University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) Department of Computer & Information Science

9-1-2000

View Maintenance for Hierarchical Semistructured Data

Hartmut Liefke
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Postprint version. Published in *Lecture Notes in Computer Science*, Volume 1874, Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery 2000 (DaWaK 2000), pages 114-123.
Publisher URL: http://springerlink.metapress.com/link.asp?id=1g7xvlhfk2phxl5a

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/125
For more information, please contact repository@pobox.upenn.edu.

View Maintenance for Hierarchical Semistructured Data *

Hartmut Liefke and **Susan B. Davidson**
Dept. of Computer and Information Science
University of Pennsylvania
liefke@seas.upenn.edu and susan@cis.upenn.edu

Abstract. Over the last few years, efficient access to heterogeneous data sources has become tremendously important. One common technique for increasing efficiency is to maintain locally stored views in data warehouses, which must be kept current with respect to the changes in the underlying data sources. While this problem has been extensively studied in the context of select-project-join (SPJ) views and relational warehouses, many of the data sources accessible today over the Web are highly irregular. Views over this irregular data often perform complex restructuring and regrouping far beyond traditional SPJ views.
This paper describes WHAX (Warehouse Architecture for XML), an architecture for defining and maintaining views over hierarchical semistructured data and relational data sources with key constraints. The WHAX model is a variant of the deterministic model of [8], but is more reminiscent of XML. The view definition language is a variation of XML-QL that has been adapted to the WHAX model, and supports selections, joins, and important restructuring operations such as regrouping, flattening, and aggregation. The incremental maintenance technique is based on the notion of *multi-linearity* and generalizes several well-known techniques from the relational case.

1 Introduction

XML has become an important standard for the representation and exchange of data over the Internet. As an instance of semi-structured data [2], which was initially proposed as a methodology in the Tsimmis project [10], it can also be used for data integration [12]. That is, data sources are represented in XML; transformations and integration are then expressed in one of several XML query languages that have been proposed [17, 13, 15] to create an XML view [1]. The view can then be stored (or materialized) by mapping it into a conventional (relational) DBMS as suggested in [18, 19].

For example, this approach is currently being explored within the biomedical community in data integration projects at GeneLogic [11] as well as in the EpoDB [28] and GUSS projects in the Center for Bioinformatics at the University of Pennsylvania (see www.pcbi.upenn.edu).

* This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF DBI99-75206, NSF IIS98-17444, ARO DAAG55-98-1-0331, and a grant from SmithKline Beecham.

Once the view has been materialized, it must be maintained as updates to the underlying data sources are made. This problem has been extensively studied for select-project-join (SPJ) views in the relational model (see [23] for a survey) and has also been investigated for object-oriented databases [25, 20]. SPJ views have an important property: They are distributive with respect to relational union. For example, $R_1 \bowtie (R_2 \cup \triangle R_2) = (R_1 \bowtie R_2) \cup (R_1 \bowtie \triangle R_2)$ holds for any relations R_1 and R_2 and a set of new tuples $\triangle R_2$. To compute the new view $R_1 \bowtie (R_2 \cup \triangle R_2)$, only the query $R_1 \bowtie \triangle R_2$ must be evaluated and the result added to the existing view $(R_1 \bowtie R_2)$. Since $\triangle R_2$ is usually much smaller than R_2, this is more efficient than reevaluating the view. More general, SPJ views are functions $f(R_1, ..., R_n)$ of (not necessarily distinct) base relations R_1, ..., R_n that are *multi-linear* with respect to union:[1]

$$f(R_1, ..., R_i \cup \triangle R_i, ..., R_n) \;=\; f(R_1, ..., R_i, ..., R_n) \;\cup\; f(R_1, ..., \triangle R_i, ..., R_n)$$

Multi-linearity is a powerful concept. It allows efficient view maintenance for bulk updates and can be extended to handle deletions. It is also fundamental for other optimization techniques, such as parallel query evaluation and pipelining.

Unfortunately, it is difficult to adopt this technique for semistructured data and XML. First, the law requires that updates are represented as data values themselves, and second, some union operation $\cup$ must be defined.

Existing techniques for view maintenance in the semistructured data model [3, 31] only allow single atomic graph updates such as edge deletion/insertion based on the node's object identities. We will show in Sec. 2 that updates with opaque OIDs carry little information about how the view is affected. Hence, view maintenance is complicated and complex auxiliary data structures must be maintained.

The WHAX *Approach.* The Warehouse Architecture for XML (WHAX) is a new architecture that combines the power and simplicity of multi-linearity with the flexibility of the semistructured data model. The WHAX model is a hierarchical, semistructured data model very much like the null-terminated deterministic model introduced in [8], and is also closely related to LDAP [29, 24]. WHAX is based on the concept of *local keys*: Each outgoing edge of the same parent node is identified by some locally unique key. This ensures that each node in the tree is identified by a unique path from the root, which provides a new notion of object identity. One consequence is that WHAX-updates can be represented as WHAX-trees themselves. WHAX data can easily be produced from XML data by the specification of keys, and can be straightforwardly produced from relational data using key information.

Contributions. In this paper, we make the following contributions:

- The WHAX data model provides a natural embedding for relational and hierarchical semistructured data sources such as XML. As in the deterministic model [8], updates are represented as WHAX-values themselves, and there is one fundamental update operation called deep-union.

