Approximate DataGuides

Roy Goldman, Jennifer Widom
Stanford University
{royg,widom}@cs.stanford.edu
www-db.stanford.edu

Abstract

DataGuides are concise and accurate summaries of semistructured databases, enabling schema exploration and improving query processing. Unfortunately, DataGuides can be very expensive to compute, especially for large, cyclic databases. For many DataGuide uses, an "approximate" summary of the database's structure can be beneficial yet much cheaper to compute. We summarize several uses of DataGuides and define *Approximate DataGuides (ADGs)*, which relax certain aspects of the DataGuide definition. An ADG allows some inaccuracy yet retains properties that make it useful in numerous situations. The core of the paper presents two general approaches for building ADGs, describing algorithms and experimental results.

1 Introduction

A *DataGuide* is a concise and accurate structural summary of a semistructured database. Originally proposed in [GW97], DataGuides have a variety of uses that enable query formulation and processing in a semistructured database management system [MAG+97]:

- *UI*: as a user interface to help users explore database structure and submit queries "by example"
- *Statistics*: to store statistics about the shape of the database, for use by a query optimizer [MW97]
- *Warnings*: as a tool to provide warnings about queries that refer to nonexistent paths
- *Path Expressions*: to enable compile-time expansion of regular path expressions [MW98]
- *Path Index*: as a path index to speed up query processing [GW97]

Semistructured databases usually are modeled as labeled, directed graphs [Abi97, Bun97]. For a tree-shaped database, DataGuide construction is linear in space and time with respect to the size of the database. For a general graph, however, the algorithm is exponential in the worst case. For many graph-structured databases, experiments show that DataGuides can be computed quickly and are much smaller than the original database [GW97]. Still, we have seen examples—particularly cyclic databases—where DataGuide construction is prohibitively expensive. In this paper we propose *Approximate DataGuides (ADGs)*. By relaxing the definition of a DataGuide, we can provide many of the benefits of a DataGuide yet avoid the associated performance traps.

We present our work in the context of *OEM* [PGMW95], a popular model for semistructured data where a database is a rooted, directed graph, with textual labels on edges and atomic values in leaves. A DataGuide is itself an OEM graph G that corresponds to an OEM database D, such that every distinct label path from the root of D appears exactly once as a path from the root of G, and every label path from the root of G has at least one matching label path in D.

Quite simply, an ADG drops the second requirement that all DataGuide paths must exist in the original database. Therefore an ADG may have "false positives" but never "false negatives" concerning the existence of database paths. Reconsider the five DataGuide uses above:

- *UI*: A user exploring an ADG may see paths that do not actually exist in the database.
- *Statistics*: We can still associate statistics with every ADG object, hence we can store statistics for every rooted path in the database. However, some statistics may be based on a superset of the actual objects reachable along that path.
- *Warnings*: The system may fail to warn the user that certain path expressions do not exist, but it will never incorrectly warn that a valid path does not exist.
- *Path Expressions*: Expanding to a superset of valid path expressions is not harmful, although it can degrade efficiency [MW98].
- *Path Index*: An index is expected to be exact, so an ADG is not usable.

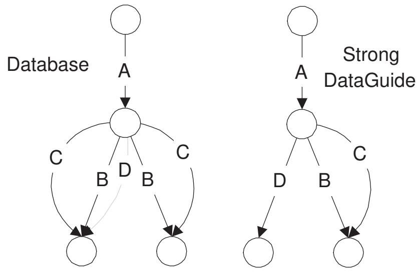

Figure 1: A sample OEM database and its strong DataGuide

We propose several strategies for building effective ADGs. In each case, we identify "similar" portions of the DataGuide and merge them. It is this merging process that may introduce superfluous paths. We see it as a requirement that all merging occur during construction—rather than as a post-processing step—because constructing a (regular) DataGuide is exactly the performance bottleneck we are trying to avoid. We discuss two general approaches to approximation:

- *Object Matching*: This approach is based on the hypothesis that two label paths in a database are "similar" if the sets of objects reachable via those paths are similar, i.e., they have a significant intersection.
- *Role Matching*: For this class of approximation, we decide whether two label paths are similar based on the paths themselves, without regard to the objects they reach.

