Repairing Dimension Hierarchies under Inconsistent Reclassification

Mónica Caniupán[1] and Alejandro Vaisman[2,3]

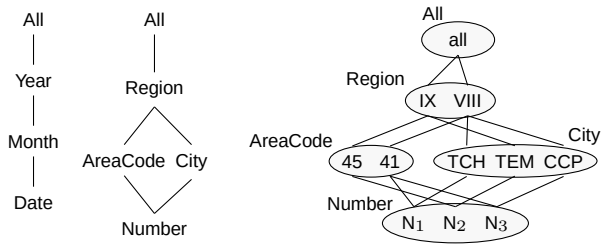
[1] Universidad del Bío-Bío, Chile
mcaniupa@ubiobio.cl
[2] Universidad de la República, Uruguay
avaisman@fing.edu.uy
[3] Université Libre de Bruxelles

Abstract. On-Line Analytical Processing (OLAP) dimensions are usually modelled as a hierarchical set of categories (the *dimension schema*), and *dimension instances*. The latter consist in a set of elements for each category, and relations between these elements (denoted *rollup*). To guarantee summarizability, a dimension is required to be *strict*, that is, every element of the dimension instance must have a unique ancestor in each of its ancestor categories. In practice, elements in a dimension instance are often reclassified, meaning that their rollups are changed (e.g., if the current available information is proved to be wrong). After this operation the dimension may become non-strict. To fix this problem, we propose to compute a set of *minimal r-repairs* for the new non-strict dimension. Each minimal *r-repair* is a *strict* dimension that keeps the result of the reclassification, and is obtained by performing a minimum number of insertions and deletions to the instance graph. We show that, although in the general case finding an *r-repair* is NP-complete, for real-world dimension schemas, computing such repairs can be done in polynomial time. We present algorithms for this, and discuss their computational complexity.

1 Introduction

Data Warehouses (DWs) integrate data from different sources, also keeping their history for analysis and decision support [1]. DWs represent data according to *dimensions* and *facts*. The former are modeled as hierarchies of sets of elements (called *dimension instances*), where each element belongs to a category from a hierarchy, or lattice of categories (called a *dimension schema*). Figure 1(a) shows the dimension schema of a *Phone Traffic DW* designed for an online Chilean phone call company, with dimensions Time and Phone (complete example in [2]). Figure 1(b) shows a dimension *instance* for the Phone schema. Here, TCH (Talcahuano), TEM (Temuco) and CCP (Concepción) are elements of the category City, and IX and VIII are elements of Region. The facts stored in the Phone Traffic DW correspond to the number of incoming and outgoing calls of a phone number at a given date. The fact table is shown in Figure 1(c).

To guarantee summarizability [3,4], a dimension must satisfy some constraints. First, it must be *strict*, that is, every element of a category should reach (i.e., roll-up to) no more that one element in each ancestor category (for example, the dimension instance in Figure 1 (b) is strict). Second, it must be *covering*, meaning that every member of a dimension level rolls-up to some element in another dimension level. Strict and



(a) Dimensions schemas (b) Phone dimension instance

Calls

Number	Date	In	Out
N_1	Jan 1,07	3	0
N_2	Jan 1,07	2	1
N_3	Jan 1,07	5	5
N_1	Jan 2,07	8	0
N_2	Jan 2,07	0	3
N_3	Jan 2,07	5	1

(c) Fact table

Fig. 1: The Phone Traffic DW (c.f [2])

covering dimensions allow to pre-compute aggregations, which can be used to compute aggregations at higher category levels, increasing query answering efficiency [5].

Dimensions can be updated to adapt to changes in data sources or modifications to the business rules [6,7], or even due to errors in the original data. Since strictness is not enforced in current commercial DW systems, after a dimension update is performed, a dimension instance may become non-strict. Considering that organizational DWs can contain terabytes of data [1], ensuring strictness of dimensions is crucial for efficient query answering. As an example, suppose that in Figure 1, the phone number N_2 was incorrectly classified as belonging to Temuco (TEM) while in fact it belonged to Concepcion (CCP). The DW administration must re-assign N_2 to the correct city, an operation we denote $Reclassification$. In the presence of strictness and covering constraints, assigning N_2 to Concepcion makes the dimension to become non-strict, since N_2 still has 45 as its area code. Thus, while summarizing fact data, we can reach region IX if we choose the path Number→AreaCode→ Region, or region VIII if the path is Number→City→Region. To fix this inconsistency, a solution could be to move N_2 to the area code 41. There are, however, situations where the solution is not obvious, as we show in this paper.

