A Tree Comparison Approach to Detect Changes in Data Warehouse Structures

Johann Eder, Christian Koncilia, and Karl Wiggisser

University of Klagenfurt
Dep. of Informatics-Systems
{eder,koncilia,wiggisser}@isys.uni-klu.ac.at

Abstract. We present a technique for discovering and representing changes between versions of data warehouse structures. We select a tree comparison algorithm, adapt it for the particularities of multidimensional data structures and extend it with a module for detection of node renamings. The result of these algorithms are so called edit scripts consisting of transformation operations which, when executed in sequence, transform the earlier version to the later, and thus show the relationships between the elements of different versions of data warehouse structures. This procedure helps data warehouse administrators to register changes. We describe a prototypical implementation of the concept which imports multidimensional structures from Hyperion Essbase data warehouses, compares these versions and generates a list of differences.

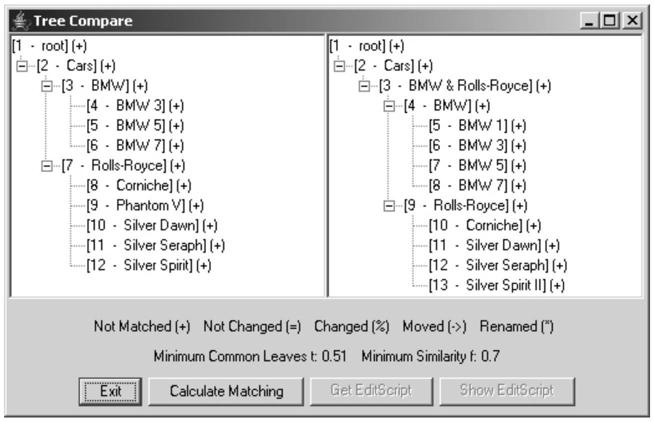
1 Introduction and Motivation

Data warehouses provide sophisticated features for aggregating, analyzing, and comparing data to support decision making in companies. The most popular architecture for data warehouses are multidimensional data cubes, where transaction data (called cells, fact data or measures) are described in terms of master data (also called dimension members). Usually, dimension members are hierarchically organized in dimensions, e.g., $university \leftarrow faculty \leftarrow department$ where $B \leftarrow A$ means that A rolls-up to B.

As most data warehouses typically comprise a time dimension, available data warehouse systems are able to deal with changing measures, e. g. , changing margin or sales. They are however not able to deal with modifications in dimensions, e. g. , if a new faculty or department is established, or a faculty is split into two, or departments are joined.

In [1] we presented the COMET approach, a temporal data warehouse metamodel, which allows to represent not only changes of transaction data, but also of schema, and structure data. The COMET model can then be used as basis of OLAP tools which are aware of structural changes and permit correct query results spanning multiple periods and thus different versions of dimension data.

Temporal data warehouses, however, need a representation of changes which took place between succeeding structure versions of the dimension data. Typically, a change log is not available, but the data warehouse administrator has



```
[1 - root] (+)
  [2 - Cars] (+)
    [3 - BMW] (+)
      [4 - BMW 3] (+)
      [5 - BMW 5] (+)
      [6 - BMW 7] (+)
    [7 - Rolls-Royce] (+)
      [8 - Corniche] (+)
      [9 - Phantom V] (+)
      [10 - Silver Dawn] (+)
      [11 - Silver Seraph] (+)
      [12 - Silver Spirit] (+)
```

```
[1 - root] (+)
  [2 - Cars] (+)
    [3 - BMW & Rolls-Royce] (+)
      [4 - BMW] (+)
        [5 - BMW 1] (+)
        [6 - BMW 3] (+)
        [7 - BMW 5] (+)
        [8 - BMW 7] (+)
      [9 - Rolls-Royce] (+)
        [10 - Corniche] (+)
        [11 - Silver Dawn] (+)
        [12 - Silver Seraph] (+)
        [13 - Silver Spirit II] (+)
```

Not Matched (+)   Not Changed (=)   Changed (%)   Moved (->)   Renamed (*)

Minimum Common Leaves t: 0.51   Minimum Similarity f: 0.7

Fig. 1: Our running Example

to create and maintain a mapping between snapshots of the dimension structure. The contribution of this paper is to assist data warehouse administrators in detecting and describing these structural changes, i.e. insertion, deletion and rearrangement of dimension members. Besides these structural changes in tree structured dimensions there are also semantical changes (e.g. merging or splitting of dimension members) which cannot be discovered by looking at dimension data alone. For these semantical changes we have developed a change detection procedure based on data mining techniques [2]. Since this method analyzes outliers in the cell values, the computational costs are rather high. Therefore, it is desirable to find structural changes first by mere structural comparisons with more efficient algorithms.

