Generating Synthetic Complex-structured XML Data

Ashraf Aboulnaga Jeffrey F. Naughton Chun Zhang

Computer Sciences Department
University of Wisconsin - Madison
{ashraf,naughton,czhang}@cs.wisc.edu

Abstract

Synthetically generated data has always been important for evaluating and understanding new ideas in database research. In this paper, we describe a data generator for generating synthetic complex-structured XML data that allows for a high level of control over the characteristics of the generated data. This data generator is certainly not the ultimate solution to the problem of generating synthetic XML data, but we have found it very useful in our research on XML data management, and we believe that it can also be useful to other researchers. Furthermore, we hope that this paper starts a discussion in the XML community about characterizing and generating XML data, and that it may serve as a first step towards developing a commonly accepted XML data generator for our community.

1 Introduction

Synthetically generated data is very useful in evaluating and understanding new ideas in database research. For example, research on relational databases often uses synthetic data from the Wisconsin benchmark [DeW93], TPC-C [TPCC], or TPC-H [TPCH], and research on object oriented databases often uses synthetic data from the OO7 benchmark [CDN93].

Synthetic data generators allow us to generate large volumes of data with well-understood characteristics. We can easily vary the characteristics of the generated data by varying the input parameters of the data generator. This allows us to systematically cover much more of the space of possible data sets than relying solely on real data over which we have little or no control. As such, using synthetic data for evaluating research ideas and testing the performance of database systems can provide us with deeper insights and stronger conclusions than relying solely on real data. Of course, while experimenting with synthetic data is an ideal way to explore the behavior of different solutions on data with different characteristics, an additional validation step may be necessary to ensure that the conclusions drawn from synthetic data extend to real world applications.

In this paper, we describe a data generator that generates synthetic complex-structured XML documents. The data generator can generate XML documents of arbitrary complexity, and it allows the user to specify a wide range of characteristics for the generated data by varying a number of simple and intuitive input parameters. The data generator is certainly not a finished product, but it is our hope that making it publicly available and discussing it and related issues will serve as a first step toward the development of a common data generator for the XML community.

An alternative to using synthetic data is to use only real data for XML research. Using only real data can be very limiting for three reasons. First, there is not much publicly available XML data at this time. Second, all the real XML data that we have encountered has relatively simple structure. Using more complex XML data can provide better insights, even if this data is synthetic. Third, like all real data, we have very little control over the characteristics of real XML data.

Our data generator can be used to generate synthetic XML data that resembles real data, but this is not the goal of the data generator. The goal of the data generator is rather to generate XML data sets with widely varying characteristics by varying the input parameters, thereby covering many different parts of the space of possible XML data sets. This allows the data generator to be used in a wide range of applications to gain insights into the performance of proposed techniques on different kinds of XML data. The simple and intuitive nature of the data generation parameters means that the characteristics of the generated XML data will be easy to understand, even though this data may not necessarily resemble any available real data.

The data generator generates tree-structured XML documents of arbitrary complexity. It generates XML elements and values within these elements, but it does not currently handle the generation of attributes. The data generator starts by generating a tree, which we call the *path tree*, that represents the structure of the XML data. The data generator assigns tag names to the nodes of this tree, and specifies the frequency distribution of the XML elements represented by these nodes. It uses the information in this tree to generate one or more XML documents, and it also generates values for the elements in these documents. The input parameters of the data generator provide a high degree of control over all the steps of the data generation process.

We have found our data generator very helpful in our research on XML data management [AAN01, ZND+01] and in evaluating and testing the Niagara system [Nia]. We believe that the data generator can be as useful to other researchers as it is to us. Furthermore, we hope that this paper stimulates a discussion among researchers in the XML community about characterizing and generating XML data. It is also our hope that our data generator can be a seed that, through extensions and modifications by other researchers, can eventually grow into a useful common data generator for our community. The data generator is publicly available from http://www.cs.wisc.edu/niagara.