Note that related work from [NAM98] gives algorithms for finding "approximate typings" of semistructured databases based on patterns of incoming and outgoing edges. In comparison, we are less concerned with extracting a set of object types; rather, our goal is to provide a structural summary that allows a semistructured database system (or a user of one) to quickly extract information about label paths in the database. In other related work, choosing an effective ADG can also be thought of as fitting a "good" *graph schema* to an existing database [BDFS97].

2 Object Matching

In [GW97], we specify the definition of a *strong* DataGuide, which allows us to store annotations—such as statistics and sample values—with each label path in the DataGuide. The definition of a strong DataGuide is based on *target sets*. The target set of a path is the set of all objects reachable via that path. In a strong DataGuide, each DataGuide object corresponds to the target set of all label paths that reach that DataGuide object. Two label paths in the DataGuide point to the same DataGuide object if and only if the target sets of both label paths are exactly the same in the original database. There exists exactly one strong DataGuide for any database. Figure 1 shows a small database and its strong DataGuide. Notice that paths A.B and A.C in the DataGuide point to the same object because the target sets of A.B and A.C are the same in the database. The path A.D gets its own object in the DataGuide because its target set is different from the others.

The algorithm presented in [GW97] creates a strong DataGuide by performing a depth-first exploration of the database, building up target sets of the label paths visited. Each target set is stored in a hash table. Each time the graph exploration generates a target set, the algorithm checks the hash table to see if that target set has already been discovered. If not, then a DataGuide object is created for that specific target set, the target set is added to the hash table (along with its corresponding DataGuide object), and the new object is linked into the DataGuide according to the path used to reach the target set. If, on the other hand, we have already seen the target set, then we can find its corresponding DataGuide object, and we add an edge in the DataGuide to that existing object.

Suppose that instead of requiring target sets to be exactly equal before equating their corresponding DataGuide objects, we instead allow DataGuide paths to point to the same object when their target sets are "almost" equal. In doing so, we may introduce DataGuide paths that do not exist in the database (false positives).

Consider Figure 2(a), depicting an actual database about the Stanford database group (DBGroup). This database has information about the members, projects, and publications of the group. Compare the target sets of DBGroup.GroupMember (52 objects) and DBGroup.Project.ProjectMember (38 objects). Because the target sets are

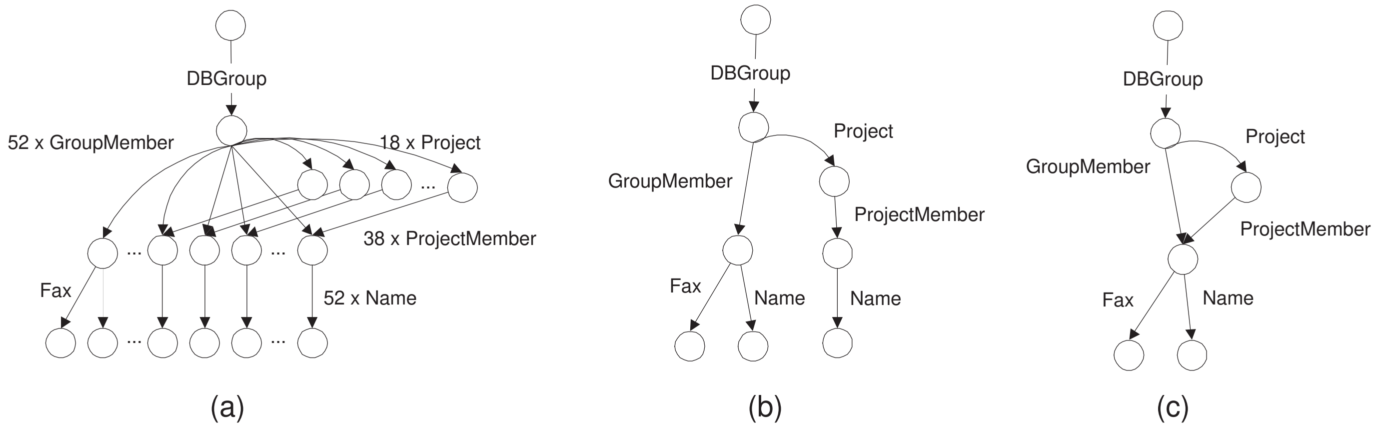

Figure 2: An OEM database, its strong DataGuide, and an approximate DataGuide

not identical, each will correspond to a different strong DataGuide object. Note that all GroupMembers have a Name, and one who is not a ProjectMember also has a Fax. Figure 2(b) is the strong DataGuide for Figure 2(a).