[1] We slightly misuse the notion of multi-linearity here: To be multi-linear, $\cup$ must be a group operation with an inverse function similar to the definition in [20]. We address this issue in Sec. 6.

```
<Conf name="STACS" year="1996">
   <Publ> <Title> Views </Title>
          <Author> Tim </Author>
          <Author> Peter </Author>
          <Pages> <From> 117 </From>
                  <To> 127 </To>
          </Pages>
   </Publ>
   <Publ> <Title> Types </Title>
          <Author> Tim </Author>
          <Pages> <From> 134 </From>
                  <To> 146 </To>
          </Pages>
  </Publ>
</Conf>
```

```
<Person>
   <Name> Tim </Name>
   <Age> 35 </Age>
</Person>

<Person>
    <Name> Peter </Name>
    <Age> 45 </Age>
</Person>
```

Fig. 1. A Publication Database in XML

- The query language WHAX-QL, based on XML-QL [17], generalizes relational SPJ queries and additionally allows powerful restructuring through regrouping and aggregations. We present a restriction of WHAX-QL that is provably multilinear, and hence allows efficient incremental view maintenance.
- For deletion updates, we develop an extension of the counting technique used for SPJ views in the relational model. The technique can also be used for view definitions involving aggregation.

Limitations. In this paper, we do not consider *ordered* data structures as in XML. Since positions of elements can change dynamically, view maintenance of ordered structures is more difficult. Initial ideas can be found in [26] and are briefly described in Sec. 8. Furthermore, we do not investigate properties of the query language, such as expressiveness under multi-linearity. However, we believe that the techniques proposed in this paper are fundamental and can be extended to other, more powerful languages as well.

The rest of the paper is organized as follows. Sec. 2 presents a motivating XML example, and shows how the WHAX approach differs from existing view maintenance techniques for semistructured data. The WHAX data model is explained in Sec. 3, followed by the view definition language in Sec. 4. We describe the multi-linearity property for WHAX-views in Sec. 5 and extend the incremental view maintenance technique to deletions in Sec. 6. Sec. 7 describes how aggregate queries can be efficiently maintained in WHAX. We conclude in Sec. 8 with a brief discussion of future work.

2 Motivating Example

Consider the XML example shown in Fig. 1 with information about conference publications and authors.[2] Each publication has a title and a list of author

[2] Any resemblence with real people is unintentional!

names. Publications are grouped within the conference in which they appear and each author name refers to a person.

There are two ways to identify (and update) an object in XML[3]. First, one can use *positions* to describe a path within the document, e.g. the third paper in the second conference. However, positions can change during updates (e.g. the insertion of a new first author), and positions in the view can be different from the position in the source (e.g. "select all authors with Age>30").

The second approach is to adopt the (unordered) semistructured data model. Nodes are identified by object identities and three updates are considered [3, 31]: *edge insertion*, *edge deletion*, and *value modification*. Updates based on object identities, however, make it difficult to reason about what parts of the view have been affected.

To illustrate, consider the following simple view: store all authors who published in STACS'96. Furthermore, consider the insertion of a new author into some publication with OID o. Since it is not directly observable whether o belongs to STACS'96 or not, it is not clear whether the author should be inserted into the view. The algorithms described in [3, 31] therefore need auxiliary data structures to represent the relationship between objects in the view and those in the source database. These data structures can be large and expensive to maintain.

The approaches in [3, 31] have two other serious drawbacks: First, the algorithms are restricted to *atomic* updates, i.e. each single atomic update causes queries to the source database to update the view. Second, objects in the views must always correspond to exactly one object in the source. Hence, important regrouping or aggregate operations cannot be performed.

Intuitively, the problem can be simplified considerably by providing and using existing key information. In the WHAX model, each node is identified by the path from the root to the node. The keys along the path capture much more information than opaque object identities. For example, Fig. 2 shows the WHAX representation of the XML database in Fig. 1. Each conference is identified by its name and the year, a publication within a conference is identified by its title, and an author is identified by its name.

The previous update can now easily be propagated into the view, since the key information (STACS'96) is part of the identity of the inserted author. We elaborate on the model in the next section.

3 The WHAX Data Model

The WHAX data model is an unordered edge-labeled tree. The tree is *deterministic* in that the outgoing edges of a node have different *local identifiers*. More formally, let B be the set of all base values (strings, integers, ...) and V be the set of WHAX-values. Local identifiers are pairs $l(k)$ with *label* $l \in B$ and *key* $k \in V$. The set of all WHAX-values is defined recursively as the (minimal) set of finite partial functions from local identifiers to trees: $V = (B \times V) \rightsquigarrow V$.

[3] As with relational view maintenance, we do not consider bulk update expressions (e.g. "Increase the age of all persons by 1"). They must first be evaluated on the database to generate updates on *specific* objects.

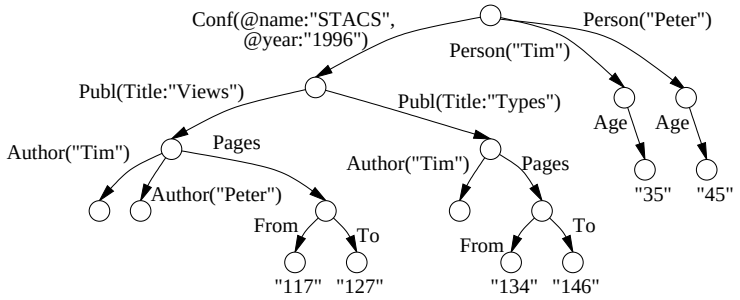

Fig. 2. Sample WHAX Tree

Values are constructed using $\{l_1(k_1) : \{...\}, ..., l_n(k_n) : \{...\}\}$. If k_i in $l_i(k_i)$ is the empty partial function $\{\}$, then we write l_i. If k_i is a tree construct $\{...\}$, then we write $l_i(...)$ instead of $l_i(\{...\})$. For example, the value $\{\mathsf{Conf}(@\mathsf{name} : \{\mathsf{STACS} : \{\}\}, @\mathsf{year} : \{1996 : \{\}\}) : ...\}$ represents conference STACS'96. Singleton trees of the form $\{\mathsf{str} : \{\}\}$ occur frequently and are abbreviated as double-quoted literals "str". Hence, the value above is equivalent to $\{\mathsf{Conf}(@\mathsf{name} : \text{“}\mathsf{STACS}'', @\mathsf{year} : \text{“}1996'') : ...\}$.