For detecting structural changes, we present a novel comparison algorithm for multidimensional data by adopting and extending an existing tree comparison algorithm to the particularities of data warehouses. One of the main advantages of our approach is that beside detecting changes like insert, delete and update an element on the instance level, it also supports the detection of key modifications (e.g. renamings of departments).

Such an approach can only be heuristic in nature and can never be complete, as we will explain below. Therefore, this method is intended to support the administrator but not to fully automate the task. Since dimension members can be numerous (e.g. product catalog), the productivity gain will be significant.

Throughout the rest of this paper, we will use the following running example. Consider a car dealer who wants to keep track of his sales. For this, he implements a data warehouse with several dimensions. For sake of simplicity, we take a closer look at only one of these dimensions, namely the Cars dimension.

The left tree in Fig. 1 depicts the original version of this dimension. As can be seen, this dealer sells two different brands: *BMW* and *Rolls-Royce*. Each brand consists of several different car types. For instance, *Corniche*, *Phantom V* and *Silver Dawn* are different Rolls-Royce cars. The right tree in Fig. 1 shows the subsequent version of this dimension. As can be seen, different modifications have been made: first of all, both brands united and are now known as *BMW & Rolls-Royce*. A new car was introduced, namely *BMW 1*. Another car, *Phantom V*, is no longer part of the product portfolio. *Silver Spirit* has been renamed and is now known as *Silver Spirit II*. Moreover, for all Rolls-Royce cars power is no longer given in kW but in horsepower.

The rest of this paper is organized as follows: in section 2 we will briefly discuss different approaches in temporal data warehousing and tree comparison. In section 3 we will introduce the necessary data structure and operations. In the subsequent section, we will discuss our algorithm in detail, including some remarks on the complexity. In section 5 we will present the implementation of our approach. We will conclude in the last section of this paper.

2 Related Work

Our approach builds on the techniques developed in two different research areas, namely temporal data warehousing and tree comparison algorithms. As we will discuss later, all these tree comparison approaches are not designed for the special needs and requirements of data warehouses, i.e., the comparison of different versions of multidimensional databases. Therefore, extensions and adoptions for these particularities are necessary.

During the last years different temporal data warehouse approaches have been published. [3–8] are just a few of them. They differ in different aspects. For instance some of them support only changes on the schema level (e.g. [7]) or on the instance level (e.g. [3] and [4]), some support changes on both, the schema and the instance level (e.g. [8]).

There are several tree comparison algorithms. For instance, Zhang and Shasha [9] worked on ordered trees. The Stanford DB Group proposed algorithms for ordered [10] and unordered [11] trees. Nowadays as XML is very popular many algorithms for comparing XML documents, which are also trees, have been defined, for instance the approach of Cobena, Abiteboul and Marian [12] or the approach of Wang, DeWitt and Cai [13].

3 Comparison of Data Warehouse Structures

3.1 The Data Structure

From our running example it is easy to see, that a DWH structure can be represented by a tree. To define our data structure formally, we introduce a tree $\mathbb{T} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V} = \{m_1, \ldots, m_m\}$ is a set of nodes and $\mathbb{E} = \{e_1, \ldots e_n\}$ is a set of edges. A node is representing a dimension member of a DWH cube, therefore node m_i is defined as triple $m_i = \langle id, label, value \rangle$, where *id* is a unique identifier for each node, *label* is the dimension member's name and *value* is an object containing all other characteristics – i.e. formula, alias, description, consolidation, ... – of the dimension member. The id may stem from the data source or may be generated during the process of building the tree. An edge e_i is a hierarchical relation between two members and therefore defined as tuple $e_i = \langle m_j, m_k \rangle$, meaning that m_j is the parent of m_k. $\mathbb{E}^+$ is the transitive closure of $\mathbb{E}$ and therefore holding all ancestor relations.

Depending on the underlying DHW system, a tree can either be seen as *ordered tree* or *unordered tree*. In an ordered tree, members have a designated order within their parents, whereas in an unordered tree they don't. To be able to identify the nodes in the tree and to map them to the underlying DWH-system, we define labels to be unique in an unordered tree. As we always can identify a node through its parent and position within the parent in ordered trees, labels don't have to be unique in such a tree.

3.2 Transformations and Operations

We compare two versions of a DWH structure. Therefore we represent each version by a tree. Between this two versions a sequence of tree transformations occurred. Our goal is to find this transformations.

We identified five possible operations on the member level, which will be discussed in this section. Hereafter $t_1 : \mathbb{T} = (\mathbb{V}, \mathbb{E})$ references the old version of a tree and $t_2 : \mathbb{T} = (\mathbb{V}', \mathbb{E}')$ references the new version. Hence $\mathbb{V}$,$\mathbb{E}$ and $\mathbb{V}'$,$\mathbb{E}'$ are the sets of nodes and edges before and after the transformation respectively.