The rest of this paper is organized as follows. Section 2 presents an overview of related work. Section 3 describes generating the tree that represents the structure of the XML data. Section 4 describes assigning tag names to the nodes of this tree. Section 5 describes our approach to specifying a frequency distribution for the XML elements. Section 6 presents our method of generating the XML documents from the specified structure. Section 7 provides details on generating the values within the XML elements. Section 8 contains concluding remarks.

2 Related Work

An early example of using synthetic data to evaluate relational database systems is the Wisconsin benchmark [DeW93]. Our data generator is similar in spirit to the Wisconsin benchmark in that we do not try to assign a real world meaning to the data that we generate. We provide a high degree of control over the characteristics of the generated XML data without requiring it to have a real world interpretation. Other examples of synthetic data that is commonly used in research on relational database systems include TPC-C data [TPCC] and TPC-H data [TPCH].

Synthetic data that is more complex in structure than relational data includes the graph-structured data from the OO7 benchmark for object-oriented database systems [CDN93] and from the BUCKY benchmark for object-relational database systems [CDN+97].

Relational, object-oriented, or object-relational data can certainly be represented in XML, but the XML representation of such data has a fairly simple structure. Our data generator can generate simple-structured data that resembles these kinds of data, but the real power of the data generator is that it can also generate much more complex-structured XML data while providing a much higher degree of control over the data generation process.

We are aware of several proposals for generating synthetic XML data. In [FK99], synthetic XML data is used to evaluate different strategies for storing XML in relational database systems. The XML data used consists of elements at one level with no nesting. The elements are randomly connected in a graph structure using IDREF attributes. This graph-structured view of XML data is useful in some contexts, but XML data is by nature tree structured, and it may often be useful to have a tree-structured view of this data. For example, the important notion of "element containment" only applies to tree-structured XML data. Furthermore, the data generation process of [FK99] has very few opportunities for varying the structure and distribution of the generated data.

In [BR01] and [SWK+01], two benchmarks are proposed for evaluating the performance of XML data management systems. Both benchmarks use synthetic XML data that models data from high-level applications: a database of structured text documents and a directory of these documents in [BR01], and data about on-line auctions in [SWK+01]. The structure of the data in both cases is fixed and simple, and there is very little opportunity for varying it. This kind of data may be adequate for a benchmark that serves as a standard yardstick for comparing the performance of XML data management systems. But if our goal is to test and evaluate a particular XML data management system or a particular feature or component of such a system and to gain insights into its performance, then using XML data with widely varying structure over which we have more control can be more helpful.

IBM provides a data generator that generates XML data that conforms to an input DTD [IBM]. Like the previous approaches, the IBM data generator is limited in the control it provides over the data generation process. For example, we can specify a maximum number of levels for the generated XML documents, but we cannot dictate that these documents have *exactly* this number of levels. Other limitations include using only uniform frequency distributions with no opportunity for generating skewed data.

In contrast to these proposals for generating synthetic XML data, our data generator can generate much more complex data, and it provides much more control over the characteristics of the generated data. Nevertheless, it may be possible to use ideas from these proposals to extend our data generator. For example, IDREF attributes may be used to connect the elements of the generated documents as in [FK99].

Next, we describe the different steps of generating synthetic XML data, and we point out the input parameters that control each step.

```
<A>
   <B/>
   <B>
      <D/>
   </B>
   <C>
      <D/>
      <D/>
   </C>
</A>
```

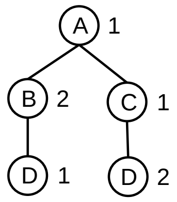


Figure 1: An XML document and its path tree

3 Generating the Tree Structure of the Data

The data generation process starts by generating a tree that specifies the structure of the XML data to be generated. We call this tree the *path tree*.

Every node in the path tree represents a possible path starting from the root of the XML document. The root node of the path tree represents the root element of the XML document. The children of a path tree node represent elements with distinct tag names that are directly nested in the XML elements represented by this node. Thus, every path tree node represents a set of XML elements with the same tag name and reachable by the same path from the root of the document. Every path tree node is labeled with this tag name and with the number of XML elements it represents, which we call the *frequency* of the node. A path tree is an annotated and simplified form of a *DataGuide* [GW97]. Figure 1 presents a simple XML document and its path tree.