Fig. 2 shows the WHAX representation of the XML example in Fig. 1. We use the XPath [14] convention and preceed attribute labels with '@'. The Age label forms an identifier with an empty key $\{\}$; this represents the constraint that there can be only one age per Person. Recall that "str" represents singleton function $\{\mathsf{str} : \{\}\}$. E.g. local identifier Person("Tim") is the same as $\mathsf{Person}(\{\mathsf{Tim} : \{\}\})$ and leaf nodes marked with some string "str" have, in fact, one outgoing edge with label str and the empty value $\{\}$.

Deep Union. The fundamental operation on WHAX-trees is *deep union* [8]. The deep union of two WHAX-trees v_1 and v_2 matches the common identifiers of v_1 and v_2 and recursively performs deep union on their respective subvalues. The edges that only occur in v_1 (or v_2), but not in v_2 (v_1, respectively) are copied into the resulting tree:

$$\begin{aligned} v_1 \uplus v_2 ::= & \{l(k) : \ s_1 \uplus s_2 \mid l(k) : s_1 \in v_1, l(k) : s_2 \in v_2\} \cup \\ & \{l(k) : s \mid l(k) : s \in v_1, l(k) \notin \mathsf{dom}(v_2)\} \cup \\ & \{l(k) : s \mid l(k) : s \in v_2, l(k) \notin \mathsf{dom}(v_1)\} \end{aligned}$$

where $\mathsf{dom}(v)$ is the set of local identifiers in v: $\mathsf{dom}(v) \ := \ \{l(k) \mid l(k) : s \in v\}$. Similar to union in relational databases, deep union is the core mechanism for *inserting* data into a database. For example, Fig. 3 shows how a shoesize and address value for Tim can be added to a database with Tim and Peter.

We will describe in Sec. 5 how WHAX-QL queries are multi-linear with respect to deep union and we show in Sec. 6 how how deep union can be extended to handle deletions.

XML $\Rightarrow$ WHAX. The conversion from (unordered) XML data into a WHAX tree is not difficult. Given the information about keys, labels in the XML tree are

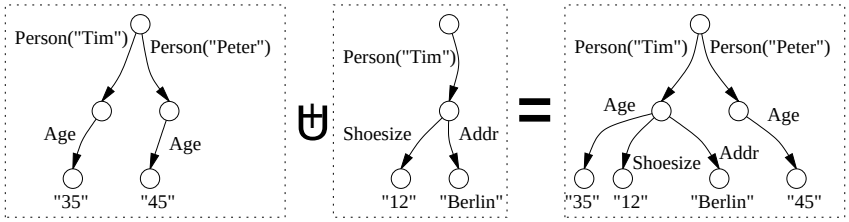

Fig. 3. Deep Union

annotated with the key values and the key is "pulled" out from the XML tree. Details about key specifications and the transformation can be found in [26].

Relational Databases in WHAX. There is a natural translation from relational databases into WHAX: Each tuple is represented by an outgoing edge from the root. The relation name R and the key k of the tuple form the local identifier $\mathsf{R}(k)$. All non-key attributes form a subtree under $\mathsf{R}(k)$. Fig. 4 shows an example.

R1	A	B	C	D
	a1	b1	c1	d1
	a2	b2	c2	d2

The primary key of R1 is {A,B}

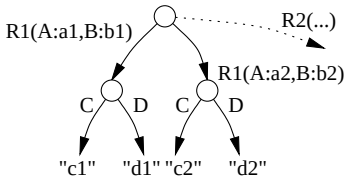

Fig. 4. Translation of a relational database to some WHAX-tree

4 Defining Views in WHAX

Over the past few years, several query languages for semi-structured and XML data have been proposed [17, 27, 13, 4, 7]; as of yet, however, there is no standard. In this paper, we consider a language, called WHAX-QL, which is based on XML-QL [17]. WHAX-QL differs from XML-QL in that local identifiers (i.e. labels and keys) can be matched against patterns. XML-QL bindings such as <tag> $\$x$ </> **in** $\$db$ are modified to <tag($KPat$)> $\$x$ </> **in** $\$db$, where key pattern $KPat$ contains variable bindings, constant values, or combinations of them.

We start by illustrating WHAX-QL through some examples.

Example View V_1: Select the name and the age of all authors older than 36:

```
V1($db) =  where       <Person($n).Age> $a </> in $db,
                       $a > 36
           construct   <MyPerson($n).Age> $a </>
```

The path expression Person($\$n$).Age identifies the age of each person in the database $\$db$ and binds the person's name to $\$n$. The age of the person is bound

to variable *$a*. Recall that the age is represented as a single outgoing edge from the Age node. To satisfy $\$a > 36$, the value of $\$a$ must be a label l or of the form $\{l : \{\}\}$ where l is an integer with $l > 36$.

Note that only persons with an Age element appear in the output, which is different from the following view V_1':

```
V1'($db) =  where      <Person(Name:$n)> $p </> in $db
            construct  <MyPerson(Name:$n)>
                          where     <Age> $a </> in $p,
                                    $a > 36
                          construct <Age> $a </>
                       </>
```

Example View V_2: For each author, return the title, conference name, and page numbers for each publication:

```
V2($db) =
  where     <Conf(@name:$n,@year:$y).Publ(Title:$t)>
                <Author($a)> </>
                <Pages> $p </>
            </> in $db
  construct <Author(Name:$a).Publ(Title:$t,Conf:$n,Year:$y).Pages> $p </>
```

Variable *$a* binds to any author name and *$p* is bound to the pages of the publication identified by *$n*, *$y*, and *$t*. The view is a tree with authors at the root and their publications at the leaves. This illustrates the regrouping power of WHAX-QL.

