Efficiently Maintaining Structural Associations of Semistructured Data

Dimitrios Katsaros

Department of Informatics, Aristotle University of Thessaloniki
Thessaloniki, 54124, Greece
dkatsaro@csd.auth.gr

Abstract. Semistructured data arise frequently in the Web or in data integration systems. Semistructured objects describing the same type of information have similar but not identical structure. Finding the common schema of a collection of semistructured objects is a very important task and due to the huge volume of such data encountered, data mining techniques have been employed. Maintenance of the discovered schema in case of updates, i.e., addition of new objects, is also a very important issue. In this paper, we study the problem of maintaining the discovered schema in the case of the addition of new objects. We use the notion of "negative borders" introduced in the context of mining association rules in order to efficiently find the new schema when objects are added to the database. We present experimental results that show the improved efficiency achieved by the proposed algorithm.

1 Introduction

Much of the information that is available on-line, is semistructured [1]. Documents like XML, BibTex, HTML and data encountered in biological applications are examples of such information. The intrinsic characteristic of semistructured data is that they do not have a rigid structure, either because the data source does not force any structure on them (e.g., the Web) or because the data are acquired from various heterogeneous information sources (e.g., in applications that use business-to-business product catalogs, data from multiple suppliers – each with its own schema – must be integrated, so that buyers can query them).

It is quite common that semistructured objects representing the same sort of information have similar, though not identical, structure. An example of semistructured objects is depicted in Figure 1, where a portion of semistructured "fish" objects, maintained by the "Catalogue of Life" database (found in URL http://www.sp2000.org), is illustrated.

Finding the common schema of a large collection of semistructured objects is very important for a number of applications, such as querying/browsing information sources, building indexes, storage in relational or object oriented database systems, query processing (regular path expressions), clustering documents based on their common structure, building wrappers.

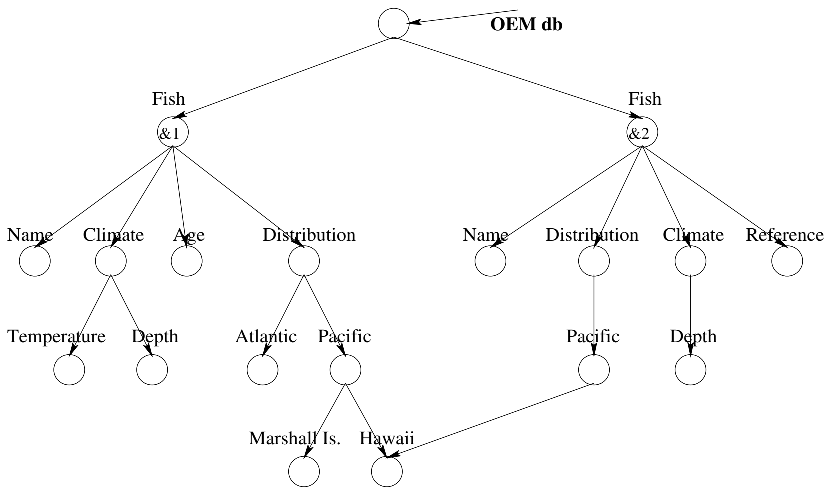

Fig. 1. A portion of "fish" objects

Semistructured schema discovery is a challenging task, mainly for two reasons. The first is the huge volume of data and the second is their irregularity. Several approaches targeting at this goal have been done [8, 14, 13, 10] to name a few. Due to the huge amount of data to be processed, the primary requirement for the algorithm employed is its scalability, both in terms of input and output data. The algorithm presented by Wang and Liu in [13], *WL* in the sequel, meets this requirement. Its objective is to discover all "typical" structures (substructures) that occur in a minimum number of objects specified by the user. *WL* is based on the association rule discovery [2] paradigm.

When insertions take place into the collection (i.e., an "increment" database is added into the "regular" database), which is a very frequent situation in the Web, then the set of the aforementioned "typical" structures may change. Thus, arises the need to maintain the set of discovered structures.

1.1 Motivation

The only work addressing this issue is that reported in [15], which presents the algorithm *ZJZT*. They adopt the FUp [4] algorithm, that was proposed for the maintenance of the discovered large itemsets from a transaction database. In each iteration, *ZJZT* makes a scan over the whole updated database. The increment database is scanned first and the results are used to guide the mining of the regular database. The number of iterations is k, where k is the size of the largest (in terms of the number of path expressions it contains) *tree expression* (the definition of tree expression is presented in Section 2).

