# Summarizing and Searching Sequential Semistructured Sources[*]

Roy Goldman, Jennifer Widom

Stanford University
{royg,widom}@cs.stanford.edu, http://www-db.stanford.edu

## 1 Introduction

*XML*, the *eXtensible Markup Language* [XML97], is fast becoming the de-facto representation for semistructured data. In the research community, initial work on semistructured databases was based on simple graph-based data models such as the *Object Exchange Model* (*OEM*) [PGMW95]. Though XML and OEM are similar, there are some differences [DFF$^+$99, GMW99], and one of the most significant of these concerns data ordering. OEM and other original semistructured data models are *set-based*: an object has a set of subobjects. However, since XML is a textual representation, any XML document specifies order inherently: an element has a *list* of subelements. Of course, some applications may treat order as an irrelevant artifact of the serialization "forced" by an XML representation. Still, we cannot preclude XML content authors from taking advantage of order. For example, a publications database in XML may represent a publication's author ordering simply by using an ordered list of Author subelements under each Publication.

As researchers have adapted their work on semistructured data to XML, the issue of order already has been addressed at the data model and query language level [DFF$^+$99, GMW99]. In this paper, we focus on the impact of ordered subelements on two important technologies associated with semistructured data: *DataGuides* [GW97] and *proximity search* [GSVGM98].

A *DataGuide* is a concise, accurate structural summary of a semistructured database [GW97]. DataGuides are constructed and maintained dynamically from a database, and they have proved useful for a variety of purposes: browsing, query formulation, storing statistics, query optimization, and most recently compression of XML data [LS00]. DataGuides were defined originally in the context of the OEM model: DataGuides summarize unordered OEM databases, and a DataGuide is itself an unordered OEM object. It is straightforward to use our original DataGuide algorithms to create and maintain unordered XML DataGuides over XML data. However, capturing order in XML DataGuides introduces some new and interesting issues. In this paper we present and evaluate several approaches for creating ordered XML DataGuides that effectively summarize the order in the original XML data.

*Proximity search* is a concept from information retrieval (IR) that we applied to searching graph-structured databases [GSVGM98]. In a traditional IR setting, proximity search is typically implemented with a Near operator and is effective for identifying documents that contain multiple keywords in close proximity—where distance is defined based on the number of characters separating the keywords. But when structure is present, textual nearness is not always appropriate. For example, in an XML publication list an author subelement for a publication could be textually closer to the following publication's title than to its own title. Our proximity search approach takes structure into account by instead considering shortest paths in graph representations of the data. We build special indexes for this purpose, and experience indicates that our approach does an effective job of capturing what proximity search should mean in a structured or semistructured database [GSVGM98]. As with our DataGuide work, however, our proximity search work was based on an unordered data model. In this paper we show how to modify proximity search to incorporate the inherent order of XML subelements. Specifically, we show how to augment the graph representation of XML data such that shortest path computations account for subelement order. We demonstrate the impact of our changes in a sample scenario where subelement order is clearly relevant to proximity search.

Throughout this paper we assume the mapping of XML data to an ordered labeled graph as specified in [GMW99]. In brief, each element and each attribute maps to a node in the (rooted) graph, and edges cor-

[*]This work was supported by the National Science Foundation under grant IIS-9811947 and by NASA Ames under grant NCC2-5278.

respond to element-subelement and element-attribute relationships. Edges are labeled with tags or attribute names as appropriate, and the outgoing edges of each node are ordered (with attribute edges preceding subelement edges by default). The distinction between *semantic mode* (where IDREF attributes become graph edges) and *literal mode* (where XML data always maps to a tree) from [GMW99] has no effect on the algorithms presented in this paper. Our work also can be applied directly to other graph-based data models for XML, e.g., [DFF+99].

## 2 DataGuides

From [GW97], a DataGuide $G$ of a graph-structured source database $D$ is itself a graph such that every label path from the root of $D$ appears exactly once in $G$, and every label path from the root of $G$ appears in $D$.

Consider the following tiny snippet of abstract XML data, contrived to illustrate a point.

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

If we assume unordered subelements, then one valid DataGuide is:

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

Each remaining permutation of the A, B, C, and D subelements forms a valid DataGuide as well.