It is always possible to "unnest" nested variable bindings. For example, View V_2 could be equivalently written as:

```
V'2($db) =
  where     <Conf(@name:$n,@year:$y).Publ(Title:$t).Author($a)> </> in $db,
            <Conf(@name:$n,@year:$y).Publ(Title:$t).Pages> $p </> in $db
  construct <Author(Name:$a).Publ(Title:$t,Conf:$n,Year:$y).Pages> $p </>
```

As in XML-QL, multiple occurrences of the same variable require the corresponding values to be equal, which is an intuitive way to represent joins. In the example above, the uniqueness of paths in WHAX ensures that author *$a* and pages *$p* belong to the *same* publication.

Example View V_3: For each person, return their age and all STACS publications. Furthermore, group all publication by their year:

```
V3($db) =
where     <Person(Name:$n).Age> $a </> in $db,
          <Conf(@name:"STACS",@year:$y).Publ(Title:$t).Author($n)></> in $db
construct <Author(Name:$n).Age> $a </>,
          <Author(Name:$n).STACS(Year:$y).Title($t)> </>
```

This query performs a join between persons and authors over variable $\$n$. Fig. 5 shows the regrouping effect of this query.

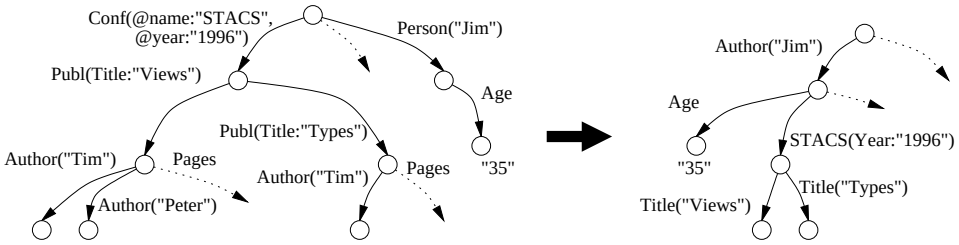

Fig. 5. Regrouping result for view V_3

$$
\begin{array}{lll}
\mathsf{e} & ::= & \text{"}str\text{"} \mid \$x \mid \mathsf{e}_1 \ \mathsf{op}\ \mathsf{e}_2 \mid \mathsf{e}_1 \uplus \mathsf{e}_2 \mid Q \mid (LPat(\mathsf{e}_1) : \mathsf{e}'_1, ..., LPat_n(\mathsf{e}_n) : \mathsf{e}'_n) \\
Q & ::= & \textbf{where} \quad <PPat_1> \ \$x_1 \ </> \ \textbf{in} \ \$d_1, \\
 & & \quad ... \\
 & & \quad <PPat_m> \ \$x_m \ </> \ \textbf{in} \ \$d_m, \\
 & & \quad cond_1, ..., cond_n \\
 & & \textbf{construct} \ <PExpr_1> \ \mathsf{e}_1 \ </>, \ ..., \ <PExpr_p> \ \mathsf{e}_p \ </>
\end{array}
$$

$$
\begin{array}{lll}
PPat & ::= & LPat_1(KPat_1). \ ... \ .LPat_n(KPat_n) \\
LPat & ::= & l \mid \$x \\
KPat & ::= & \$x \mid (l_1(\mathsf{v}_1) : KPat_1, ..., l_n(\mathsf{v}_n) : KPat_n) \\
PExpr & ::= & LPat_1(\mathsf{e}_1). \ ... \ .LPat_n(\mathsf{e}_n)
\end{array}
$$

Fig. 6. Syntax of WHAX-QL

4.1 The Syntax of WHAX-QL

Fig. 6 shows the syntax of WHAX-QL. A WHAX expression e can either be a string constant *str*, a variable, an arithmetic or comparison operation e_1 op e_2, the deep union $e_1 \uplus e_2$, a where-construct-expression Q, or a value constructor $(LPat(\mathsf{e}_1) : \mathsf{e}'_1, ..., LPat_n(\mathsf{e}_n) : \mathsf{e}'_n)$.

The where-clause of Q describes the *variable bindings*. For each *valuation* of the variables in the where-clause, the construct-clause is evaluated and the results are deep-unioned together.

The identifiers $\$d_1$, ..., $\$d_n$ in the where-clause denote WHAX data sources and are called *base variables*. A *path pattern* $PPat_i$ matches a path in $\$d_i$ against a given sequence of *label patterns* $LPat$ and *key patterns* $KPat$, separated by dots: $LPat_1(KPat_1). \ ... \ .LPat_n(KPat_n)$. The value at the end of the path is bound to variable $\$x_i$. As in XML-QL, variable $\$x_i$ can be omitted if it is not used anywhere.

A label pattern $LPat$ is either a constant label l or a label variable $\$x$, and a key pattern $KPat$ is either a variable $\$x$ or a complex pattern $(l_1(\mathsf{v}_1) : KPat_1, ..., l_n(\mathsf{v}_n) : KPat_n)$. It matches a WHAX-tree if the tree has exactly n elements with distinct local identifiers $l_1(\mathsf{v}_1)$, ..., $l_n(\mathsf{v}_n)$ and the element values match patterns $KPat_1$, ..., $KPat_n$, respectively. If $KPat$ (or v) in $LPat(KPat)$ ($l(\mathsf{v})$, respectively) is the empty value $\{\}$, then we write $LPat$ (l, respectively).

A path expression $PExpr_i$ is a sequence of label and key constructs. The leaf value of the path is determined by WHAX-QL expression e_i. The values of all path expressions under all valuations are (deep-)unioned to form the output.

Variables in WHAX-*QL.* We distinguish several types of variables by their first occurrence within a WHAX-expression. Parameters to the query (such as $\$db$) are called *parameter variables.* Variables in path patterns $PPat_i$ are bound to labels or key values and are called *label variables* and *key variables*, respectively. Variables $\$x_i$ at the leaves of path bindings are called *value variables.* Variable $\$db$ in $<PPat_i>$ $\$x_i$ $</>$ **in** $\$db$ must be bound in an outer scope, either as a parameter, value, or key variable.