1. `DELETE` (`DEL(`m_i`)`): The dimension member m_i is deleted from the DWH structure. A node can only be deleted if it does not have children.
   (a) Precondition: $m_i \in \mathbb{V}$, $\exists e_j = \langle \_, m_i \rangle \in \mathbb{E}$, $\nexists e_k = \langle m_i, \_ \rangle \in \mathbb{E}$
   (b) Operation: $\mathbb{V}' = \mathbb{V} \backslash \{m_i\}$, $\mathbb{E}' = \mathbb{E} \backslash \{e_j\}$
   (c) Postcondition: $m_i \notin \mathbb{V}'$, $\nexists e = \langle \_, m_i \rangle \in \mathbb{E}'$
2. `INSERT` (`INS((`m_i, l, v`),`m_j, m_k`)`): The dimension member m_i with label l and value v is inserted as child of node m_j directly after node m_k. If m_k is $NULL$, m_i is inserted as first child of m_j. For unordered trees m_k may always be $NULL$.
   (a) Precondition: $m_i \notin \mathbb{V}$, $m_j \in \mathbb{V}$, $m_k = NULL \vee (m_k \in \mathbb{V} \wedge (m_j, m_k) \in \mathbb{E})$
   (b) Operation: $\mathbb{V}' = \mathbb{V} \cup \{m_i\}$, $\mathbb{E}' = \mathbb{E} \cup \{\langle m_j, m_i \rangle\}$
   (c) Postcondition $m_i \in \mathbb{V}'$, $\langle m_j, m_i \rangle \in \mathbb{E}'$
3. `MOVE` (`MOV(`m_i, m_j, m_k`)`): The dimension member m_i is moved to the parent m_j or is moved within its parent m_j to be directly after m_k. If m_k is $NULL$, m_i becomes the first child of m_j. For unordered trees m_k may be $NULL$.
   (a) Precondition: $m_i \in \mathbb{V}$, $m_j \in \mathbb{V}$, $m_k = NULL \vee (m_k \in \mathbb{V} \wedge (m_j, m_k) \in \mathbb{E})$, $\langle m_i, m_j \rangle \notin \mathbb{E}^+$
   (b) Operation: $\mathbb{V}' = \mathbb{V}$, $\mathbb{E}' = (\mathbb{E} \backslash \langle \_, m_i \rangle) \cup \{\langle m_j, m_i \rangle\}$
   (c) Postcondition $\langle m_j, m_i \rangle \in \mathbb{E}'$
4. `UPDATE` (`UPD(`m_i, v`)`): The characteristics of a dimension member m_i – i.e. the node's value – is changed to v.
   (a) Precondition: $m_i = \langle id, label, \_ \rangle \in \mathbb{V}$
   (b) Operation: $\mathbb{E}' = \mathbb{E}$,$m_i' = \langle id, label, v \rangle$, $\mathbb{V}' = (\mathbb{V} \backslash m_i) \cup \{m_i'\}$
   (c) Postcondition: $m_i' \in \mathbb{V}' = \langle id, label, v \rangle$

5. `RENAME` (`REN(`m_i, l`)`): The name of a dimension member m_i – i.e. the node's label – is changed to l.
   (a) Precondition: $m_i = \langle id, \_, value \rangle \in \mathbb{V}$
   (b) Operation: $\mathbb{E}' = \mathbb{E}$,$m_i' = \langle id, l, value \rangle$, $\mathbb{V}' = (\mathbb{V} \backslash m_i) \cup \{m_i'\}$
   (c) Postcondition: $m_i' \in \mathbb{V}' = \langle id, label, v \rangle$

One may argue that `MOVE` is not a basic operation, as it may be composed using `DELETE` and `INSERT`. This is true for most cases, but in our context we want to identify relations between the old and the new version of a member. This enables us to define what we called *transformation functions* to transform data between different versions [1].

We distinguish between `UPDATE` and `RENAME` because in many commercial systems, e.g. Hyperion Essbase, the member's name is a key and we want to distinguish between value changes and key changes of a member.

3.3 Comparison of Dimension Members

To compare different versions of dimension members, we define a $compare(x, y)$ function that takes into account the various characteristics of a dimension member. The $compare(x, y)$ function takes two value-objects as defined above as parameters and returns a value in the range of $[0, 1]$ as *degree of similarity*. Characteristics to be compared may for instance include formulae, user defined attributes (UDAs), aliases, comments, shared member relations, consolidation function and other DWH system specific attributes.