Incremental schema maintenance for semistructured data as addressed by *ZJZT*, suffers from two main drawbacks. The first is that the employed algorithm is inefficient, since it requires at least as many passes over the database as the "size" of the longest *j-sequence*. Even in the case that the new results are merely a subset of the old results, that is, the updates do not modify the schema, their algorithm will make the same constant number of passes over the database. The second is that their method cannot provide the mining results for the increment database itself. These results are important in order to discover temporal changes in the schema and drive decisions regarding storage issues [5]. So, we employ the notion of *negative borders* [3, 6, 9, 11] in order to efficiently deal with the problem of efficient incremental schema maintenance for semistructured data.

1.2 Contributions

In this paper, we deal with the problem of how to efficiently maintain the discovered schema (structural associations) of a collection of semistructured objects in the case of insertions of new objects into the collection. We utilize the notion of *negative borders* [7] and devise the $\mathcal{D}eltaSSD$ algorithm, which is an adaptation of the *Delta* algorithm [9], in order to efficiently find the new schema of the collection.

We present a performance evaluation of $\mathcal{D}eltaSSD$ and a comparison with existing algorithms using synthetic data. The experiments show that $\mathcal{D}eltaSSD$ incurs the least number of database scans among all algorithms, which indicates its superiority.

The rest of this paper is organized as follows: Section 2 defines the problem of the incremental maintenance of semistructured schema. Section 3 presents the proposed algorithm $\mathcal{D}eltaSSD$. Section 4 presents the experimental results and finally, Section 5 contains the conclusions.

2 Incremental Schema Mining

For our convenience, we recall some definitions from [13] and some features of the *WL* and *ZJZT* algorithms.

2.1 Overview of the *WL* Algorithm

We adopt the *Object Exchange Model* [1] for the representation of semistructured objects, where each object is identified by a unique identifier $\&a$ and its value $val(\&a)$. Its value may be *atomic* (e.g., integer, string), a *list* $\langle l_1 : \&a_1, l_2 : \&a_2, \cdots, l_n : \&a_n\rangle$ or a *bag* $\{l_1 : \&a_1, l_2 : \&a_2, \cdots, l_n : \&a_n\}$[1], where each l_i identifies a label. For the incremental schema maintenance problem (to be defined shortly after) the user must specify some objects, called *transaction objects* and denoted as $\top$, whose common structure we are interested in identifying (e.g., in Figure 1 the transaction objects are the fish objects &1 and &2).

[1] Order does matter in a list but it does not in a bag. We deal only with nodes of list type, since our target is ordered semistructured data (e.g., XML).

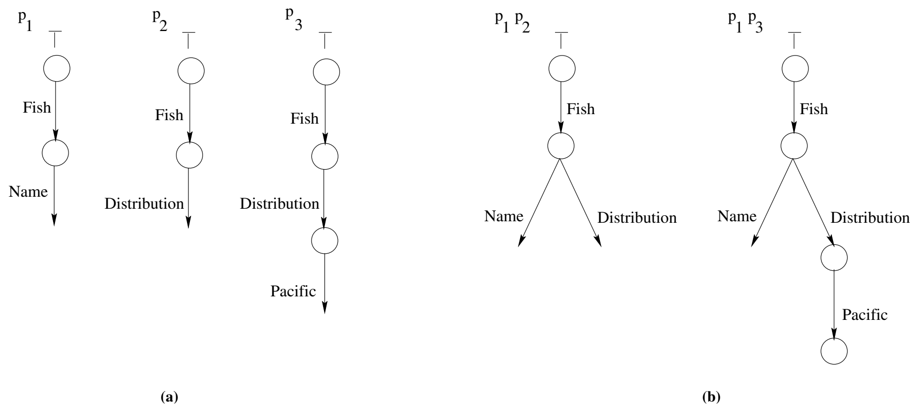

Fig. 2. Examples of tree expressions

Definition 1 (Tree Expressions). *Consider an acyclic OEM graph. For any label l, let l^* denote either l or the wild-card label ?, which matches any label.*

1. *The nil structure $\perp$ (that denotes containment of no label at all) is a tree-expression.*
2. *Suppose that te_i are tree expressions of objects a_i, $1 \leq i \leq p$. If val(&a)= $\langle l_1 : \&a_1, l_2 : \&a_2, \cdots, l_p : \&a_p \rangle$ and $\langle i_1, i_2, \cdots, i_q \rangle$ is a subsequence of $\langle 1, \cdots, p \rangle$ $q > 0$, then $\langle l^*_{i_1} : te_{i_1}, \cdots, l^*_{i_q} : te_{i_q} \rangle$ is a tree-expression of object a.*

