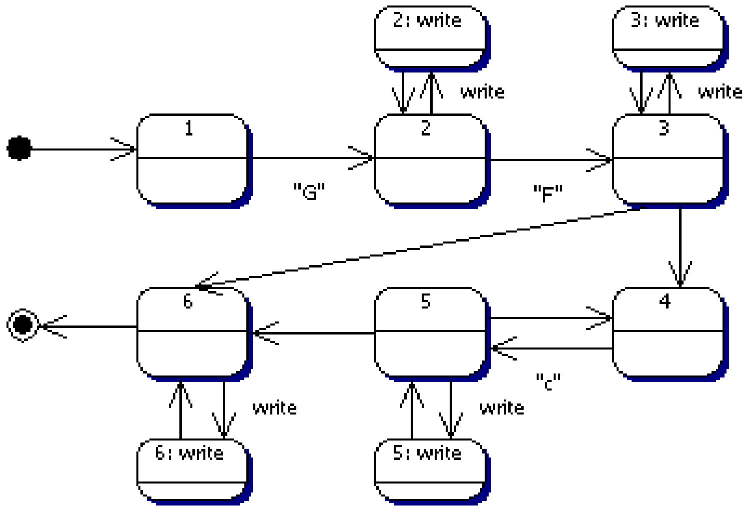

Fig. 2. Bottom-Up automaton

Top-Down automaton is very similar to Bottom-Up one. We again use Thompson algorithm to construct it and add some extra states where Top-Down automaton looks into results provided by Bottom-Up automaton to decide if it possible to stop work here. Positions of extra states are determined by suffix element of pairs produced by Bottom-Up automaton.

In stop states Top-Down automaton makes a decision whether it needs to explore Top-Down trie further. Automaton stops if it knows that subtrie rooted by vertex $v \in V$ together with some path(s) in Bottom-Up trie match the rest of pattern. The following conditions must be held to stop exploring the direction v:

- Unprocessed pattern suffix s is contained in key set of Bottom-Up automaton output. Let $Bottom_s$ be a set of bottom search directions corresponding to s.
- The direction v (partially) matches the beginning of s. This is the case, e.g. if either v is leaf vertex of Top-Down trie or beginning of s is ".*"
- $Targets(v) \cap (\cup_{u \in Bottom_s} Targets(u)) \neq \emptyset$ where $Targets(v)$ is a set of index keys pointed by paths passing through v.

4 Evaluation and Comparisons

In this section we provide a performance analysis of our indexing structures and search algotihms both in terms of time and disk space. Our main time measure is disk I/O operations count. However, we also consider CPU operations cost.

We start with evaluation of the cost of finding search directions.

Both indices require a full scan of regular path expression to find fixed symbols and possibly extract those located inside label bounds. It costs $O(|Q|)$ where $|Q|$ is a length of regular path expression. For each fixed symbol α found we have to extract IDs of blocks containing paths with α from blocks index. We claim that in order to obtain an acceptable result we need to access only a few pages from blocks index.

Actually for some α there may be $p = [\frac{B}{b}] + 1$ pages in blocks index where B is a total number of blocks in index structure and b is a total number of block IDs fitting in one page of blocks index and thus in the worst case I/O cost of accessing blocks index is $O(B)$. However b tends to be quite big and if p is more than 1 for some symbol α then the algorithm shoud decide whether it needs to consider α, or it is cheaper either to proceed to the next symbol (remember that finally we will build an intersection) or to make a full scan of paths. Thus, we claim that I/O cost of search directions calculation for Patricia trie based index is $O(|Q|)$ and for B-tree it is $O(|Q|_{labels})$ where $|Q|_{labels}$ is the count of labels in Q.

Disk space used for blocks index is proportional to the number of labels in the stored data. For each label we have to store IDs of blocks holding keys containing that label. In the worst case for each label index allocates $\frac{|P|}{cap}$ disk blocks where $|P|$ is the total number of paths indexed and *cap* is a capacity of B-tree leaf pages and disk space estimation for blocks index equals to $O(|L|) * O(|P|)$.

Let $|Q|_{useful} = min(|Q|_{fixed}, height(Bottom))$. Then Bottom-Up automaton construction takes time and memory space $O(|Q|)$ and it runs in $O(|Q|) \times O(|Q|_{useful})$ steps. The latter estimation is based on algorithm described in [1] and observation that depth of vertices in set V increases with every next fixed symbol from Q. Thus when either V contains only leaves or input sequence is finished automaton will be stopped.