Some of these attributes are more distinguishing than others. So if for example two nodes have exactly the same formula and three out of four UDAs in common but a different consolidation function, it's rather likely that they represent the same element, so $compare(x, y)$ gives a value near to 1. On the other hand, if the consolidation function is the same, but one is a shared member and the other is not, it may be quite unlikely that these nodes represent the same element, hence $compare(x, y)$ results in a value near to 0.

We define these weighting factors to be *parametrizeable*. Hence the user may decide what "similar" exactly means in his situation.

4 Treecomparison and Extensions

As tree comparison is a well explored area, there was no need to develop a new comparison algorithm for trees. Instead we evaluated different existing methods in order to find the ideal base for our approach.

In [10] Chawathe et al. present an algorithm for comparing two versions of a tree. The result of the algorithm is an *editscript* consisting of tree transformations which transforms t_1 into t_2. The time complexity of this method is $O(nd + d^2)$, where n is the *number of leaves* in the tree and d is a measure for the *difference of the two versions*. As we assume that there are only a few changes between two versions, we can say that $d \ll n$. We chose Chawathe et al.'s algorithm as base for our work because of a couple of reasons:

1. Its support of the essential `MOVE` operation.
2. Its low time complexity.
3. Its ready-to-use edit script as representation for changes.
4. It can be used for ordered and unordered trees.
5. It's easy to adapt this approach for our own needs.

In this section we will introduce Chawathe et al.'s treecomparison algorithm and our extensions to make it applicabe for our domain. We will also give a few complexity considerations on our extension.

4.1 Treecomparison in Detail

Chawathe et al.'s algorithm is defined on ordered trees but is applicable on unordered trees as well. Each node has a label and a value which are obtained from the data source. Furthermore, each node has a unique identifier which can either stem from data source or be generated during data extraction. The id may not identify nodes over different versions, so nodes representing the same elements in different versions may have different ids and vice versa.

The result of this procedure is a so called *edit script* which transforms t_1 into t_2. This edit script is a sequence consisting of four atomic operations: `INSERT, UPDATE, MOVE` and `DELETE`. As the matching of nodes between versions relies on node labels, change of node labels (`RENAME`) is not supported.

Chawathe et al. define their `INSERT` and `MOVE` operations index based, meaning that determining the position where to insert a node they use the index within the children of a node. To adapt their approach to our predecessor based data structure we modified the *FindPos(x)* function to return the predecessor for the node or *NULL* if there is no predecessor instead of the index.

The algorithm compares two tree versions: $t1 : \mathbb{T}$ and $t2 : \mathbb{T}$. For simplicity reasons we write $x \in t$ meaning that node x is an element of the node set $\mathbb{V}$ of tree t. The comparison algorithm needs the two tree versions and a matching set as input. The matching set $\mathbb{M}$ is a set of pairs (a, b) with $a \in t_1$ and $b \in t_2$, where a and b represent the same member in the two versions.

As the node ids may not identify elements over versions other *matching criterions* for nodes have been defined. Before we can give a definition of these criterions, we have to introduce some terms: $l(x)$ and $v(x)$ give the label and value of node x respectively. The function $common(x, y)$ gives $\{(v, w) \in \mathbb{M} | v$ is leaf descendant of x and w is leaf descendant of $y\}$ the so called *Common Leaves* of x and y. Finally $|x|$ is the number of leaf descendants of node x. We also slightly modified the original matching criterions for leaf nodes for sake of simplicity. So two nodes x and y are seen as representing the same element iff one of the following conditions holds:

1. x and y are *leaves*, $l(x) = l(y)$, and $compare(v(x), v(y)) \geq f$, where f is a user defined threshold in the range $[0, 1]$. f is called *Minimum Similarity.*
2. x and y are *inner nodes* and $l(x) = l(y)$ and $\frac{|common(x,y)|}{\max(|x|,|y|)} \geq t$ for $\frac{1}{2} < t \leq 1$. We call t the *Minimum Common Leaves*. t is defined by the user.

It can easily be seen that in addition to their label leaf nodes are compared using their values, but inner nodes are only compared using their descendants. For sure changes in the values of inner nodes will be detected later on in the update phase. Zhang [14] proposes a similar matching constraint for inner nodes.

As labels have to be equal for allowing matches, renamings of nodes – i.e. changes of labels – can not be detected in the approach of Chawathe et al.

After the matching set $\mathbb{M}$ is calculated the edit script can be generated. This happens in five phases. The phases will only be described very briefly. Please have a look at [10] for details.

1. Update Phase: For all pairs of nodes $(x, y) \in \mathbb{M}$ where $v(x) \neq v(y)$ an update operation is generated.
2. Align Phase: For all misaligned nodes, i.e. if the order is different in the two versions, appropriate move operations are generated. This step may be omitted with unordered trees.
3. Insert Phase: For all unmatched nodes $x \in t_2$ an insert is generated.
4. Move Phase: For all pairs of nodes $(x, y) \in \mathbb{M}$ such that $(p(x), p(y)) \notin \mathbb{M}$ ($p(x)$ is the parent of x) a move to the correct parent is generated.
5. Delete Phase: For all unmatched nodes $x \in t_1$ a delete operation is generated.