Therefore, a tree expression represents a partial structure of the corresponding object. A *k-tree-expression* is a tree-expression containing exactly k leaf nodes. Hence, a 1-tree-expression is the familiar notion of a *path expression*. Figure 2(a) shows three 1-tree expressions, p_1, p_2, p_3. Each k-tree-expression can be constructed by a sequence of k paths $(p_1, p_2, \cdots, p_k)$, called *k-sequence*, where no p_i is a prefix of another. For example, Figure 2(b) illustrates two 2-tree expressions. In the same example, we see that we can not combine the 1-tree expressions p_2 and p_3 to form a 2-tree expression, since the former is prefix of the latter.
In order to account for the fact that some children have repeating outgoing labels, *WL* introduced superscripts for these labels. Hence, for each label l in *val*(&a), l^i represents the i-th occurrence of label l in *val*(&a). Consequently, a k-tree-expression can be constructed by a k-sequence $(p_1, p_2, \cdots, p_k)$, each p_i of the form $[\top, l^1_{j_1}, \cdots, l^n_{j_n}, \perp]$. Although Definition 1 holds for acyclic graphs, in [13] cyclic OEM graphs are mapped to acyclic ones by treating each reference to an ancestor (that creates the cycle) as a reference to a terminating leaf node. In this case, the leaf obtains a label which corresponds to the distance of the leaf from its ancestor (i.e., the number of intermediate nodes). For this reason, henceforth, we consider only acyclic OEM graphs. Additionally, *WL* replicates each node that has more than one ancestors. The result of the above transformation is that each object is equivalently represented by a tree structure.

Definition 2 (Weaker than). *The nil structure $\perp$ is weaker than every tree-expression.*

1. *Tree-expression $\langle l_1 : te_1, l_2 : te_2, \cdots, l_n : te_n \rangle$ is weaker than tree-expression $\langle l'_1 : te'_1, l'_2 : te'_2, \cdots, l'_m : te'_m \rangle$ if for $1 \leq i \leq n$, te_i is weaker than some te'_{j_i}, where either $l'_{j_i} = l_i$ or $l_i =?$ and $\langle j_1, j_2, \cdots, j_n \rangle$ is a subsequence of $\langle 1, 2, \cdots, m \rangle$.*
2. *Tree-expression te is weaker than identifier &a if te is weaker than val(&a).*

This definition captures the fact that a tree-expression te_1 is weaker than another tree-expression te_2 if all information regarding labels, ordering and nesting present in te_1 is also present in te_2. Intuitively, by considering the paradigm of association rules [2], the notion of tree expression (Definition 1) is the analogous of the *itemset* of a transaction and the notion of weaker-than relationship (Definition 2) corresponds to the *containment* of an itemset by a transaction (or by another itemset).

The target of *WL* is to discover all tree expressions that appear in a percentage of the total number of the transaction objects. This percentage is defined by the user and it is called minimum support, *MINSUP*.

WL works in phases. In each phase, it makes a pass over the transaction database. Firstly, it determines the frequent path expressions, that is, frequent 1-tree-expressions. Then it makes several iterations. At the k-th ($k \geq 1$) iteration *WL* constructs a set of candidate $(k+1)$-tree-expressions using the frequent k-sequences and applying some pruning criteria. This set is a superset of the actual frequent $(k+1)$-tree-expressions. Then, it determines the support of the candidates by scanning over the database of transaction objects.

2.2 Problem Definition

We describe below the problem of incremental semistructured schema maintenance in the case that new objects are added into the database.

Definition 3 (Incremental Schema Maintenance). *Consider a collection of transaction objects in an OEM graph and a minimum support threshold MINSUP. Let this collection be named db (regular database). Suppose that we have found the frequent (or large) tree expressions for db, that is, the tree expressions which have support greater than or equal to MINSUP. Suppose that a number of new objects is added into the collection. Let the collection of these objects be named idb (increment database). The incremental schema maintenance problem is to discover all tree expressions which have support in $db \cup idb$ greater than or equal to MINSUP.*

When insertions into a database take place, then some large tree expressions can become small in the updated database (called "losers"), whereas some small tree expressions can become large (called "winners").

ZJZT is based on the idea that instead of ignoring the old large tree expressions, and re-running *WL* on the updated database, *the information from the old large tree expressions can be reused.*

ZJZT works in several passes. In the k-th phase ($k \geq 1$), it scans the increment and recognizes which of the old large k-tree expressions remain large and which become "losers". In the same scan, it discovers the k-tree expressions, which are large in the increment and do not belong to the set of the old large k-tree expressions. These are the candidates to become large k-tree expressions in the updated database. Their support is checked by scanning the regular database. A more detailed description of *ZJZT* can be found in [4, 15].