At least two approaches to the reclassification problem could be followed: (a) To prevent reclassification if we know that it can lead to a non-strict dimension [7]. We denote *this consistent reclassification*. (b) To allow any reclassification and *repair* the updated dimension if it becomes non-strict. Note that, in the general case, the repair may undo the reclassification, in particular if we are looking for a minimal repair (i.e., the repair 'closest' to the original dimension, cf. Section 3). In this paper we study approach (b), which captures the case when the user defines a reclassification about which he/she is absolutely sure. Therefore, the new rollup must be taken for certain, and a dimension instance, consistent with a set of constraints, must be produced by means of a (possibly minimal) repair that keeps the reclassification. We refer to these repairs as *r-repairs*. The r-repaired dimension instance is obtained from $\mathcal{D}$ by performing insertions and deletions of edges between elements. We show that in the general case, finding a *minimal r-repair* is NP-hard (Section 3). We also present an algorithm (Section 4) that obtains an *r-repair* in polynomial time for the class of dimensions that contain at most one conflicting level [6] (intuitively, a dimension that can lead to non-strict paths), and study complexity issues.

2 Background and Data Model

A dimension schema $\mathcal{S}$ consists of a pair $(\mathcal{C}, \nearrow)$, where $\mathcal{C}$ is a set of *categories*, and $\nearrow$ is a child/parent relation between categories. The dimension schema can be also represented with a directed acyclic graph where the vertices correspond to the categories and the edges to the child/parent relation. The transitive and reflexive closure of $\nearrow$ is denoted by $\nearrow^*$. *There are no shortcuts in the schemas*, that is, if $\mathsf{c}_i \nearrow \mathsf{c}_j$ there is no category c_k such that $\mathsf{c}_i \nearrow^* \mathsf{c}_k$ and $\mathsf{c}_k \nearrow^* \mathsf{c}_j$. Every dimension schema contains a distinguished top category called All which is reachable from all other categories, i.e. for every $\mathsf{c} \in \mathcal{C}$, $\mathsf{c} \nearrow^* \mathsf{All}$. The leaf categories are called bottom categories.[4]

A dimension instance $\mathcal{D}$ over a dimension schema $\mathcal{S} = (\mathcal{C}, \nearrow)$ is a tuple $(\mathcal{M}, <)$, such that: (i) $\mathcal{M}$ is a finite collection of ground atoms of the form $\mathsf{c}(\mathsf{a})$ where $\mathsf{c} \in \mathcal{C}$ and a is a constant. If $\mathsf{c}(\mathsf{a}) \in \mathcal{M}$, a is said to be an element of c. The constant all is the only element in category All. Categories are assumed to be disjoint, i.e. if $c_i(a), c_j(a) \in \mathcal{M}$ then $i = j$. There is a function Cat that maps elements to categories so that $\mathsf{Cat}(\mathsf{a}) = \mathsf{c}_i$ if and only if $\mathsf{c}_i(\mathsf{a}) \in \mathcal{M}$. (ii) The relation $<$ contains the child/parent relationships between elements of different categories, and is compatible with $\nearrow$: If $\mathsf{a} < \mathsf{b}$, then $\mathsf{Cat}(\mathsf{a}) \nearrow \mathsf{Cat}(\mathsf{b})$. We denote $<^*$ the reflexive and transitive closure of $<$.

In what follows we use the term dimension to refer to dimension instances. For each pair of categories c_i and c_j such that $\mathsf{c}_i \nearrow \mathsf{c}_j$ there is a *rollup* relation denoted $\mathcal{R}_{\mathcal{D}}(\mathsf{c}_i, \mathsf{c}_j)$ that consists of the following set of pairs $\{(\mathsf{a}, \mathsf{b}) \mid \mathsf{Cat}(\mathsf{a}) = \mathsf{c}_i, \mathsf{Cat}(\mathsf{b}) = \mathsf{c}_j \text{ and } \mathsf{a} <^* \mathsf{b}\}$. We next define two kinds of constraints a dimension should satisfy.