When we take order into account, we would like to preserve the original definition of a DataGuide as much as possible, but extend the definition to summarize the order of subelements as well as the overall structure. We thus propose to keep the size of the ordered DataGuide the same as the size of the unordered DataGuide, choosing the "best" subelement ordering for each element in the DataGuide. (If we want to store further information about the actual orderings, we can *annotate* the DataGuide elements as in [GW97].)

In our example, intuitively ABCD does the best job of approximating the subelement order for the X instances in the source data: A is the first subelement in 75% of the instances; B follows A in two instances and precedes it in one; C follows A and B in all three instances where they all appear; and D is last in the three instances it's a part of. While it may be easy to choose a "best" order for this simple example, it is a challenge to define the "best" order for an XML DataGuide in general, and the definition could easily change depending on the application. Hence, we have devised several strategies for summarization and report on their effectiveness through an experimental framework.

### 2.1 Problem Formulation

The problem of ordering a DataGuide can be broken down recursively into the problem of ordering the subelements of each DataGuide element. (If we also wish to order the attributes of each element, the problem can be treated in the same way.) Suppose we create an XML DataGuide $G$ of a source database $D$. Consider any element $e$ in $G$, reachable from $G$'s root by some sequence of tags $p$. (By the definition of a DataGuide, $e$ is the only element in $G$ reachable via $p$.) Let $T$ be the set of elements in database $D$ reachable by $p$. (In [GW97] we call $T$ the *target set* of $p$ in $D$.) By the original DataGuide definition, each unique subelement tag of the elements in $T$ appears exactly once as a subelement tag of $e$, and as discussed above we retain this requirement in the presence of order. To order the DataGuide, we must order the subelements of each such element $e$. We will do so based on subelement ordering in all of the elements in $T$.

The problem can be stated more formally (and abstractly) as follows. Consider a set $S = \{\sigma_1, \ldots, \sigma_n\}$ where each $\sigma_i$ is a sequence of labels. Construct a single sequence $\sigma$ of labels that "best" summarizes the sequences in $S$, where $\sigma$ contains each label appearing in any $\sigma_i$ exactly once. $S$ corresponds to target set $T$, $\sigma_1, \ldots, \sigma_n$ to the elements in $T$, and each $\sigma_i$ encodes the subelement ordering of one element of $T$. In our simple example at the start of this section, $S = \{$AABC, ACD, BACD, ABCD$\}$ and we constructed $\sigma =$ABCD.

## 2.2 Algorithms

We now describe three proposed algorithms for solving the problem specified in Section 2.1. Note that for the simple example given at the beginning of Section 2, all three algorithms select ABCD (which was proposed as the best permutation). The algorithms are evaluated experimentally in Section 2.3.

### 2.2.1 Greedy

One option is to use a simple greedy algorithm to generate $\sigma$ from $S = \{\sigma_1, \ldots, \sigma_n\}$. To begin, select the label $L$ that appears at the head of the largest number of sequences in $S$. Label $L$ becomes the first label in $\sigma$. Remove all instances of $L$ from $S$, and repeat the process until all sequences in $S$ are empty. $\sigma$ will contain all labels exactly once. This algorithm is simple and can effectively summarize sequence order in many cases, but there are several situations where it can produce counterintuitive results. Consider:

$S$ = {BABB, BABB, BABB, ABB, ABB, XABB}

For this input, the greedy algorithm will construct BAX. However, this choice does a poor job of reflecting the fact that A precedes B in the data far more often than B precedes A.

### 2.2.2 Edit Distance

A more intricate algorithm can be constructed using *string edit distance* [Gus97], which measures the minimum number of character insertions, deletions, or changes required to transform one string into another. (For example, the words wall and still have an edit distance of 3: starting with wall, we can change the w to s, change the a to t, and insert an i. Note that edit distance is symmetric.)

With a brute force approach, we can consider as candidates for $\sigma$ all permutations of all labels in any $\sigma_i$. Then we compute the sum of the edit distances from each candidate $\sigma$ to all of the sequences in $S$. The $\sigma$ permutation with the minimum overall edit distance is selected. For the example in Section 2.2.1, ABX and AXB tie as the best permutations according to this algorithm.