Syntactic Simplifications We do not consider nested binding patterns in this paper. Views V_2 and V_2' in Sec. 4 illustrate how such nested bindings can be eliminated. Similarly, we do not consider the case where some value variable $\$x_i$ from $<PPat_i>$ $\$x_i$ $</>$ **in** $\$db_i$ is the base variable $\$db_j$ of some other pattern in the same where-clause: $<PPat_j>$ $\$x_j$ $</>$ **in** $\$x_i$. The second pattern can be replaced by pattern $<PPat_i.PPat_j>$ $\$x_j$ $</>$ **in** $\$db_i$.

XML-QL vs. WHAX-*QL.* Since WHAX-QL is based on a deterministic tree model, WHAX-QL does not require Skolem functions, which are used in XML-QL to generate OIDs. Lastly, although we did not describe regular path expressions, they can easily be introduced with a pattern $RegExpr[\$p]$ where $RegExpr$ denotes a regular expression over local identifiers and $\$p$ binds to the entire path matching $RegExpr$. All view maintenance results in this paper will still hold.

5 View Maintenance through Multi-Linearity

As mentioned earlier, the multi-linearity law is the foundation for efficient incremental view maintenance of bulk updates. For example, a relational SPJ view $V(R_1, R_2)$ is distributive in parameters R_1 and R_2 with respect to union: $V(R_1 \cup \triangle R_1, R_2) = V(R_1, R_2) \cup V(\triangle R_1, R_2)$ and $V(R_1, R_2 \cup \triangle R_2) = V(R_1, R_2) \cup V(R_1, \triangle R_2)$. The updated view $V(R_1 \cup \triangle R_1, R_2)$ is therefore easily computed by inserting the result of query $V(\triangle R_1, R_2)$ to the view $V(R_1, R_2)$.

The same multi-linearity law should hold for WHAX queries under deep union $\uplus$. Unfortunately, WHAX-QL queries are not necessarily multi-linear in their given form. We identify two crucial properties to make WHAX-QL queries multi-linear: First, the *key variable constraint* forbids the use of parameter and value variables as key and operand variables. Second, the *base variable constraint* forbids the multiple use of base variables. While the first constraint is a restriction of the language, the second condition can always be fulfilled by query rewriting. Both properties are described next.

Key Variable Constraint. Intuitively, a view is not multi-linear if it is not monotone, i.e. if an insertion at the source can cause a deletion in the view. Recall example view V_1 from Sec. 4, which accesses value variable $\$a$ in condition $\$a > 36$. The condition is only true if $\$a$ is a singleton tree with an integer label > 36. Consider the (rather unusual) insertion of a second age for some author. Then the condition will become false because $\$a$ is no longer a singleton tree, and the author must be deleted from the view. Hence, the view is not monotone and not multi-linear.

Similarly, consider the use of a parameter/value variable $\$x$ as a key variable in a path pattern *PPat* (or path expression *PExpr*). The value of the parameter/value variable can change during an update and this will make *PPat* (*PExpr*) refer to a *completely different* value. This change in the view is only possible by deletion of the previous value. Hence, such queries are non-monotone.

Therefore, we syntactically restrict WHAX-QL queries as follows:

Definition 1. *A* WHAX-*QL query is* maintainable *if no parameter/value variable* $\$x$ *occurs as a key variable or operand of some base operation* e_1 op e_2.

It is ususally not possible to rewrite a query into an equivalent, maintainable query. However, one can often replace the query by a similar query that returns the same expected result. For example, view V_1 can be transformed into a maintainable view V_1'' by binding $\$a$ to the label (i.e. the age value) of each Age edge, instead of the set of all ages.

```
V''1($db) =  where       <Person($n).Age.$a> </> in $db,
                         $a > 36
             construct   <MyPerson($n).Age.$a> </>
```

Base Variable Constraint. Consider view $V_3(\$db)$ in Sec. 4. Here, variable *$db* is used twice as a base variable to perform a join between persons and authors of publications. Intuitively, the view is *not* multi-linear for a database DB and update $\triangle DB$ (i.e. $V_3(DB \uplus \triangle DB) \neq V_3(DB) \uplus V_3(\triangle DB)$), since some persons or authors in $\triangle DB$ might join with authors (persons, respectively) in DB.

Fortunately, the view can be made multi-linear by replacing the second occurrence of *$db* with a fresh variable *$db'* and adding $\$db$ as a parameter variable:

```
V'3($db,$db') =
 where     <Person(Name:$n).Age> $a </> in $db,
           <Conf(@name:"STACS",@year:$y).Publ(Title:$t).Author($n)></> in $db'
 construct <Author(Name:$n).Age> $a </>,
           <Author(Name:$n).STACS(Year:$y).Title($t)> </>
```

The new multi-linear view $V_3'(\$db, \$db')$ is equivalent to $V_3(\$db)$, if applied to the same database: $V_3'(DB, DB) = V_3(DB)$.

Theorem 1. *A maintainable* WHAX-*QL view* $V(\$d_1, \ldots \$d_n)$ *is multi-linear in its parameters* $\$d_1, \ldots \$d_n$ *if all base variables* $\$db$ *of the same* where-construct-*expression are distinct and do not occur in the* construct-*clause.*

Lemma 1. *A maintainable view* $V(\$db_1, ..., \$db_n)$ *can always be transformed into an equivalent multi-linear view.*

Intuitively, the query is rewritten into a query $V'(\$db_1', ..., \$db_m')$ $(n \leq m)$ where each original parameter $\$db_i$ is replaced by several new parameters $\$db_j'$ in the new view.[4] The views are equivalent for databases $DB_1, ..., DB_n$, if each new parameter $\$db_j'$ maps to the database DB_i of the corresponding original parameter

[4] In computational geometry and linear algebra this rewriting is called *polarization* [30].

$\$db_i$. Note that an update $\triangle DB_i$ might be propagated into several parameters and the multi-linearity law must be applied several times to maintain the view.