3 The $\mathcal{D}$*eltaSSD* Algorithm

3.1 The Notion of the *Negative Border*

It is obvious that scanning the regular database many times, as *ZJZT* does, can be time-consuming and in some cases useless, if the large tree expressions of the updated database are merely a subset of the large tree expressions of the regular database. So, [11, 6, 3] considered *using not only the large tree expressions of the regular database, but the candidates that failed to become large in the regular database*, as well. These candidates are called the *negative border* [12].

Below we give the formal definition of the *negative border* of a set of tree expressions.

Definition 4 ([7]). *Let the collection of all possible 1-tree expressions be denoted as R. Given a collection of $S \subseteq \mathcal{P}(R)$ of tree expressions,*[2] *closed with respect to the "weaker than" relation, the negative border Bd^- of S consists of the minimal tree expressions $X \subseteq R$ not in S.*

The collection of all frequent tree expressions is closed with respect to the "weaker than" relationship (Theorem 3.1 [13]). The collection of all candidate tree expressions that were not frequent is the negative border of the collection of the frequent tree expressions.

3.2 The $\mathcal{D}$*eltaSSD* Algorithm

The proposed algorithm utilizes negative borders in order to avoid scanning multiple times the database for the discovery of the new large tree expressions. It differs from [11, 6, 3] in the way it computes the negative border closure. It adopts a hybrid approach between the one layer at a time followed by [6, 3] and the full closure followed by [11]. In summary, after mining the regular database, $\mathcal{D}$*eltaSSD* keeps the support of the large tree expressions along with the support of their negative border. Having this information, it process the increment database in order to discover if there are any tree expressions that moved from the negative border to the set of the new large tree expressions. If there are such tree expressions, then it computes the new negative border. If there are tree expressions with unknown support in the new negative border and

[2] The "power-set" $\mathcal{P}(R)$ includes only "natural" and "near-natural" tree expressions (see [13]).

Table 1. Symbols

Symbol	**Explanation**
db, idb, DB $(= db \cup idb)$	regular, increment and updated database
L^{db}, L^{idb}, L^{DB}	frequent tree expressions of db, idb and DB
N^{db}, N^{idb}, N^{DB}	negative border of db, idb and DB
TE^{db}	$L^{db} \cup N^{db}$
L, N	$L^{DB} \cap (L^{db} \cup N^{db})$, Negative border of L
SupportOf(set, database)	updates the support count of the tree expressions in *set* w.r.t. the *database*
NB(set)	computes the negative border of the *set*
LargeOf(set, database)	returns the tree expressions in *set* which have support count above MINSUP in the *database*

are large in the increment database, then $\mathcal{D}eltaSSD$ scans the regular database, in order to find their support.

The description of the $\mathcal{D}eltaSSD$ requires the notation presented in Table 1.

First Scan of the Increment. Firstly, the support of the tree expressions which belong to L^{db} and N^{db} is updated with respect to the increment database. It is possible that some tree expressions of L^{db} may become small and some others of N^{db} may become large. Let the resulting large tree expressions be denoted as L and the remaining $(L^{db} \cup N^{db}) - L$ tree expressions as *Small*. If no tree expressions that belonged to N^{db} become large, then the algorithm terminates. This is the case that the new results are a subset of the old results and the proposed algorithm is optimal in that it makes only a single scan over the increment database. This is valid due to the following theorem [11]:

Theorem 1. *Let s be a tree-expression such that $s \notin L^{db}$ and $s \in L^{DB}$. Then, there exists a tree-expression t such that t is "weaker than" s, $t \in N^{db}$ and $t \in L^{DB}$. That is, some "component" tree-expression of s moved from N^{db} to L^{DB}.*

Second Scan of the Increment. If some tree expressions do move from N^{db} to L, then we compute the negative border N of L. The negative border is computed using the routine presented in [13], which generates the k-sequences from the $k-1$ sequences. Tree expressions in N with unknown counts are stored in a set N^u. Only the tree expressions in N^u and their extensions may be large. If N^u is empty, then the algorithm terminates. Any element of N^u that is not large in db cannot be large in $db \cup idb$ [4]. Moreover, none of its extensions can be large (antimonotonicity property [2]). So, a second scan over the increment is made in order to find the support counts of N^u.

Third Scan of the Increment. Then, we compute the negative border closure of L and store them in a set C. After removing from C the tree expressions that belong to $L \cup N^u$ for which the support is known, we compute the support counts of the remaining tree expressions in the increment database.

First Scan of the Regular Database. The locally large in *idb* tree expressions, say $ScanDB$, of the closure must be verified in *db*, as well, so a scan over the regular database is performed. In the same scan, we compute the support counts of the negative border of $L \cup ScanDB$, since from this set and from $Small$ we will get the actual negative border of the large tree expressions of $db \cup idb$. After that scan the large tree expressions from $ScanDB$ and the tree expressions in L comprise the new set of the large tree expressions in $db \cup idb$.