4.2 Detecting Renaming of Nodes

With the plain algorithm described above changes of labels – which are renamings of dimension members in our case – cannot be detected . Hence, our approach extends Chawathe et al.'s approach in order to *detect label changes*.

In the original algorithm a renaming will always result in two operations, one `DELETE` and one `INSERT`. But this does not represent the real semantics. We want the different versions of members to be connected over structure versions. Therefore, we introduce the new operation `RENAME`, denoted as `REN`(x, l) where x is the node to be renamed and l is its new label.

The renaming detection takes place after the matching set calculation but before generating the edit script. All nodes $x \in t_1$ and $y \in t_2$ which are not part of a matching are possible candidates for beeing renamed nodes. They are added to the set *OldNames* $\mathbb{O}$ and *NewNames* $\mathbb{N}$. We define $\mathbb{O} = \{x \in t_1 | \nexists (a, b) \in \mathbb{M} \bullet a = x\}$ and $\mathbb{N} = \{y \in t_2 | \nexists (a, b) \in \mathbb{M} \bullet b = y\}$

The number of renamings is limited by $\min(|\mathbb{O}|, |\mathbb{N}|)$. So if one of the sets is empty, no renaming is possible. For reduction of complexity we only consider *renamings within the same parent* to be detected. So parents of possibly renamed nodes have to match. But as the parents may also be renamed and therefore no match is possible yet, this rule may foreclose detecting many renamings. Hence we also consider possibly renamed parents as matched parents. For this purpose we define a set *PossibeRenamings* $\mathbb{P}$ as follows: $\mathbb{P} = \{(a, b) | a \in \mathbb{O} \wedge b \in \mathbb{N} \bullet (p(a), p(b)) \in \mathbb{M} \vee (p(a), p(b)) \in \mathbb{P}\}$. So $\mathbb{P}$ is the set of all pairs of nodes from $\mathbb{O}$ and $\mathbb{N}$ where either the parents are matched or possibly renamed. One can create $\mathbb{P}$ during a Top-Down traversal of the tree or by a repeated application of the build rule until the set remains stable.

We also define an *order* on $\mathbb{P}$ which is important for the appropriate traversal of $\mathbb{P}$ in the next step. To define this order formally, we have to introduce the *level* of a member x as the height of the subtree rooted at x. So leaf nodes are at level 0, all inner nodes have levels greater than 0. We define the two operators "$<$" and "$>$" on pairs $(a, b) \in \mathbb{P}$ as follows:

1. $\forall (a, b), (x, y) \in \mathbb{P} : (a, b) < (x, y) \Leftrightarrow \min(lev(a), lev(b)) < \min(lev(x), lev(y))$
2. $\forall (a, b), (x, y) \in \mathbb{P} : (a, b) > (x, y) \Leftrightarrow \min(lev(a), lev(b)) > \min(lev(x), lev(y))$

The order within leaf node pairs and inner node pairs respectively is irrelevant. So if $\mathbb{P}$ is traversed following this order, all pairs containing at least one leaf node will be examined before any pair containing only inner nodes.

The similarity check for renamed nodes has in principle the same constraints as mentioned before, but of course, labels are not checked for equality. Inner nodes are seen as similar, if they have enough common leaves, leaf nodes are seen as similar, if their values don't differ too much. So we define a function $likelyRenamed(a, b)$ which checks if $(a, b) \in \mathbb{P}$ is a likely renaming, with f and t as defined above, to return *true* iff one of the following conditions holds:

1. a and b are leaf nodes and $compare(v(a), v(b)) \geq f$
2. a and b are inner nodes and $\frac{|commonRename(a,b)|}{\max(|a|,|b|)} \geq t$

The function $commonRename(x, y)$ gives $\{(v, w) | v$ is leaf descendant of x and w is leaf descendant of y, and $(v, w) \in \mathbb{M}$ or $(v, w) \in \mathbb{L}\}$, so all common leaves plus all common likely renamed leaf nodes.

In an ordered tree within the calculation of $likelyRenamed(a, b)$ one may also take into account the *siblings* of the nodes. So if the predecessors and the successors of a and b respectively, match, one may increase the degree of similarity a bit, although it must not reach 1, as this would mean identical.