Below we perform a comparison of our two algorithms performance and their asymptotic behavior.

Assuming that symbols index always requires not more than 1 IO operation for each symbol in Q we conclude that Bottom-Up automaton works in

$$|Q| \times IO + |Q| \times |Q|_{useful} \times (T_1(V) + T_2(V, R_{next})) \times CPU \tag{1}$$

where T_1 is a cost of leaves test and T_2 is a cost of intersection. Both T_1 and T_2 require time linear with respect to sum of their arguments. For Top-Down automaton estimation is the same to Q_{useful} calculation.

For B-tree based index assuming that secondary index requires not more that 1 IO operation for each label found in Q estimation is

$$|Q|_{labels} \times IO + |Q| \times |p|_{avg} \times CPU \tag{2}$$

where $|p|_{avg}$ is an average length of indexed path.

4.1 Asymptotic behavior

As Bottom-Up trie considers only suffixes of the indexed paths it is not likely to grow with the growth of structural part of document. However, it will grow linearly with the growth of document character data. The most sensible component of 1 with respect to trie size is T_1 which tests if V contains leaf vertices. Thus, Bottom-Up automaton work time grows linearly together with the growth of character data amount in the document.

Top-Down automaton is more likely to grow as structural part of the document grows. When Top-Down trie is not very populated each new path adds one new vertex. However, when trie is well populated the growth slows down because paths usually trend towards common prefixes. Thus we expect a logarithmic growth of Top-Down trie with respect to growth of structural data amount and the same decrease of performance.

For B-tree index structure IO operations count seems to grow linearly with the total number of paths indexed. CPU time consumed by automaton growth linearly with character data amount and logarithmically with structural data amount due to prefix truncation.

4.2 Comparison with other indexing structures

Index Fabric suggested in [3] has no any special support for regular expressions searches. Neverthless, regular expressions can be searched with Index Fabric using some algorithm of searching over Patricia trie (e.g. [2]). In this case performance strongly depends on the regular expression class.

For expressions like ".*a" Index Fabric requires a full scan of the indexed values while for expressions like "ˆabc" where "a", "b" and "c" are element names it accesses only three internal vertices. Actually Index Fabric will show a comparable and probably better performance only for regular expressions either having an anchor "ˆ" or having a match close to the root of the trie. In all other cases especially if communication cost between internal vertices of trie is rather high, our indexing structures and algorithms seem to have a better performance due to labels indexes.

5 Conclusions

The data structures presented in this paper have different characteristics and complementary advantages and disadvantages and therefore are suitable in different circumstances.

The B-tree index can be used for documents stored on the single database node where all blocks can be accessed locally and it is possible to run automaton in main memory for quite big amount of data. The important practical advantage is that this structure can be relatively easily implemented on top of standard database structures and hence inherits concurrency control, recovery etc. However, this structure appears less efficient than alternatives.

Patricia trie index is implemented outside of the database anyway, and hence is more useful in a distributed and heterogeneous environments, including external data sources, where it is impossible to quickly load a huge amounts of data into the main memory and cost of navigation between vertices is high. The major disadvantage is poor worst-case performance.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1985.
2. Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM*, 43(6):915–936, November 1996.
3. F. Cooper, Neal Sample, Michael J. Franklin, Gisli Hjaltason, and Moshe Shadmon. Fast index for semistructured data. In *Proc. VLDB 2001*, pages 341–350, 2001.
4. Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1999.
5. Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. In *Proceedings of the VLDB'99*, 1999.
6. Roy Goldman and Jennifer Widom. Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of Workshop on Query Processing for Semistructured Data*, January 1997.
7. IBM. *XML Extender Brochure*, 2000.
8. Jason McHugh, Jennifer Widom, Serge Abiteboul, Quingshan Luo, and Anand Rajaraman. Indexing semistructured data. Technical report, Stanford University, 1998.
9. Microsoft Corp. *Microsoft SQL Server 2000 Books online*, 2000.
10. Donald R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.
11. Oracle Corp. *Application Developer's Guide -XML*, 2001.
12. Li Quanzhong and Moon Bongki. Indexing and quering xml data for regular path expressions. In *Proc. VLDB 2001*, pages 361–370, 2001.
13. Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. de Witt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the VLDB'99*, pages 302–314. Morgan Kaufmann, September 1999.