Suppose that when comparing the target sets for the GroupMembers and ProjectMembers, we merged the corresponding DataGuide objects because of their significant intersection. We then build the "sub-DataGuide" over the union of the GroupMembers and the ProjectMembers. This case is shown in Figure 2(c). Performing the merge of DataGuide objects can introduce false paths: in our case the ADG incorrectly suggests that at least one ProjectMember has a Fax.

This object-matching approach to DataGuide approximation introduces many interesting issues:

- How do we define whether two sets are "similar"? One simple criterion (used in the remainder of this paper) is to consider two sets X and Y similar when $|X \cap Y|$ / $\max(|X|, |Y|)$ is above some threshold t.
- How does the DataGuide construction algorithm change? Again, we need to make our approximations during construction rather than reducing a constructed (full) DataGuide. This on-line approach unfortunately gives some importance to the way we traverse the original database to construct target sets. For example, we may decide that sets X and Y are similar enough to merge them into Z. At this point, the original sets disappear. Suppose we then encounter another set W. W may be similar to X, but not to the newly created set Z. If we would have traversed the database differently, W and X may have been merged. Our intuition is that choosing different traversals will not have a severe effect on the overall size and structure of ADGs, but experiments are needed for verification. In addition, we may want to limit the number of times any given (original) target set can participate in a merge operation in an effort to bound the difference between a target set and the final object set it is a part of. We plan to consider other possible approaches for dealing with this anomaly.

  Suppose that we have already constructed the "sub-DataGuide" for some target set X. If we encounter some target set Y that we decide is similar to X, there are two possible scenarios for the algorithm: if Y is a subset of X we can simply merge the ADG objects and halt further processing of X, since we know that Y cannot introduce any new paths that were not considered when processing X. On the other hand, if Y is not a subset, it is necessary to continue by examining the "union" of the substructure of objects in X with those in Y. We can reuse our incremental maintenance algorithms from [GW97] to minimize the amount of redundant work.
- How do we efficiently decide whether two sets are similar? Recent work has shown that we can efficiently determine whether two sets have a high percentage of elements in common [BGMZ97]. But the decision becomes more expensive as the threshold similarity percentage drops, since we cannot disqualify a potential match as quickly.

Similarity Threshold	Objects	Edges	False Paths
100% (Strong)	273	366	-
95%	241	308	0
90%	214	256	2
80%	200	241	6
70%	170	238	9
50%	117	140	9
30%	115	140	9
15%	110	138	9
1%	65	124	21

Table 1: Object-matching ADGs for the DBGroup database

3 Object Matching Experiments

For our experiments, we focused on the size and accuracy of the ADGs rather than absolute speed of construction, since for this initial work we used a simple, untuned B-tree-based data structure for computing set similarity.

We begin by testing the object matching approach over our DBGroup database, which contains about 3600 objects and 4200 edges. The database is highly cyclic, and while the overall structure is regular there are many "islands" of irregularity and incompleteness. The first row of Table 1 shows the size of the strong DataGuide. The remaining rows show the different ADG sizes for varying similarity threshold percentages. Quantifying the level of approximation is a challenge. As one simple metric, we counted how many false paths appear in the ADG but not in the strong DataGuide. Using depth-first search, once we determine that a path p is false, we do not continue to count paths for which p is a prefix, since of course they will be false as well.

Next, we attempted to analyze a 4MB subset of the Internet Movie database (www.imdb.com), a highly cyclic semistructured database with information about movies, actors, directors, producers, writers, etc. The database has about 60,000 objects and 95,000 edges. Unfortunately, our strong DataGuide algorithm did not terminate before exhausting resources: because of certain kinds of database cycles, the algorithm generated many very long paths (over 1000 labels) without finding a repeated target set. We were hopeful that the ADG would perform better, but unfortunately we hit the same problem. The algorithm generated too many small, nearly disjoint target sets that did not merge. Thus, while object-matching ADGs are efficient and effective when the number of target sets is manageable, the algorithm still is too expensive for certain larger, cyclic databases.

4 Role Matching

Rather than approximating DataGuides based on target sets, another approach is to merge DataGuide objects based on label paths (*roles*). More formally, we consider building ADGs based on Boolean *path merging* functions. If such a function $M(p_1, p_2)$ returns *True* for label paths p_1 and p_2, then paths p_1 and p_2 will point to the same ADG object. We discuss two possible merging functions.