With the help of $likelyRenamed(a, b)$ we split $\mathbb{P}$ into two disjoint sets *LikelyRenamings* $\mathbb{L}$ and *UnlikelyRenamings* $\mathbb{U}$. $\mathbb{L}$ contains all pairs of nodes which are sufficiently similar to be seen as representing the same element. All other elements of $\mathbb{P}$ are moved to $\mathbb{U}$. As for one node only one real renaming can have happened, the following restriction has to hold for $\mathbb{L}$: $\forall (a, b), (x, y) \in \mathbb{L} \bullet a = x \Leftrightarrow b = y$. So a node can only appear in at most one pair in $\mathbb{L}$. If more than one likely renaming for one node is detected, we define the one with the highest similarity of the involved nodes to go to $\mathbb{L}$, all others are moved to $\mathbb{U}$.

Because of $\mathbb{P}$'s order all leaf nodes will be handled first. Likely matchings of leaf nodes are considered in the similarity check of inner nodes, i.e. leaf nodes contained in a likely matching will be treated as common leaf nodes. Therefore, it is important to follow $\mathbb{P}$'s order during this step, as otherwise renamed inner nodes may not be detected correctly.

The renaming detection component cannot replace human interaction. It relies on heuristics which may return a wrong result. Therefore, a human user has to acknowledge all detected renamings by checking $\mathbb{L}$ and $\mathbb{U}$. All renamings confirmed by the user are moved into the set *Renamings* $\mathbb{R}$. For all pairs $(a, b) \in \mathbb{R}$ a rename operation `REN(a,l(b))` is generated and (a, b) is added to $\mathbb{M}$.

After this verbal description, we now formally describe the steps which are necessary to detect renamings of nodes. This phase takes place after the matching phase, so the matching set $\mathbb{M}$ already exists.

```
1. $\mathbb{O} = \{x \in t_1 | \nexists (a,b) \in \mathbb{M} \bullet a = x\}$
2. $\mathbb{N} = \{y \in t_2 | \nexists (a,b) \in \mathbb{M} \bullet b = y\}$
3. $\mathbb{P} = \{(a,b) | a \in \mathbb{O}, b \in \mathbb{N} \bullet (p(a),p(b)) \in \mathbb{M} \vee (p(a),p(b)) \in \mathbb{P}\}$
4. $\forall (a,b) \in \mathbb{P}$ in traversal order$\bullet$
   (a) if $likelyRenamed(a,b) = true$
       i. $\mathbb{P} = \mathbb{P} \backslash \{(a,b)\}$, $\mathbb{L} = \mathbb{L} \cup \{(a,b)\}$
       ii. $\forall x \in t_1 | (x,b) \in \mathbb{P} \bullet \mathbb{P} = \mathbb{P} \backslash \{(x,b)\}$, $\mathbb{U} = \mathbb{U} \cup \{(x,b)\}$
       iii. $\forall y \in t_2 | (a,y) \in \mathbb{P} \bullet \mathbb{P} = \mathbb{P} \backslash \{(a,y)\}$, $\mathbb{U} = \mathbb{U} \cup \{(a,y)\}$
   (b) if $likelyRenamed(a,b) = false$
       i. $\mathbb{P} = \mathbb{P} \backslash \{(a,b)\}$, $\mathbb{U} = \mathbb{U} \cup \{(a,b)\}$
5. Let the user acknowledge all real renamings and insert them to $\mathbb{R}$
6. $\forall (a,b) \in \mathbb{R} \bullet$
   (a) Generate operation REN(a,l(b))
   (b) $\mathbb{M} = \mathbb{M} \cup \{(a,b)\}$
```

The algorithm may not detect all renamings. For instance, if a node is renamed, its value changed very much and it is moved to another parent, then the renaming will not be detected. For complexity purposes only renamed node within the same parent will be considered. One may omit this restriction but this will increase runtime complexity considerably.

4.3 Complexity Considerations

As one of the main targets during the definition of the renaming component was not to worsen the overall runtime complexity too much, i.e. stay linear with the number of nodes in the tree, we adhere to the restriction mentioned before. Hereafter we will briefly calculate the space and runtime complexity of our renaming detection to show that it does not change the overall runtime complexity class of $O(nd + d^2)$.

Let r be the number of renamings occurred and let d be the number of edit operations returned by the original algorithm. (In fact d is called "weighted edit distance" which correlates with the number of edit operations.) As this algorithm does not handle changes in labels, every renaming will result in two operations – one `DELETE` and one `INSERT` – so $r \leq \frac{d}{2}$.

The size of the sets $\mathbb{O}$ and $\mathbb{N}$ is limited by the d. As $\mathbb{P}$ is a subset of $(\mathbb{O} \times \mathbb{N})$, the size of $\mathbb{P}$ is limited by d^2. Since $|\mathbb{O}| + |\mathbb{N}| \leq d$ we can drop the upper bound for $|\mathbb{P}|$ to $\frac{d^2}{4}$. The sets $\mathbb{L}$ and $\mathbb{U}$ are subsets of $\mathbb{P}$. As every node can only be part of one likely renaming $|\mathbb{L}| \leq r \leq \frac{d}{2}$, $\mathbb{R}$'s size has to be $r \leq \frac{d}{2}$.