Table 2. The $\mathcal{D}eltaSSD$ algorithm

```
$\mathcal{D}eltaSSD\ (db, idb, L^{db}, N^{db})$
//$db$: the regular database, $idb$: the increment database
//$L^{db}, N^{db}$: the large tree expressions of $db$ and their negative border, respectively.

BEGIN
1       $SupportOf(TE^{db}, idb)$ //First scan over the increment.
2       $L = LargeOf(TE^{db}, DB)$
3       $Small = TE^{db} - L$
4       if $(L == L^{db})$ //New results alike the old ones.
            $RETURN(L^{db}, N^{db})$
5       $N = NB(L)$
6       if $(N \subseteq Small)$
            $RETURN(L, N)$
7       $N^u = N - Small$
8       $SupportOf(N^u, idb)$ //Second scan over the increment.
9       $C = LargeOf(N^u)$
10      $Small^{idb} = N^u - C$
11      if $(|C|)$
12          $C = C \cup L$
13          repeat //Compute the negative border closure.
14              $C = C \cup NB(C)$
15              $C = C - (Small \cup Small^{idb})$
16          until ($C$ does not grow)
17          $C = C - (L \cup N^u)$
18          if $(|C|)$ then $SupportOf(C, idb)$ //Third scan over the increment.
19      $ScanDB = LargeOf(C \cup N^u, idb)$
20      $N' = NB(L \cup ScanDB) - Small$
21      $SupportOf(N' \cup ScanDB,\ db)$ //First scan over the regular.
22      $L^{DB} = L \cup LargeOf(ScanDB, DB)$
23      $N^{DB} = NB(L^{DB})$
END
```

Table 3. The regular and increment database

db		*idb*	
1)	a, b	1)	a, b, c
2)	a, b	2)	a, b, c, d
3)	a, c	3)	a, d, g
4)	b, c		
5)	c, d		
6)	a, d, f		
7)	a, d, g		
8)	b, f, g		
9)	a, c, i		

3.3 Mining Results for the Increment

Executing the above algorithm results in computing $L^{db \cup idb}$ and $N^{db \cup idb}$ and their support. We also need the complete mining results for the increment database *idb*, that is, L^{idb} and N^{idb} and their support. We describe how this can be achieved without additional cost, but during the three passes over the increment database. After the first pass, we know the support of the tree-expressions belonging to $L^{db \cup idb}$ and $N^{db \cup idb}$ in the increment itself. From these, we identify the frequent ones and compute their negative border. If some tree-expressions belonging to the negative border are not in $L^{db \cup idb} \cup N^{db \cup idb}$ we compute their support during the second pass over the increment. Then the negative border closure of the resulting (frequent in *idb*) tree-expressions is computed. If there are new tree-expressions, which belong to the closure and whose support in *idb* is not known, then their support is computed in the third pass over the increment.

Example 1. We give a short example of the execution of the $\mathcal{D}eltaSSD$. For the sake of simplicity, we present the example using flat itemsets and the set containment relationship instead of tree expressions and the weaker than relationship. Suppose that all the possible "items" are the following $R = \{a, b, c, d, f, g, i\}$. Let the regular database be comprised by nine transactions and the increment database be comprised by three transactions. The databases are presented in Table 3. Let the support threshold be 33.3%. Thus, an item(set) is large in the regular database, if it appears in at least three transactions out of the nine.

We can confirm that the frequent "items" in the regular database *db* are the following $L^{db} = \{a, b, c, d\}$. Thus, their negative border, which is comprised by the itemsets that failed to become large, is $N^{db} = \{f, g, i, ab, ac, ad, bc, bd, cd\}$. The steps of the $\mathcal{D}eltaSSD$ proceed as shown in Table 4.