4.1 Suffix Matching

In basic *suffix matching*, the merging function $M(p_1, p_2)$ returns *True* if and only if the last labels of p_1 and p_2 are the same. This approximation restricts the ADG to have one object per label. Figure 3(a) shows a sample database fragment, and its suffix-matching ADG is shown in Figure 3(b). Note that a suffix-matching ADG is essentially the same as the *1-Representative Object* described in [NUWC97], the original Stanford paper on which DataGuides were based.

The suffix-matching ADG is straightforward to characterize. While we can create it with a merging variant of the DataGuide construction algorithm, a simpler method is just to build a hash table: for each label l, we store information about all of the labels that directly follow l in the database. For the final step, we can construct the ADG by identifying the root label and walking what is essentially an adjacency-list graph representation inside the hash table. Construction

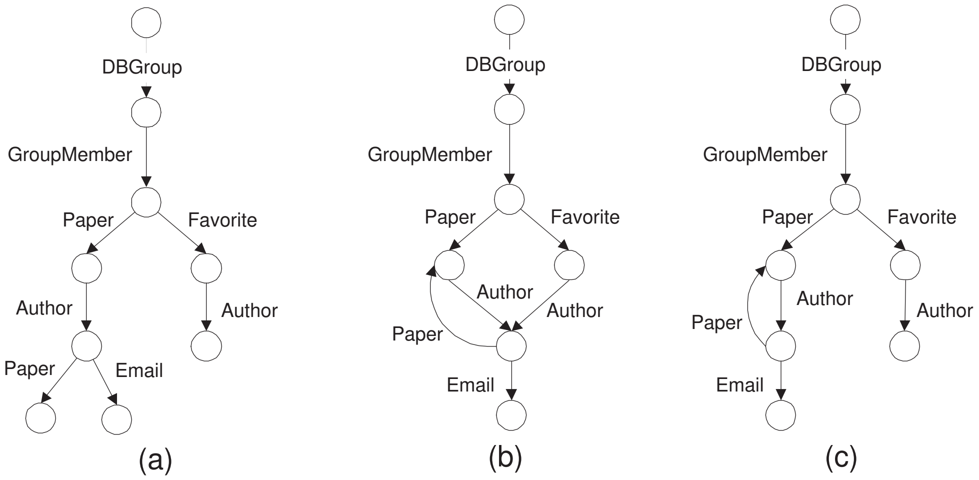

Figure 3: A sample OEM database, its suffix-matching ADG, and its path-cycle matching ADG

time is at worst quadratic in the size of the database, since building the hash table requires examination of all paths of length 2.

The suffix-matching ADG is very effective when each label consistently identifies the "same type" of object. As one example where it could be problematic, consider our DBGroup database, where the Author label is used to identify both the authors of group publications and authors of members' favorite books. The suffix-matching ADG implies that Asimov, Salinger, and Kerouac may have Stanford email addresses! To help alleviate such problems, a natural extension to this approach is to match suffixes of length 2, 3, or k. As described in [NUWC97], we can generalize the hash-table approach to k-length suffixes, and that paper also proposes several algorithms for building more compact representations.

4.2 Path-cycle Matching

As an alternative to matching suffixes of a particular length, we consider a different path merging function that specifically addresses DataGuide performance problems caused by cyclic databases. Note that a strong DataGuide can have cycles itself when target sets are repeated along a path. But for larger databases, experience shows that paths grow to giant lengths before reaching an identical target set. As mentioned in Section 2, the problem persists even when we are willing to settle for similar target sets. Hence, we encode in a path merging function the following heuristic: if we see a specific label more than once along a path from the root, we assume that we have hit a "semantic" cycle and we merge the paths. For example, in our DBGroup database, suppose that at some point we create an ADG object for the path DBGroup.Paper. As we continue to explore this path, we create a new ADG object for DBGroup.Paper.Author, but when we encounter DBGroup.Paper.Author.Paper we assume that seeing Paper again indicates a schema cycle. Hence, we point back to the ADG object for DBGroup.Paper. This *path-cycle matching* ADG is shown in Figure 3(c); note that this approach avoids the suffix-merging problem of combining paper authors with group members' favorite authors.

Within our merging function framework, the path-cycle matching function $M(p_1, p_2)$ returns *True* if and only if p_1 is a prefix of p_2 (or p_2 is a prefix of p_1) and the last labels of p_1 and p_2 are the same.