The sets $\mathbb{O}$ and $\mathbb{N}$ may be created during the matching phase so no additional operations are needed. In step 3 one has to check for all possible pairs, if their parents (possibly) match. This gives at most $\frac{d^2}{4}$ operations. In step 4 for every pair $(a,b) \in \mathbb{P}$ a comparison is done. This gives again at most $\frac{d^2}{4}$ operations. The generation of `RENAME` operations in step 6 results in at most $r \leq \frac{d}{2}$ operations. So the whole renaming detection component results in at most $\frac{d^2+d}{2}$ operations which lies in $O(d^2)$. Compared to the basic algorithm's complexity of $O(nd+d^2)$ this doesn't change the algorithm's time complexity class.

5 Implementation

We implemented our approach prototypically in Java. In this prototype the user is able to import and compare two different cubes from the commercial multidimensional database Hyperion Essbase.

After importing both cubes from Hyperion Essbase, the prototype presents both versions as trees (see Fig. 1). The left tree represents the old version and right tree the new version of the data warehouse. The user triggers the matching procedure from the interface. Two elements match, if their names are equal and their contents are similar. The user may define what similar exactly means, i.e. she/he can define different weighing factors for different attributes of elements. For instance, the user could define that it is very likely that two elements are equal, if the UDAs (User Defined Attributes) of both elements are equal.

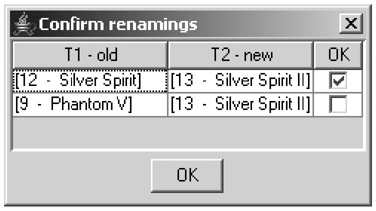
After the matchings are calculated the systems tries to find renamed nodes. Nodes which were not matched in the first step may be renamed nodes. Figure 2a shows how the user can acknowledge the renamings found by the system. The resulting trees are presented to the user so she/he can evaluate, if his weighting factors are adequate. After the user confirmed the matchings found by the system, the system starts to generate the "*edit script*". This *edit script* is a list of operations (Insert, Update, Delete, Move and Rename) that, if applied to the old structure versions, leads to the new structure version.

The resulting *EditScript* of our running example is shown in Fig.2b. Figure 2c shows the two trees after the edit script is calculated. Different symbols are used to depict different types of modifications that where detected. "`=`" means that a corresponding, unchanged dimension member has been found, "`->`" means that a member has been moved to another position, "`%`" means that the corresponding member has been changed, e.g., that a user defined attribute has been modified, "`+`" means that no corresponding member in the other tree could be found. Finally, "`*`" means that a member has been renamed.

We also applied our prototype on a larger cube with about 16.400 members and 8 differences – 2 inserts, 1 delete , 2 moves, 1 update, and 2 renames – between the versions. The matching and the editscript generation took in average 0.6 and 1.15 seconds respectively. Hence see that this approach may also be applied on large cubes in reasonable time. All changes were recognized correctly.

6 Conclusions

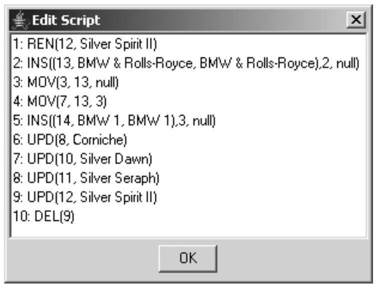
For the validity of OLAP queries spanning several periods of data collection it is essential to be aware of the changes in the dimension structure of the



Confirm renamings

T1 - old	T2 - new	OK
[12 - Silver Spirit]	[13 - Silver Spirit II]	✓
[9 - Phantom V]	[13 - Silver Spirit II]	☐