Table 4. An example execution of the $\mathcal{D}eltaSSD$ algorithm

```
$\mathcal{D}eltaSSD\ (db, idb, L^{db}, N^{db})$
Input: $L^{db} = \{a, b, c, d\}$ and $N^{db} = \{f, g, i, ab, ac, ad, bc, bd, cd\}$

BEGIN
1       count support of $(L^{db} \cup N^{db} = \{a, b, c, d, f, g, i, ab, ac, ad, bc, bd, cd\})$ in $idb$
2       $L =$ LargeOf$(L^{db} \cup N^{db})$ in $DB \implies L = \{a, b, c, d, ab, ac, ad\}$
3       $Small = (L^{db} \cup N^{db}) - L \implies Small = \{f, g, i, bc, bd, cd\}$
4       $L \neq L^{db}$
5       $N =$ NegativeBorderOf$(L) \implies N = \{f, g, i, bc, bd, cd, abc, abd, acd\}$
6       $N \subsetneq Small$
7       $N^u = N - Small \implies N^u = \{abc, abd, acd\}$
8       count support of $(N^u = \{abc, abd, acd\})$ in $idb$
9       $C =$ LargeOf$(N^u)$ in $idb \implies C = \{abc, abd, acd\}$
10      $Small^{idb} = N^u - C \implies Small^{idb} = \emptyset$
11      $C \neq \emptyset$ thus
12          $C = C \cup L \implies C = \{a, b, c, d, ab, ac, ad, abc, abd, acd\}$
13          repeat //Compute the negative border closure.
14              $C = C \cup$ NegativeBorderOf$(C)$
15              $C = C - (Small \cup Small^{idb})$
16          until ($C$ does not grow)
            Finally: $C = \{a, b, c, d, ab, ac, ad, abc, abd, acd, abcd\}$
17          $C = C - (L \cup N^u) \implies C = \{abcd\}$
18          $C \neq \emptyset$ thus
                count support of $(C = \{abcd\})$ in $idb$
19      $ScanDB =$ LargeOf $(C \cup N^u)$ in $idb \implies ScanDB = \{abc, abd, acd, abcd\}$
20      $N' =$ NegativeBorderOf$(L \cup ScanDB) - Small \implies N' = \emptyset$
21      count support of $(N' \cup ScanDB = \{abc, abd, acd, abcd\})$ in $db$
22      $L^{DB} = L \cup$ LargeOf$(ScanDB)$ in $DB \implies L^{DB} = \{a, b, c, d, ab, ac, ad\}$
23      $N^{DB} =$ NegativeBorderOf$(L^{DB}) \implies N^{DB} = \{f, g, i, bc, bd, cd\}$
END
```

4 Experiments

We conducted experiments in order to evaluate the efficiency of the proposed approach $\mathcal{D}eltaSSD$ with respect to *ZJZT*, and also with respect to *WL*, that is, re-running Wang's algorithm on the whole updated database.

4.1 Generation of Synthetic Workloads

We generated acyclic transaction objects, whose nodes have list semantics. Each workload is a set of transaction objects. The method used to generate synthetic transaction objects is based on [2, 13] with some modifications noted below.

Each transaction object is a hierarchy of objects. *Atomic* objects, located at level 0, are the objects having no descendants. The *height* (or *level*) of an object is the length of the longest path from that object to a descendant atomic object. All transaction objects are at the same level m, which is the *maximal nesting level*. Each object is recognized by an identifier. The number of identifiers for objects of level i is N_i. Each object is assigned one (*incoming*) label, which represents a "role" for that object. Any object i that has as subobject an object j, will be connected to j through an edge labelled with the label of object j. All transaction objects have the same incoming label.

Objects belonging to the same level are assigned labels drawn from a set, different for each level i, with cardinality equal to L_i. We treat each object serially and draw a label using a self-similar distribution. This power law provides the means to select some labels ("roles"), more frequently than others. A parameter of this distribution determines the skewness of the distribution ranging from uniform to highly skewed. In our experiments, we set this parameter equal to 0.36 to account for a small bias.

The number of the subobject references of an object at level i is uniformly distributed with mean equal to T_i. The selection of subobjects models the fact that some structures appear in common in many objects. To achieve this, we used the notion of *potentially large sets* [2]. Thus, subobject references for an object at level i are not completely random, but instead are drawn from a pool of *potentially large sets*. If the *maximum nesting level* equals m, then this pool is comprised by $m-1$ portions, namely $\Gamma_1, \Gamma_2, \ldots, \Gamma_{m-1}$. Each Γ_i is comprised by sets of level-i identifiers. The average size of such a set is I_i. More details regarding these sets can be found in [2]. The construction of the objects is a bottom-up process. Starting from level-2, we must construct N_2 objects. For each object, we first choose the number of its subobject references (its size) and then pick several potential large sets from Γ_1 until its size is reached. Recursively, we construct the level-3 objects and so on. For any object belonging to any level, say level $i > 2$, we obligatorily choose one potentially large set from Γ_{i-1} and then we choose the rest of the potentially large sets equiprobably from all $\Gamma_j, 1 \leq j < i$.

Thus, a generated data set in which transaction objects are at level m will be represented as: $\langle L_1, N_1, I_1, P_1\rangle$, $\langle L_2, N_2, T_2, I_2, P_2\rangle, \ldots, \langle N_m, T_m\rangle$. [3] Table 5 presents the notation for the generation of synthetic data.