There are many possible ways to further tune this approach. For example, different costs may be assigned to different edit functions: to account for consecutive labels in an input sequence, we might want to set the cost of a label deletion to be cheaper if it's exactly the same as either adjacent label. Another possibility is to use a non-linear combination of the edit distances from $\sigma$ to the sequences in $S$ to enhance (or mitigate) the impact of any particular sequence within the set.

Unfortunately, this algorithm can be extremely expensive computationally, so pruning strategies would be essential to making it practical in general.

### 2.2.3 Weighted Averages

For our third algorithm, we calculate the average sequence position for every label across all sequences in $S$ and then pick a final sequence that most closely matches the average sequence number for each label. Here, we explicitly collapse consecutive identical labels to compute sequence positions.

More specifically, consider $S = \{\sigma_1, \ldots, \sigma_n\}$. First, for each $\sigma_i$ and each unique label $L$ in $\sigma_i$, we compute $pos(L, \sigma_i)$: the average position of $L$ in $\sigma_i$, after collapsing consecutive identical labels. As a simple example, $pos$(B, AAABBBCC) is 2, since after collapsing consecutive labels B is the second label. When a label appears in more than one position in a sequence, we use the number of consecutive instances at each position to weight the final average. For example, $pos$(B, BBBAB) is 1.5: $(3 \times 1 + 3)/4$, while $pos$(B, BAB) is 2: $(1 + 3)/2$. Finally, let $S_L$ be the set of sequences $\sigma_i$ in $S$ such that $L$ appears in $\sigma_i$. The final average position for $L$ is computed as:

$$\frac{\sum\limits_{\sigma_i \in S_L} pos(L, \sigma_i)}{|S_L|}$$

To illustrate this algorithm, consider the following input data.

$S = \{$AAABCDC, BAC, AAACCCDC$\}$

For this data, we compute the following positions:

$pos$(A, $\sigma_1$) = 1; $pos$(A, $\sigma_2$) = 2; $pos$(A, $\sigma_3$) = 1
$pos$(B, $\sigma_1$) = 2; $pos$(B, $\sigma_2$) = 1
$pos$(C, $\sigma_1$) = 4; $pos$(C, $\sigma_2$) = 3; $pos$(C, $\sigma_3$) = 2.5
$pos$(D, $\sigma_1$) = 4; $pos$(D, $\sigma_3$) = 3

The final positions for A, B, C, and D are approximately 1.3, 1.5, 3.2, and 3.5, respectively, leaving ABCD as the final choice for $\sigma$. For the example in Section 2.2.1, this algorithm selects XAB.

## 2.3 Experimental Framework and Performance Results

To evaluate our different algorithms over large data sets, we created a simple program that generates sets of label sequences with varying characteristics. The language of possible labels consists of the letters A–Z, both in upper and lower case. The program takes a lottery-based approach to picking labels to construct a sequence. Given an input integer parameter $t$, the first label picked will be $t$ times more likely to be an A than any other letter. The second label picked will be $t$ times more likely to be a B than any other letter, and so on, for up to $l$ letters, where $l$ is an input parameter that can vary from 1 to 26. The lower-case letters will be addressed momentarily. Each time a label is selected, we have another "lottery" to determine how many consecutive instances of that label to include in the sequence. We make it equally likely that 1, 2, 3, ..., $f$ consecutive instances are included, where $f$ is an input parameter. A third parameter is $n$, for noise, intended to model the occasional inclusion of atypical labels—as may happen with semistructured data. Before selecting each new label for the sequence, with chances 1 out of $n$ we will insert a randomly selected lower-case letter into the sequence.