A more complex example of rewriting is given in Appendix A. Details of the rewriting process can be found in [26].

Relational SPJ Views and WHAX-*QL Views.* We have argued in Sec. 3 that WHAX captures relational database instances with keys (but ignoring foreign key constraints). It can also be proven that WHAX-QL is a generalization of relational SPJ queries:

Lemma 2. *Relational SPJ views can be expressed as maintainable* WHAX *queries.*

6 Deletions

The pure form of multi-linearity can only be used to propagate insertion updates using (deep) union; it cannot be used for propagating deletion updates. The reason is analogous to that in relational SPJ views: a single tuple in the view may have multiple derivations from tuples in the base relations. It is therefore not clear whether a tuple in the view should be deleted (if the last derivation is eliminated) or whether the tuple should be kept (if there are more derivations).

Two approaches to deletions have been investigated for relational SPJ-views: 1) *View analysis* [9] and 2) view maintenance using *multi-set semantics* [21] or *counting* [6, 22].

The view analysis algorithm in [9] is a static decision procedure which accepts only views for which any tuple in the view is guaranteed to have exactly one derivation from the base relations. That is, it guarantees that that the union is *disjoint* under the insertion of new tuples.

Alternatively, one can keep track of the number of derivations for each distinct tuple in the view. *Multi-set semantics* for SPJ views keeps tuple duplicates and allows insertions/deletions to add/remove duplicates [21]. The *counting* approach [6, 22] annotates each tuple with its number of derivations. In contrast to the multi-set approach, counts can be negative and deletions can be treated as insertions of tuples with negative counts.

All these techniques are based on variations of the original set union operation (*disjoint*, *bag*, or *counting* union), and views are multi-linear with respect to the new union operation. Most importantly, the new deep union operations are *invertible*, i.e. given $R_1 \cup R_2 = R_3$ one can compute R_1 given R_3 and R_2 (denoted as $R_1 = R_3 - R_3$). Intuitively, invertible union operation allow the efficient propagation of deletions since the following multi-linearity law for deletion can be derived: $V(D_1, ..., D_i - \triangledown D_i, ..., D_n) = V(D_1, ..., D_i, ..., D_n) - V(D_1, ..., \triangledown D_i, ..., D_n)$.

Note that the deep union operation in WHAX is not invertible. For example, consider $A = \{\mathsf{Person}(\text{"Tom"}) : \{\}\}$, $B = \{\mathsf{Person}(\text{"Tom"}) : \{\mathsf{Age} : \text{"26"}\}\}$, and their deep union $A \uplus B = \{\mathsf{Person}(\text{"Tom"}) : \{\mathsf{Age} : \text{"26"}\}\}$, which is equal to B. However, the inverse operation $(A \uplus B) - B = B - B = \{\}$ is not equal to A.

Although it is surprisingly difficult to find a disjoint union operation [9] for WHAX (preliminary ideas of how to do this can be found in [26]), it is possible

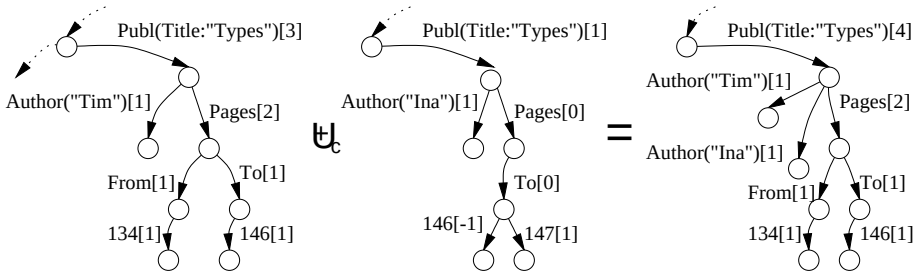

Fig. 7. Annotating a WHAX-tree with supports

to extend the counting technique. The approach is based on keeping a *derivation count* for each node in the view, and representing deletions as WHAX trees with negative counts.

6.1 Counting in WHAX

In the counting approach for SPJ views [6, 22], each tuple in the view is annotated with a count describing the number of its derivations. The count is increased/decreased by one whenever the tuple is inserted/deleted. Deletes can be represented as tuples with count -1 and a generalized "count" union $\uplus_c$ is sufficient to support inserts and deletes uniformly.

A similar approach is adopted in WHAX: Each edge with local identifier $l(k)$ in the WHAX-tree is annotated with a count c that describes the number of derivations for the edge: $l(k)[c]$. We call count c the *support of the edge*.

The tree on the left of Fig. 7 shows parts of the tree from Fig. 2 with supports. Leaf edges have support 1, and the support of the inner edges is the sum of the child's supports (the *indirect support* of the edge) and some (by default, zero) *direct support* for the edge itself. The indirect support of leaf edges is 0.

As in the relational model, insertions and deletions can be modeled as data values with positive or negative supports. The center of Fig. 7 shows the update tree for adding a new author "Ina" to publication "Types", and changing a page number from 146 to 147. To model this page change, the old edge with label 146 is deleted (support -1) and the new edge 147 is inserted (support 1). Note that the indirect support of Pages is (+1)+(-1)=0.

The deep union operator $\uplus_c$ for trees with counts is defined as follows:

$$\begin{array}{rll} v_1 \uplus_c v_2 ::= & \{l(k)[c_1+c_2] : (s_1 \uplus_c s_2) & \mid l(k)[c_1] : s_1 \in v_1, l(k)[c_2] : s_2 \in v_2, \\ & & \quad (c_1+c_2 \neq 0 \ \vee \ s_1 \uplus_c s_2 \neq \{\}) \ \} \cup \\ & \{l(k)[c] : s & \mid l(k)[c] : s \in v_1, l(k)[...] \notin \mathsf{dom}(v_2)\} \cup \\ & \{l(k)[c] : s & \mid l(k)[c] : s \in v_2, l(k)[...] \notin \mathsf{dom}(v_1)\} \end{array}$$

Note that merged edges are eliminated if their support is empty ($c_1 + c_2 = 0$) and they do not have children ($s_1 \uplus_c s_2 = \{\}$). Fig. 7 shows an example of the deep union on trees with supports.