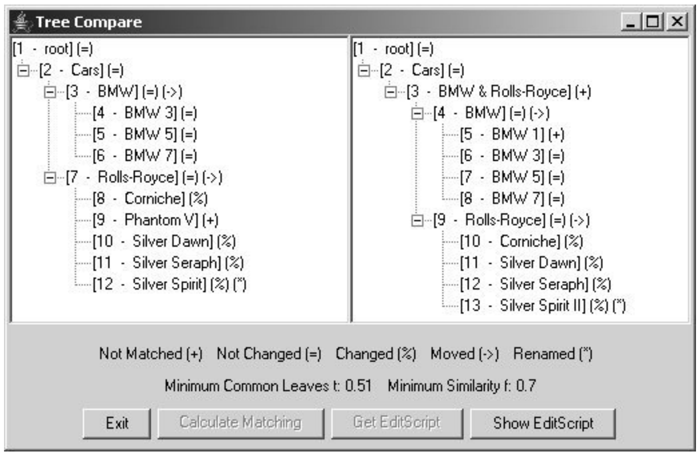
(a) Confirm renamings



Edit Script

```
1: REN(12, Silver Spirit II)
2: INS((13, BMW & Rolls-Royce, BMW & Rolls-Royce),2, null)
3: MOV(3, 13, null)
4: MOV(7, 13, 3)
5: INS((14, BMW 1, BMW 1),3, null)
6: UPD(8, Corniche)
7: UPD(10, Silver Dawn)
8: UPD(11, Silver Seraph)
9: UPD(12, Silver Spirit II)
10: DEL(9)
```

(b) Resulting EditScript



Tree Compare

```
[1 - root] (=)
  [2 - Cars] (=)
    [3 - BMW] (=) (->)
      [4 - BMW 3] (=)
      [5 - BMW 5] (=)
      [6 - BMW 7] (=)
    [7 - Rolls-Royce] (=) (->)
      [8 - Corniche] (%)
      [9 - Phantom V] (+)
      [10 - Silver Dawn] (%)
      [11 - Silver Seraph] (%)
      [12 - Silver Spirit] (%) (*)
```

```
[1 - root] (=)
  [2 - Cars] (=)
    [3 - BMW & Rolls-Royce] (+)
      [4 - BMW] (=) (->)
        [5 - BMW 1] (+)
        [6 - BMW 3] (=)
        [7 - BMW 5] (=)
        [8 - BMW 7] (=)
      [9 - Rolls-Royce] (=) (->)
        [10 - Corniche] (%)
        [11 - Silver Dawn] (%)
        [12 - Silver Seraph] (%)
        [13 - Silver Spirit II] (%) (*)
```

Not Matched (+) Not Changed (=) Changed (%) Moved (->) Renamed (*)

Minimum Common Leaves t: 0.51 Minimum Similarity f: 0.7

(c) After applying the comparison algorithm

Fig. 2: Outcomings of the Algorithm

warehouse. This means in particular, to have a representation of the changes (and implicit unchanged elements) between different versions of the multidimensional structure.

We adopted and extended a tree comparison algorithm to serve for this purpose. The output of this extended algorithm is an edit script consisting of elementary change operations. We contributed in particular a module for discovering renamings of nodes. Such an edit script facilitates the work of a data warehouse administrator who is in charge for representing structural changes. In a prototype implementation based on Hyperion Essbase we were able to demonstrate the validity of the approach, both the adequacy and validity of the algorithms and their scalability for real data warehouses.

References

1. Eder, J., Koncilia, C.: Changes of dimension data in temporal data warehouses. In: Proc. of 3rd Intl. Conf. on Data Warehousing and Knowledge Discovery 2001. (2001)

2. Eder, J., Koncilia, C., Mitsche, D.: Automatic Detection of Structural Changes in Data Warehouses. In: Proc. of the 5th Intl. Conf. on Data Warehousing and Knowledge Discovery 2003. (2003)
3. Kimball, R.: Slowly Changing Dimensions, Data Warehouse Architect. DBMS Magazine **9** (1996) URL: http://www.dbmsmag.com/.
4. Chamoni, P., Stock, S.: Temporal Structures in Data Warehousing. In: Proc. of the 1st Intl Conf. on Data Warehousing and Knowledge Discovery 1999. (1999)
5. Yang, J.: Temporal Data Warehousing. PhD thesis, Stanford University (2001)
6. Vaisman, A.: Updates, View Maintenance and Time Management in Multidimensional Databases. PhD thesis, Universidad de Buenos Aires (2001)
7. Blaschka, M.: FIESTA: A Framework for Schema Evolution in Multidimensional Information Systems. PhD thesis, Technische Universität München (2000)
8. Eder, J., Koncilia, C., Morzy, T.: The COMET Metamodel for Temporal Data Warehouses. In: Proc. of the 14th Intl. Conf. on Advanced Information Systems Engineering 2002. (2002)
9. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing **18** (1989) 1245–1262
10. Chawathe, S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proc. of the 1996 ACM SIGMOD. (1996)
11. Chawathe, S., Garcia-Molina, H.: Meaningful change detection in structured data. In: Proc. of the 1997 ACM SIGMOD. (1997)
12. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: Proc. of the 18th Intl. Conf. on Data Engineering. (2002)
13. Wang, Y., DeWitt, D., Cai, J.Y.: X-diff: An effective change detection algorithm for XML documents. In: Proc. of the 19th Intl. Conf. on Data Engineering. (2003)
14. Zhang, L.: On matching nodes between trees. Technical Report 2003–67, HP Labs (2003)