As an example let us set $l$=5 (for 5 upper-case labels), $t$=20 (making it quite likely that the letters are selected in order), $f$=5 for moderate repetition, and $n$=10 for some noise. The following results represent one run to generate 10 sequences.

```
cABBBBBBEEEE
cAAAABBBBADDDDBB
AAAAcCCCDDDE
AAAABBBBCCCCDDDEEEE
hAAABnCCDDEE
AAAAAhCCDDDDEE
AAAAAAACDDDDbEE
AEEEECCCBEEE
ABCCCCaDDE
ABBBBCCCDE
```

To compare the effectiveness of our algorithms from Section 2.2, we measure how often each algorithm chooses a "correct" permutation as we vary the input parameters. In this setting, we say a permutation is "correct" if A, B, C, and so on all appear within the permutation in lexicographic order. That is, A precedes B, which precedes C, and so on. We ignore any noise labels when determining whether a permutation is correct. A good algorithm should select a "correct" permutation more often as $t$ increases (since labels are likely to be chosen in order) and as $n$ increases (since noise is less likely to be added between labels). Furthermore, if an algorithm can consistently select the correct result for smaller $t$ and $n$, then it is likely to do well in practice at finding intuitive permutations.

Note that we are not measuring running time in these experiments, only effectiveness in selecting a good permutation. Both the weighted averages and greedy algorithms are linear in running time with respect to the total number of labels in all the sequences, while the edit distance algorithm shows factorial space and time growth in the number of distinct labels across sequences. The number of different "noise" labels

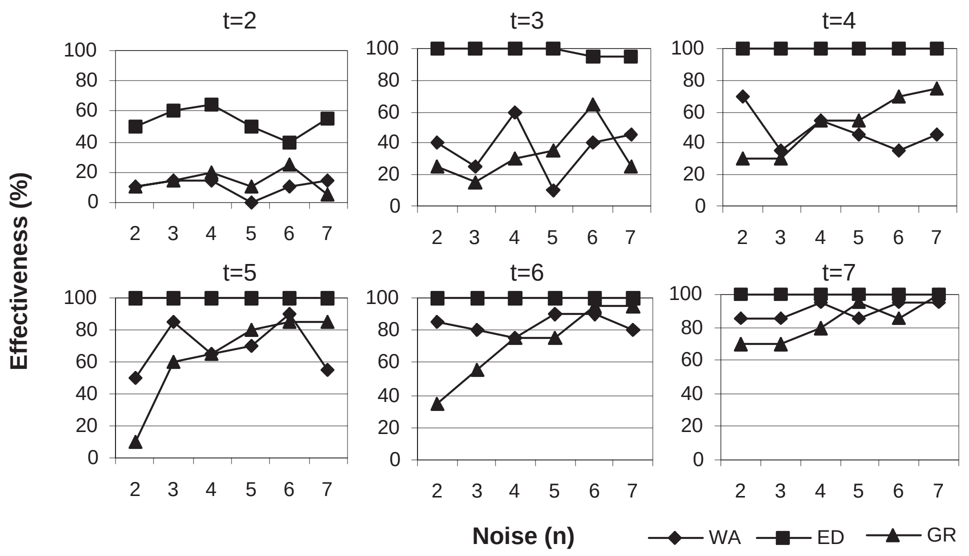

Figure 1: Comparison of ordered DataGuide algorithms

quickly makes the edit distance algorithm infeasible, so in our experiments we make one small adjustment: we consider all permutations of the "primary" labels (A, B, C, etc.) only, then measure edit distances to the original sequences, which include both primary and noise labels.

We compare the effectiveness of our algorithms in Figure 1. Six graphs are presented, corresponding to six different values for $t$ (from 2 to 7). In each graph, we show the effectiveness of all three algorithms (WA for weighted averages, ED for edit distance, and GR for greedy) for six different values of $n$, again from 2 to 7. For each combination of $t$ and $n$, we ran all three algorithms over 20 independent sets of 100 sequences. The effectiveness is the percentage of the 20 runs that return a correct permutation. We set $l = 5$ and $f = 5$ for all experiments, though we observed similar results for other values.

The results show quite conclusively that the edit distance algorithm is the most effective, reaching about 100% effectiveness for all $t \geq 3$, at any value of $n$. When noise is rare, the other two algorithms are similar to each other in accuracy, approaching 100% effectiveness when $t \geq 6$ and $n \geq 6$. The greedy algorithm is the most susceptible to large amounts of noise: it is less effective than the weighted average algorithm when $n \leq 3$.