The way we create the increment is a straightforward extension of the technique used to synthesize the database. In order to do a comparison on a database of size $|db|$ with an increment of size $|idb|$, we first generate a database of size $|db + idb|$ and then the first $|db|$ transactions are stored in the regular database and the rest $|idb|$ are stored in the increment database. This method will produce data that are identically distributed in both db and idb and was followed in [4, 9], as well.

[3] Remember that $T_1 = 0$, $L_m = 1$ and that there is no Γ_m.

Table 5. Notation used for the generation of synthetic data

Symbol	Explanation
L_i	Number of level-i labels
N_i	Number of level-i object identifiers
T_i	Average size of $val(o)$ for level-i identifiers o
I_i	Average size of potentially large sets in Γ_i
P_i	Number of potentially large sets in Γ_i
m	maximal nesting level

4.2 Experimental Results

For all the experiments reported below, we used the following dataset comprised by 30000 transaction objects: $\langle 100, 5000, 3, 100\rangle$, $\langle 500, 500, 8, 3, 400\rangle$, $\langle 3000, 8\rangle$.

We used as performance measure the number of passes over the whole database $db \cup idb$. For an algorithm, which makes α passes over the regular database and β passes over the increment database, the number of passes is estimated as $\frac{\alpha * |db| + \beta * |idb|}{|db| + |idb|}$, where $|db|$ and $|idb|$ is the number of transactions of the regular and the increment database, respectively.

Varying the Support Threshold. Our first experiment aimed at comparing the performance of the algorithms for various support thresholds and the results are depicted in Figure 3. We observe that $\mathcal{D}eltaSSD$ performs much better than the rest of the algorithms and makes on the average (almost) only one pass over the whole database. For higher support thresholds, it performs even better, because it does not scan the regular database, but scans once or twice the increment database. *ZJZT* and *WL* perform 4 full scans, because the number of passes depends on the number of leaves of the tree expression with the largest number of leaves.

Varying the Increment Size. Our second experiment aimed at evaluating the performance of the algorithms for various increment sized. The results are depicted in Figure 4. We notice that *ZJZT* and *WL* make the same constant number of scans for the reason explained earlier, whereas the number of scans performed by $\mathcal{D}eltaSSD$ increases slightly with the increment size, as a function of the increment size and the number of candidate tree expressions that move from the negative border to the set of the large tree expressions, imposing a scan over the regular database.

Comparison of *ZJZT* and *WL*. Since both *ZJZT* and *WL* perform the same number of scans over the database, we further investigated their performance by comparing the number of node comparisons they make during the tree matching

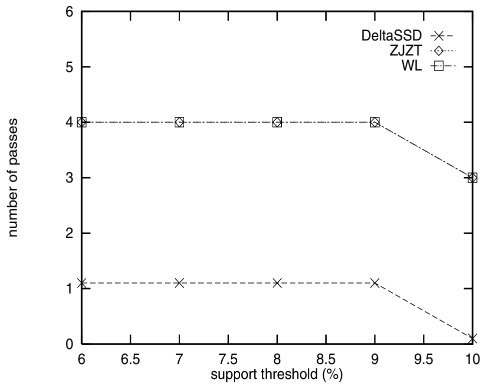

Fig. 3. Database passes with varying support threshold (10% increment)

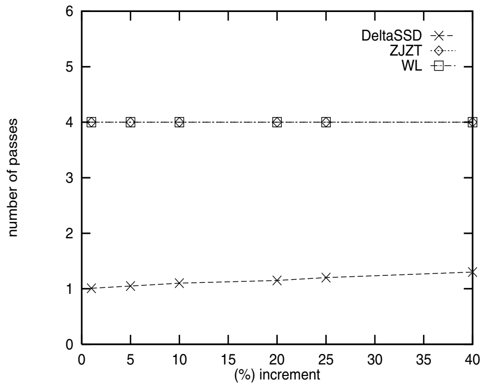

Fig. 4. Database passes with varying increment size (8% support)

operation (involved in the computation of the *weaker than* relationship). This measure is independent on any particular implementation and reflects the CPU time cost of the algorithms. The results are depicted in Figure 5.[4]

We can observe that with increasing support the performance gap between *ZJZT* and *WL* broadens, because higher support means fewer candidate tree expressions and even fewer large tree expressions and thus smaller number of tree matchings.

Increment sizes impacts also the performance of the algorithms. Larger increment means that more new candidates arise in the increment and thus larger number of tree matchings in order to count their support both in the increment and in the regular database. Thus, the number of comparisons made by

[4] The right graph presents the ratio of the number of node comparisons made by *ZJZT* to the number of comparisons made by *WL*.