At the start of the data generation process, only the structure of the path tree is generated. Path tree nodes are generated without tag names or node frequencies. This information is added to the nodes later in the data generation process.

The input parameters of the data generator include parameters that control the structure of the generated path tree and allow us to generate arbitrarily complex trees. These parameters are the *number of levels in the path tree*, and for every level of the path tree except the lowest level the *minimum and maximum number of children* for nodes at this level. Nodes at the lowest level of the path tree are all leaf nodes that have no children.

The data generator generates the path tree in a depth first manner, from root to leaves. The number of children of an internal path tree node at a particular level of the path tree is chosen at random according to a uniform distribution from the range specified by the input parameters for the possible number of children for nodes at this level.

4 Tag Names – Recursion and Repetition

The next step in the data generation process is assigning tag names to the path tree nodes. This step starts by traversing the path tree in breadth first order and assigning each node a distinct tag name. The tag names used are A, B, C, . . . , Z, AA, AB, AC, etc.

For some usage situations of the data generator, it may be acceptable for path tree nodes to have distinct tag names. For other usage situations, it may be desirable to have *repeated tag names* in the path tree for several reasons. First, it may be desirable to have *recursion* in the generated XML data (i.e., an XML element having the same tag name as its parent or one of its ancestors). Second, it may be desirable to have multiple XML elements with the same tag name reachable by different paths from the root. Third, repeating the tag names of internal path tree nodes allows us to increase the *Markovian memory* of the paths in the path tree. An important property of paths in XML data is that the next tag encountered while traversing the path may depend on some of the previous tags encountered. Such paths can be modeled as a *Markov process*, and the number of previous tags that the next tag depends on is the *order* of this process. When the next tag depends on many previous tags, we say that the paths have a lot of Markovian memory. This property of “the future depending on the present and some part of the past” is often exhibited in real XML data and can be important for some applications. When all nodes of the path tree have distinct tag names, the tag names of the children of a path tree node depend only on the tag name of this node and not on the tag names of its ancestors. In this case, paths in the path tree can be modeled as a Markov process of order 1. When there are repeated tag names in the internal nodes of the path tree, the tag names of the children of a path tree node may depend on the tag names of its ancestors as well as its own tag name. In this case, paths in the path tree can be modeled as a Markov process of order greater than 1, so there is more Markovian memory in the paths of the path tree.

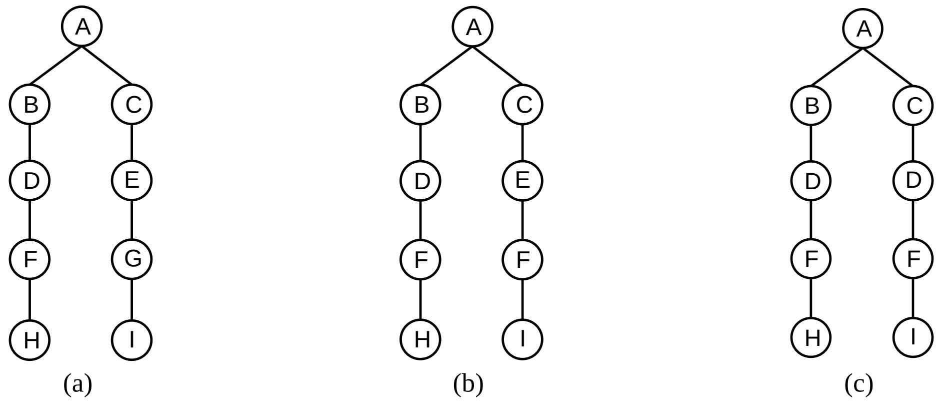


Figure 2: Markovian memory in the path tree due to repeated tag names

For example, consider the path trees in Figure 2. The nodes of the path tree in Figure 2(a) have distinct tag names. Knowing that node F has a child H does not require any information about its ancestors. Paths in this path tree can be modeled as a Markov process of order 1. In Figure 2(b), the internal nodes of the path tree have repeated tag names. In this case, determining whether node F has a child H requires knowing the tag name of its parent. A node F has a child H only if its parent is D. Paths in this path tree can be modeled as a Markov process of order 2. The path tree in Figure 2(c) has even more repetition in the tag names of its internal nodes. Paths in this path tree can be modeled as a Markov process of order 3. We know that a node F has a child H only if we know that its parent is D and its grand parent is B.