## 3 Proximity Search

Consider the following sample XML data, representing three Stanford Database Group publications.

```
<PUBLICATION>
   <TITLE>DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases</TITLE>
   <AUTHOR>R. Goldman</AUTHOR>
   <AUTHOR>J. Widom</AUTHOR>
</PUBLICATION>
<PUBLICATION>
   <TITLE>Lore: A Database Management System for Semistructured Data</TITLE>
   <AUTHOR>J. McHugh</AUTHOR>
   <AUTHOR>S. Abiteboul</AUTHOR>
   <AUTHOR>R. Goldman</AUTHOR>
   <AUTHOR>D. Quass</AUTHOR>
   <AUTHOR>J. Widom</AUTHOR>
</PUBLICATION>
<PUBLICATION>
```

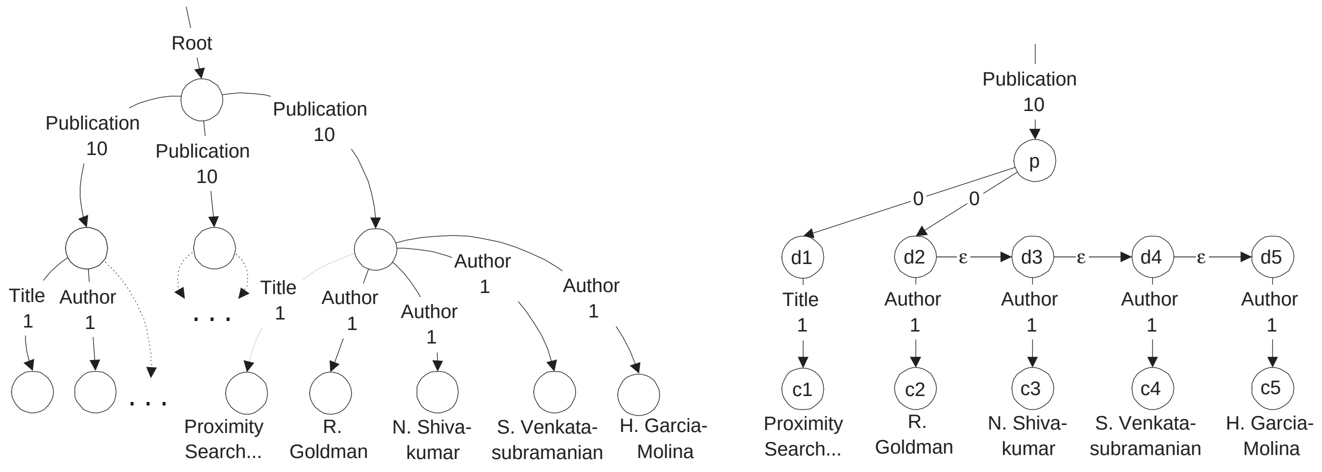

Figure 2: Original and transformed XML graphs

```
    <TITLE>Proximity Search in Databases</TITLE>
    <AUTHOR>R. Goldman</AUTHOR>
    <AUTHOR>N. Shivakumar</AUTHOR>
    <AUTHOR>S. Venkatasubramanian</AUTHOR>
    <AUTHOR>H. Garcia-Molina</AUTHOR>
</PUBLICATION>
```

Consider a search that tries to identify publication titles "near" J. Widom. The traditional definition of textual proximity does not work well in this case: J. Widom is closer in the text to Proximity Search in Databases than to Lore, even though she isn't an author of the former. Our previous work on proximity search addresses exactly this situation [GSVGM98]. We model the data as a graph as described in Section 1, and users can optionally add weights on edges to indicate the "strength" of object-subobject relationships. Distance between data objects is then measured based on the shortest weighted path in the graph, and a special index is built to speed up the computation [GSVGM98]. Even with uniform weights, the graph encoding and shortest path approach to proximity search solves the Title near J. Widom problem above.

Now let us consider the impact of order. In the example above, the order of authors is a very important aspect of the data. If we want to find Publication near Goldman, the publications where R. Goldman is a first or second author should rank higher than those where R. Goldman is a later author. To incorporate order into our proximity search framework, we preprocess each XML data graph before building our proximity search index, adding new objects and weighted edges that adjust the weighted shortest paths between objects in order to reflect ordering. For example, the left side of Figure 2 shows the XML data above modeled as a graph, with some simple weights to illustrate our approach. The right side of Figure 2 shows the transformed graph for the rightmost publication subtree. As can be seen, the first Author is distance 1 from the parent publication element, the second author is distance $1+\epsilon$ (for some small value $\epsilon$), the third is distance $1+2\epsilon$, etc. The distance between the first and second author is $2+\epsilon$, the same as the distance between the second and third author and the distance between the third and fourth.