5 Role Matching Experiments

For our experiments we modified the depth-first object-matching algorithm to merge ADG objects instead based on either suffixes or path-cycles. Table 2 shows experimental results for both types of approximations on the DBGroup and Movies databases introduced in Section 3. Again, the false paths column is in comparison to the strong DataGuide, which we were unable to generate for the Movies database.

Note that the suffix approximation produced numerous false paths for the DBGroup database, many of which were due to the Author label problem described in Section 4.1. Using suffixes of length 2 would fix the problem for

Database	Approximation	Objects	Edges	False Paths
DBGroup	Suffix	102	134	240
DBGroup	Path-cycle	240	317	3
Movies	Suffix	38	63	-
Movies	Path-cycle	76	96	-

Table 2: Role-matching ADGs

this database. Another interesting fact is that the smallest DBGroup object-matching approximation from Section 3 is actually smaller than the suffix-matching ADG, due to the fact that the database has many objects serving multiple roles (e.g., members as authors, project members, advisors, etc.). As could be expected, in both databases the path-cycle approximation is significantly larger than the suffix match. Perhaps the most striking results are the tiny sizes of the role matching approximations for the Movies database, given that we could not even build the strong DataGuide (or the object-matching approximation) for this database.

6 Conclusion

Since the space of possible semistructured databases is enormous and varied, it is difficult to choose the best approximation for every situation. Nonetheless, we can summarize the best and worst features of strong DataGuides and the ADGs we've described in this paper.

- *Strong DataGuides*: Strong DataGuides are always accurate and can be used as a path index, something for which we cannot use ADGs. For tree-structured, acyclic, or smaller cyclic databases, strong DataGuides usually perform well. For larger cyclic databases, it may be better to use an ADG. For path indexing, we plan to explore recent work by Milo and Suciu [MS99], which proposes a graph-based path indexing structure that relaxes the DataGuide (and ADG) constraint of a database label path existing only once in the index. (It would be interesting to explore this approach in general, although for some DataGuide uses we do depend on a path existing only once, such as for the user interface.)

- *Object Matching*: This approach approximates a DataGuide based on objects having multiple incoming paths. Hence, it is an approximation for graph-structured databases only; for trees, a strong DataGuide is generated. An adjustable threshold lets administrators tune the level of approximation. Unfortunately, however, the algorithm can still be prohibitively expensive for large, cyclic databases.

- *Suffix Matching*: The best feature of suffix matching is its predictable construction performance. The algorithm also is not biased to rooted paths—it provides information about path suffixes wherever they may appear in a database. Unfortunately, this approximation can yield skewed summaries and statistics if labels are used in different ways throughout a database. We can increase the suffix length that we match to increase accuracy, although doing so makes the algorithm more expensive.

- *Path-cycle Matching*: This approximation addresses problems caused by cyclic data without bias to paths of any specific length. Unfortunately, it is difficult to characterize just how computationally expensive this approach is; we plan to investigate the matter and perform additional experiments.

Ultimately, the "best" ADG may depend on the database we are summarizing. Further, it may be possible to combine some of the above techniques, such as path-cycle and object matching. An interesting avenue of research is to devise strategies that can efficiently analyze a database and select an approximation that will be quick to create and reasonably accurate as well, perhaps with statistical guarantees on the quality of approximation.

Another important issue we haven't tackled here is ADG maintenance. While we can use a variation of incremental DataGuide maintenance algorithms proposed in [GW97], there may be opportunities for better performance. For example, any ADG will remain an ADG after a database deletion (by definition). Another possibility is to use invalidation rather than incremental maintenance. An ADG may still be quite useful even if particular regions are marked "invalid" due to updates. These regions could be recomputed in batch in the background, or perhaps the entire ADG could be regenerated when the percentage of invalid regions crosses some threshold.

References

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[BGMZ97] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, April 1997.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997. Tutorial.

[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.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MS99] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, Jerusalem, Israel, January 1999.

[MW97] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Stanford University Database Group, November 1997. Available at URL http://www-db.stanford.edu/pub/papers/qo.ps.

[MW98] J. McHugh and J. Widom. Compile-time path expansion in Lore. Technical report, Stanford University Database Group, November 1998. Available at URL http://www-db.stanford.edu/pub/papers/re.ps.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, England, April 1997.

[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.