After assigning distinct tag names to the path tree nodes, the data generator introduces different kinds of recursion and repeated tag names into the path tree as specified by several input parameters. The data generator introduces *direct recursion*, in which some nodes have the same tag name as their parents, and *indirect recursion*, in which some nodes have the same tag name as one of their ancestors. The data generator also introduces repeated tag names among *internal path tree nodes*, which increases the Markovian memory of the paths in the path tree, and among *leaf nodes of the path tree*, which does not affect the Markovian memory of the paths in the path tree. The data generator can also introduce repeated tag names among nodes of the path tree without restricting whether these nodes are internal nodes or leaf nodes.

5 Element Frequency Distribution

The next step in the data generation process is specifying the frequency distribution of the path tree nodes. One of the input parameters to the data generator is the *total number of XML elements to generate*. This is the total frequency of all path tree nodes. The frequencies of the path tree nodes follow a Zipfian distribution [Zip49]. The *skew parameter* of this distribution, z, is another input parameter of the data generator. $z = 0$ represents a uniform distribution, while $z > 0$ represents a skewed distribution in which some path tree nodes are more frequent than others.

Frequencies are assigned to the path tree nodes in *breadth first order*. An input parameter of the data generator specifies whether these frequencies are sorted in ascending order (i.e., the root node is the least frequent), descending order (i.e., the root node is the most frequent), or random order.

Figure 3 illustrates the different approaches for assigning frequencies to path tree nodes. The figure illustrates three path trees in which frequencies are assigned to nodes in ascending, descending, and random orders. The total number of XML elements that will be generated from these path trees is 31, and $z = 1$.

6 Non-determinism in the Generated Data

The frequencies assigned to the path tree nodes as described in the previous section specify the total number of XML elements to generate for every node. However, they do not specify the number of elements to generate for a path tree node within any particular XML element corresponding to the parent of this node. For example, the path tree in Figure 4 specifies that the XML document has 2 A elements within which are 4 B elements. Figure 4 also shows two XML documents satisfying this definition: a document with 2 B elements within each A element, and one with all 4 B

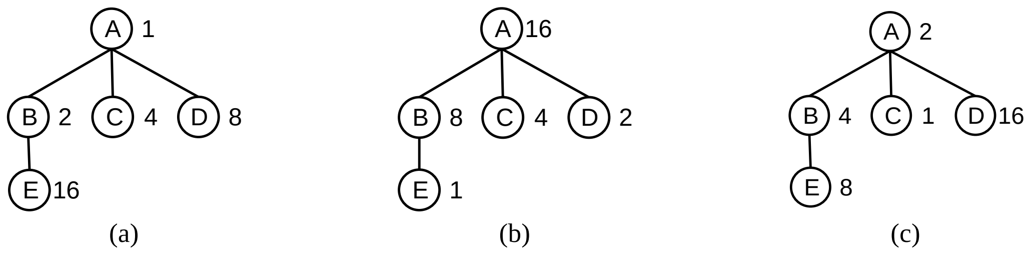
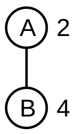


Figure 3: Assigning frequencies to path tree nodes in (a) ascending, (b) descending, and (c) random order



```
<A>
   <B/>
   <B/>
</A>
<A>
   <B/>
   <B/>
</A>
```

```
<A>
   <B/>
   <B/>
   <B/>
   <B/>
</A>
<A/>
```

Figure 4: Non-determinism in the generated data

elements within one `A` element. We say that the first document is more *deterministic* than the second.