In this case we decided that the Title subelement and the first Author should be the same distance from the parent. However, if we preferred to take into account the fact that Title precedes the first Author, we could insert an $\epsilon$ edge between nodes $d_1$ and $d_2$ and remove the edge from $d_2$ to $p$.

In the general case, consider an element $p$. We partition $p$'s list of subelements into sublists, where order is relevant within but not between sublists.[1] For each subelement $c_i$ of $p$, we "disconnect" $c_i$ from $p$ and create a new parent $d_i$ of $c_i$. The weight on the $d_i \rightarrow c_i$ edge is the weight from the original $p \rightarrow c_i$ edge.

[1]This partitioning could be provided by the XML content author. As a default when not specified, we either create one sublist containing all subelements or we partition based on repeating XML tags. In our example, tags Title and Author partition the subelements into two sublists. When present, XML attributes can be grouped into their own sublist.

We then connect adjacent $d_i$ nodes within each sublist with edges of weight $\epsilon$, and finally add an edge with weight $0$ from $p$ to the first $d_i$ for each sublist. Note that it is well-defined to perform this transformation on arbitrary graphs.

The transformation effects some important properties. Consider $p$ and one of its sublists $L$ such that the weight from $p$ to every element in $L$ is $W$. First, we see that any two adjacent siblings are now $2W + \epsilon$ apart. More generally, the $i$th and $j$th siblings are $2W + (|i - j| \times \epsilon)$ apart, assuring that sibling distances grow monotonically as their separation within the list increases. Also, the distance from $p$ to the $i$th element (counting from 0) of $L$ is $W + (i \times \epsilon)$, assuring that distance from the parent to its children grows monotonically beginning with the first child.

## 3.1 Examples

To demonstrate the impact of our transformation, we built two proximity indexes over a real XML database for the Stanford Database Group, which contains information about our group members, projects and publications (http://www-db.stanford.edu/lore). One index was based on the original graph, and the other based on the transformed graph to take ordering into account. The format of the publication data is similar to the XML shown above, though there is only one XML element per unique author, referenced (via an IDREF) from all of his or her publications.

We performed several searches using both indexes and compared results. The result of a proximity search is a ranked list of *Find* elements, where the score of each is based on proximity to all in the *Near* set. The score also is influenced by several tuning parameters described in [GSVGM98]; we use the same default parameters as in [GSVGM98]. We describe results from two representative searches.

First, we performed the search Find Publication near Goldman. All publications where R. Goldman was the first author received the highest score, followed by publications where he was the second author, etc. In the original scheme, all of R. Goldman's publications received the same score.

Next, we performed the search Find Author near Goldman. The XML element for R. Goldman received the highest score. Next was an eight-way tie among S. Abiteboul, S. Chawathe, A. Crespo, H. Garcia-Molina, J. McHugh, D. Quass, N. Shivakumar, and V. Vassalos. All were adjacent to R. Goldman in an author list except for A. Crespo and V. Vassalos. The latter two are non-adjacent co-authors of R. Goldman's, but they have other relationships to R. Goldman in the database (e.g., they worked on the same research project). The lowest non-zero scores were given to both S. Venkatasubramanian and Y. Zhuge, representing the largest "separation" between R. Goldman and his co-authors. (On one paper, R. Goldman was first author and S. Venkatasubramanian was fourth, and on another, R. Goldman was second and Y. Zhuge was fifth.) In the original scheme, all of R. Goldman's co-authors tied for second place behind the element for R. Goldman himself.

## References

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW8)*, Toronto, Canada, 1999.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, Philadelphia, Pennsylvania, June 1999.

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 26–37, New York, New York, August 1998.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, England, 1997.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[LS00] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, May 2000.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[XML97] World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/ TR/WD-xml-lang-970331.html, December 1997.