Computing the Support for the View. Although the counting approach in the WHAX-model is intuitive, the supports in the view must be carefully chosen to preserve the semantics of queries, as the following nested view illustrates:

```
where     <Person(Name:$n)> $p </> in $db
construct <MyPers(Name:$n)>
             where <Age> $a </> in $p  construct <Age> $a </>
          </>
```

First, observe that the *indirect* support of path MyPers(Name:$n) in the view may be 0 if there is no Age edge in the base data. Its *direct* support must therefore be > 0, and is determined by the (direct+indirect) support of path Person(Name:$n) in the source. The *indirect* support of MyPers(Name:$n) is determined as usual by its children, and will be either 0 or 1 in our example.

Consider the syntax of where-construct-queries (Fig. 6). Let ϕ denote a valuation for the variables in the where-clause and let $PPat_i(\phi)$ ($PExpr_j(\phi)$) be the instantiation of pattern $PPat_i$ (path expression $PExpr_j(\phi)$, respectively) under this valuation. Let the support of path p, $Supp(p)$, be the the support (direct + indirect) of the *last* edge in the path. The direct support of a path ($DSupp(p)$) is defined similarly. Then, the direct support of each output path $PExpr_j(\phi)$ is determined by the product of the supports of each of the input paths $PPat_i$:[5]

$$\forall\, 1 \leq j \leq n \;\; : \;\; DSupp(PExpr_j(\phi)) = \prod_{1 \leq i \leq m} Supp(PPat_i(\phi))$$

Lemma 3. *A multi-linear* WHAX-*QL view* $V(\$d_1, \ldots \$d_n)$ *is also multi-linear under counting semantics using counting deep union* $\uplus_c$.

7 Aggregations in WHAX

A simple extension of the data model and WHAX-QL allows the use of aggregates in WHAX. First, the syntax is extended such that expressions e_i in the construct-clause of where-construct-queries (Fig. 6) can be any aggregate function sum(e), avg(e), count(), min(e), or max(e). The following example view determines the nu,ber of pages published for each conference and year:

```
where     <Conf(@name:$cn,@year:$cy).Publ(Title:$t).Pages>
                <From> $from </> <To> $to </> </> in $db,
construct <Conf(@name:$cn,@year:$cy).SumPages> sum($to-$from+1) </>
```

The key idea is that aggregate values produced by the construct-clause are merged through the deep union operation. For this, the WHAX model is extended as follows: Edges with annotation $l[c]$[6] can be annotated with aggregate tags: aggr :: $l[c]$. Fig. 8 illustrates the use of aggregates and deep union of aggregates. For example, Count::2[2] represents a publication count of 2 (with support 2) and min::24[1] a minimum author age of 24.

The deep union $\uplus_a$ over WHAX-trees v_1 and v_2 with aggregates has the following meaning: For any two matching aggregate elements aggr :: $l_1[c_1] \in v_1$ and aggr :: $l_2[c_2] \in v_2$, a new edge with aggr :: $l[c_1 + c_2]$ with $l = \mathsf{aggr}(l_1, l_2, c_1, c_2)$

[5] Intuitively, this works, since the product of supports is linear itself with respect to summation.

[6] The key k and the value v in $l(k) : v$ are always empty for aggregate edges.

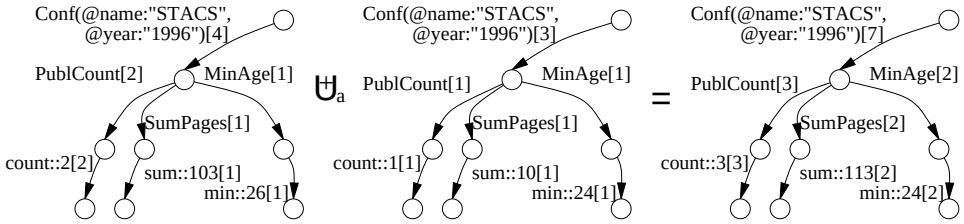

Fig. 8. Deep Union for Aggregates

is created. The aggregate combinator function $\mathsf{aggr}(l_1, l_2, c_1, c_2)$ is different for each aggregate, e.g. $\mathsf{sum}(l_1, l_2, c_1, c_2) = l_1 + l_2$ and $\mathsf{min}(l_1, l_2, c_1, c_2) = \mathsf{min}(l_1, l_2)$. Note that the count aggregate is a special case of the sum aggregate and the count aggregate value is in fact the same as the support of the edge.[7] Furthermore, note that min and max are undefined for $c_1 * c_2 < 0$, since min and max cannot be incrementally maintained under deletions.

8 Conclusion and Future Work

This paper describes a novel approach for incrementally maintaining views on hierarchical semistructured and relational databases with key constraints. In contrast to previous work on view maintenance in semistructured databases [3, 31], the WHAX architecture supports complex restructuring operations such as regrouping, flattening, and aggregations. Furthermore, while existing work only considers maintenance for atomic updates, our technique is applicable to large batch updates, which can reduce the refresh time of warehouses considerably.

The WHAX-model is a hierarchical data model that incorporates key constraints similar to the deterministic model [8] and the LDAP approach [29, 24]. The incremental view maintenance in WHAX is based on a fundamental mathematical equivalence: multi-linearity. This provides an intuitive and simple approach to view maintenance. Note that multi-linearity is also important for several optimization problems, such a query parallelization and pipelining.

The techniques in this paper are orthogonal to other results obtained in the relational setting. Most importantly, one can adopt previous algorithms to support deferred view maintenance [16] based on base logs and transition tables. The bag-union and bag-difference considered in [16] can be replaced with the deep union with counts. Furthermore, detection mechanisms for irrelevant updates [5] can be adopted to WHAX-updates: Before an update is sent to the view, it is verified whether the where-clause can actually match the update.