The path tree in Figure 4 specifies that the average number of `B` elements in an `A` element is 2. The number of `B` elements in an `A` element can follow any frequency distribution with a mean value of 2. The variance of this frequency distribution specifies the *non-determinism* in the generated XML data. In our data generator, if the average number of elements to generate for a path tree node within an XML element corresponding to its parent node is n, we choose the number of elements to generate for this path tree node in any particular XML element corresponding to its parent from a *uniform distribution* ranging from $(1-p)n$ to $(1+p)n$. The fraction p is an input parameter of the data generator. The greater the value of p, the more the non-determinism in the generated data.

The path tree with tag name and node frequency information, together with the non-determinism parameter p, fully specify the structure of the XML data to generate. The data generator performs a depth first traversal of the path tree and generates XML elements according to this specification. To ensure that the generated XML is well-formed, the data generator adds a top-level $<$`ROOT`$>$ element. Next, we describe our approach to generating values for the XML elements.

7 Element Values

The data generator generates one or more XML documents having exactly the same structure, and it generates values for some of the XML elements in these documents. The generated documents differ only in the values generated for the XML elements. These generated values are strings consisting of one or more *text words*. The text words follow a Zipfian frequency distribution and are controlled by input parameters of the data generator that specify the number of XML documents to generate, the total number of text words to generate in all these documents, and the number of distinct text words to generate. These text words are synthetically generated and have the values `tw1`, `tw2`, `tw3`, etc. The input parameters of the data generator also specify the Zipfian skew parameter, z, of the text word distribution. The Zipfian frequencies from this distribution are assigned to text words in descending order. The first text word, `tw1`, is the most frequent. The input parameters also control the number of XML elements corresponding to internal or leaf nodes of the path tree that have text words.

Text words for the XML elements in all generated documents are generated from the text word distribution in round-robin order (`tw1`, `tw2`, $\ldots$, `tw`m, `tw1`, $\ldots$). When the frequency of a text word is exhausted, we stop using this word. This approach allows us to vary the selectivity of query predicates on the values of the elements in the generated XML documents. Note that it is possible to generate XML documents that have no values in the generated elements.

8 Conclusions

In this paper, we presented a data generator for generating synthetic complex-structured XML data, which can be of use to researchers in XML data management. The data generator has several input parameters that control the characteristics of the generated data. The parameters all have simple and intuitive meanings, so it is easy to understand the structure of the generated data and to set the parameters that are not important for a specific usage situation to reasonable default values. The data generator is publicly available from http://www.cs.wisc.edu/niagara. It can easily be extended and modified to allow for different methods of data generation not covered in this paper. Areas for possible extension include, among others, generating attributes and references, generating data that conforms to a given DTD, and alternative ways of generating text.

While we think our data generator and the ideas it incorporates are useful, our goal at this point is definitely not to claim that it is a finished product. Rather, our goal in writing this paper is to initiate a discussion in the research community with the eventual goal of developing a shared synthetic XML data generation resource.

Acknowledgements

Funding for this work was provided by NSF through grants CDA-9623632 and ITR 0086002, and by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908.

References

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. Submitted for publication, 2001.

[BR01] Timo Böhme and Erhard Rahm. XMach-1: A benchmark for XML data management. In *Proc. German Database Conference (BTW2001)*, Oldenburg, Germany, March 2001.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 12–21, Washington, D.C., May 1993.

[CDN$^+$97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, and Dhaval Shah. The BUCKY object-relational benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146, Tucson, Arizona, May 1997.

[DeW93] David J. DeWitt. The Wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, second edition, 1993.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[IBM] IBM XML generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[Nia] The Niagara project. http://www.cs.wisc.edu/niagara/.

[SWK$^+$01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML benchmark project. Technical Report INS-R0103, CWI, April 2001.

[TPCC] TPC benchmark C. Transaction Processing Performance Council (TPC). Available from http://www.tpc.org/.

[TPCH] TPC benchmark H. Transaction Processing Performance Council (TPC). Available from http://www.tpc.org/.

[Zip49] George K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Reading, Massachusetts, 1949.

[ZND$^+$01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Santa Barbara, California, May 2001.