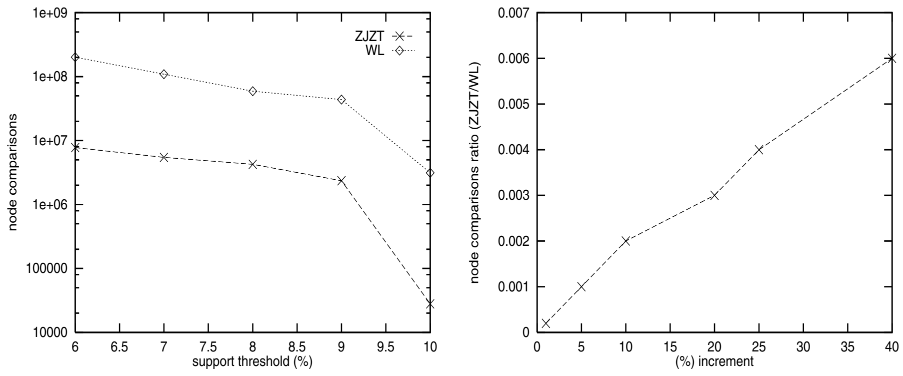

Fig. 5. *Left* Varying support (10% increment), *Right* Varying increment size (8% support)

ZJZT increases with respect to *WL* (larger ratio, as depicted in the right part of Figure 5).

The results clearly indicate the superiority of the $\mathcal{D}eltaSSD$ algorithm, which performs the smaller number of scans over the database. The *ZJZT* algorithm performs the same number of scans with *WL*. This is expected, since the number of scans depends on the size (in terms of the number of path expressions it contains) of the largest *tree-expression*. But, *ZJZT* is much better than *WL* for low and large support thresholds and small increment sizes, whereas their performance gap narrows for moderate support thresholds and large increment sizes.

5 Conclusions

As the amount of on-line semistructured data grows very fast, arises the need to efficiently maintain their "schema". We have considered the problem of incrementally mining structural associations from semistructured data. We exploited the previous mining results, that is, knowledge of the *tree-expressions* that were frequent in the previous database along with their negative border, in order to efficiently identify the frequent *tree-expressions* in the updated database.

We presented the $\mathcal{D}eltaSSD$ algorithm, which guarantees efficiency by ensuring that at most three passes over the increment database and one pass over the original database will be conducted for any data set. Moreover, in the cases where the new "schema" is a subset of the old, $\mathcal{D}eltaSSD$ is optimal in the sense that it will make only one scan over the increment database.

Using synthetic data, we conducted experiments in order to assess its performance and compared it with the *WL* and *ZJZT* algorithms. Our experiments showed that for a variety of increment sizes and support thresholds,

$\mathcal{D}eltaSSD$ performs much better than its competitors making (almost) only one scan over the whole database.

In summary, $\mathcal{D}eltaSSD$ is a practical, robust and efficient algorithm for the incremental maintenance of structural associations of semistructured data.

References

[1] S. Abiteboul. Querying semistructured data. In *Proceedings 6th ICDT Conference*, pages 1–18, 1997. 118, 120

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings 20th VLDB Conference*, pages 487–499, 1994. 119, 122, 124, 127, 128

[3] Y. Aumann, R. Feldman, O. Liphstat, and H. Mannila. Borders: an efficient algorithm for association generation in dynamic databases. *Journal of Intelligent Information Systems*, 12(1):61–73, 1999. 120, 123

[4] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings 12th IEEE ICDE Conference*, pages 106–114, 1996. 119, 123, 124, 128

[5] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings ACM SIGMOD Conference*, pages 431–442, 1999. 120

[6] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *Proceedings ACM DMKD Workshop*, 1997. 120, 123

[7] H Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997. 120, 123

[8] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings ACM SIGMOD Conference*, pages 295–306, 1998. 119

[9] V. Pudi and J. Haritsa. Quantifying the utility of the past in mining large databases. *Information Systems*, 25(5):323–343, 2000. 120, 128

[10] A. Rajaraman and J. Ullman. Querying Websites using compact skeletons. In *Proceedings 20th ACM PODS Symposium*, 2001. 119

[11] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings KDD Conference*, pages 263–266, 1997. 120, 123, 124

[12] H. Toivonen. Sampling large databases for association rules. In *Proceedings 22nd VLDB Conference*, pages 134–145, 1996. 123

[13] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):353–371, 2000. 119, 120, 121, 123, 124, 127

[14] Q. Y. Wang, J. X. Yu, and K.-F. Wong. Approximate graph schema extraction for semi-structured data. In *Proceedings 7th EDBT Conference*, pages 302–316, 2000. 119

[15] A. Zhou, Jinwen, S. Zhou, and Z. Tian. Incremental mining of schema for semistructured data. In *Proceedings Pasific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 159–168, 1999. 119, 123