Several research problems remain to be investigated. Our approach currently does not support ordered data structures, which are important for XML and document-oriented databases. Ordered structures are difficult to maintain, if updated elements are identified over their position. Positions are "dynamic" in two essential ways: 1) positions of elements might change during updates, and 2) positions may be different in views. Therefore, a mapping between "dynamic"

[7] This can be used for the implementation of avg.

positions and "static" keys must be provided. This mapping changes during updates and efficient index schemes (e.g. based on B-trees) are required.

Furthermore, we do not consider several extensions to the query language, such as negation and recursion [22], and did not address the issue of allowing references within the WHAX-tree. A prototype of the WHAX system is in progress and we expect to obtain experimental result that underscore the efficiency of the WHAX approach.

Acknowledgements: We would like to thank Peter Buneman and Wang-Chiew Tan for fruitful discussions about the deterministic data model and Jean Gallier for comments on polarization.

References

1. S. Abiteboul. On views and XML. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 1–9, Philadelphia, PA, May 1999.
2. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, San Francisco, California, 1999.
3. S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Int'l Conference on Very Large Databases (VLDB)*, pages 38–49, New York City, NY, August 1998.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1996.
5. J.A. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.
6. J.A. Blakeley, P.-A. Larson, and F. Tomba. Efficiently updating materialized views. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, DC, May 1986.
7. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.
8. P. Buneman, A. Deutsch, and W.C. Tan. A deterministic model for semi-structured data. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.
9. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *17th Int'l Conference on Very Large Data Bases (VLDB)*, pages 577–589, Barcelona, Spain, September 1991. Morgan Kaufmann.
10. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In *Proceedings of the Information Processing Society of Japan Conference*, Tokyo, Japan, October 1994.
11. I.A. Chen, A.S. Kosky, V.M. Markowitz, and E. Szeto. Constructing and maintaining scientific database views. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 237–248, Olympia, Washington, 1997.
12. V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *ACM SIGMOD Conference on Management of Data*, 2000. (to appear).

13. J. Clark. XSL transformations (XSLT), version 1.0. *W3C Proposed Recommendation*, Octover 1999. Available as `http://www.w3.org/TR/xslt`.
14. J. Clark and S. DeRose. XML path language (XPath), version 1.0. *W3C Working Draft*, August 1999. Available as `http://www.w3.org/TR/xpath`.
15. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversation! In *Proceedings of ACM-SIGMOD International Conference*, pages 177–188, Seattle, Washington, June 1998.
16. L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *1996 ACM SIGMOD Conference*, pages 469–480, Montreal, Canada, June 1996.
17. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: a query language for XML. *W3C Note Note-xml-ql-19980819*, 1998. Available as `http://www.w3.org/TR/NOTE-xml-ql`.
18. A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with S-TORED. In *Proceedings of the ACM-SIGMOD International Conference*, pages 431–442, Philadelphia, May 1999.
19. D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.
20. D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized OQL views. *Lecture Notes in Computer Science (LNCS)*, pages 52–66, December 1997.
21. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD Conference*, pages 328–339, San Jose, California, May 1995.
22. A. Gupta, I. S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Conference*, pages 157–166, Washington, DC, May 1993.
23. A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.
24. H.V. Jagadish, L.V.S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Philadelphia, June 1999.
25. H.A. Kuno and E.A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), 1998.
26. H. Liefke and S.B. Davidson. Efficient view maintenance in XML data warehouses. Technical Report MS-CIS-99-27, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, November 1999.
27. J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *W3C Query Languages Workshop (QL'98)*, Boston, December 1998.
28. C.J. Stoeckert, F. Salas, B. Brunk, and G.C. Overton. EpoDB: a prototype database for the analysis of genes expressed during vertebrate erythropoiesis. *Nucleic Acids Research*, 27(1), January 1999.
29. M. Wahl, T. Howes, and S. Killes. Lightweight directory access protocol (v3). Technical report, IETF, December 1997.
30. H. Weyl. *The Classical Groups. Their Invariants and Representations.* Princeton Mathematical Series, No. 1. Princeton University Press, second edition, 1946.
31. Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *14th Int'l Conference on Data Engineering (ICDE)*, pages 116–125, Orlando, Florida, February 1998.

A An Example of Multi-Linearization

Consider the following query, which extract authors and page numbers from publications:

```
V4($db)=
    where     <Conf(@name:$n,@year:$y).Publ(Title:$t)> $p </> in $db
    construct <Publ(Title:$t,Conf:$n,Year:$y)>
                  where     <Pages> $pages </> in $p,
                            <Author($a)> </>   in $p,
                            <Person($a)> </>   in $db
                  construct <Pages> $pages </>,
                            <Author($a)> </> </>
```

Here, the base variable *$db* occurs in the outer and inner where-clause, and *$p* occurs twice in the inner where-clause. We can rename the second occurrence of *$p* to fresh variable *$p'* and the inner occurrence of *$db* to *$db'*. To ensure that $\$p' = \$p$, we duplicate the path binding for *$p* in the outer where-clause and introduce fresh variable *$db"*. Variables *$db'* and *$db"* become additional parameters of the view:

```
V'4($db,$db',$db'')
 =where     <Conf(@name:$n,@year:$y).Publ(Title:$t)> $p  </> in $db
            <Conf(@name:$n,@year:$y).Publ(Title:$t)> $p' </> in $db''
  construct <Publ(Title:$t,Conf:$n,Year:$y)>
               where     <Pages> $pages </> in $p,
                         <Author($a)> </>   in $p',
                         <Person($a)> </>   in $db'
               construct <Pages> $pages </>,
                         <Author($a)> </> </>
```

The view $V'4(\$db, \$db', \$db'')$ is multi-linear and $V'4(DB, DB, DB) = V4(DB)$ holds for any database DB.