Definition 1. [Strictness and Covering Constraints [8]] A *strictness constraint* over a hierarchy schema $\mathcal{S} = (\mathcal{C}, \nearrow)$ is an expression of the form $\mathsf{c}_i \to \mathsf{c}_j$ where $\mathsf{c}_i, \mathsf{c}_j \in \mathcal{C}$ and $\mathsf{c}_i \nearrow^* \mathsf{c}_j$. The dimension $\mathcal{D}$ *satisfies* the strictness constraint $\mathsf{c}_i \to \mathsf{c}_j$ if and only if the rollup relation $\mathcal{R}_{\mathcal{D}}(\mathsf{c}_i, \mathsf{c}_j)$ is strict. A *covering constraint* over $\mathcal{S}$ is an expression of the form $\mathsf{c}_i \Rightarrow \mathsf{c}_j$ where $\mathsf{c}_i, \mathsf{c}_j \in \mathcal{C}$ and $\mathsf{c}_i \nearrow^* \mathsf{c}_j$. The dimension $\mathcal{D}$ *satisfies* the covering constraint $\mathsf{c}_i \Rightarrow \mathsf{c}_j$ if and only if the rollup relation $\mathcal{R}_{\mathcal{D}}(\mathsf{c}_i, \mathsf{c}_j)$ is covering. $\Sigma_s(\mathcal{S})$ and $\Sigma_c(\mathcal{S})$ denote, respectively, the set of all possible strictness and covering constraints over hierarchy schema $\mathcal{S}$. □

A dimension $\mathcal{D}$ over an schema $\mathcal{S}$ is *inconsistent* if it fails to satisfy $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$. A dimension $\mathcal{D}$ is said to be *strict* (*covering*) if all of its rollup relations are strict (covering) [3]. Otherwise, the dimension is said to be *non-strict* (*non-covering*). The dimension in Figure 1(b) is both covering and strict. Thus, a query asking for the number of calls grouped by Region can be computed by using pre-computed aggregations at categories AreaCode or City. This is not the case of dimension $\mathcal{D}$ in Figure 2, which is covering, but not strict (N_3 rolls-up to both IX and VIII in category Region).

Most research and industrial applications of DWs assume strict and covering dimensions [1,9,10]. For these kinds of dimensions, summarization operations between two categories that are connected in the schema are always correct [3]. In practice, dimensions may be non-strict and non-covering [11,12], which can lead to incorrect results when summarizing data. We can prevent these errors specifying integrity constraints for rollup relations [11].

[4] To simplify the presentation and without loss of generality, we assume that categories do not have attributes, and schemas have a unique bottom category.

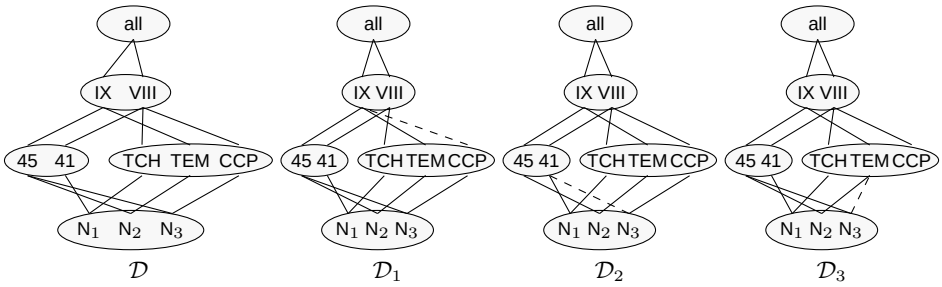


Fig. 2: (Non-strict dimension instance (category names are omitted)($\mathcal{D}$); Repairs of the dimension (dashed edges were inserted to restore strictness) ($\mathcal{D}_1$-$\mathcal{D}_3$)

3 Inconsistent Reclassification and r-repairs

The reclassify operator presented in [7] is not defined for every possible dimension. It allows to reclassify edges of dimensions only if the resulting dimension satisfies $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$. When the dimension contains a *conflicting level* (Definition 2), a reclassification may leave the dimension inconsistent with respect to strictness, and therefore the operation is forbidden.

Definition 2. [Conflicting Levels (cf. [13])] Given a dimension instance $\mathcal{D} = (\mathcal{M}, <)$ over a schema $\mathcal{S} = (\mathcal{C}, \nearrow)$, a pair of categories c_i and c_j such that $\mathsf{c}_i \nearrow \mathsf{c}_j$, a pair of elements a, b with $\mathsf{Cat}(\mathsf{a}) = \mathsf{c}_i$ and $\mathsf{Cat}(\mathsf{b}) = \mathsf{c}_j$. A category c_k such that $\mathsf{c}_j \nearrow^* \mathsf{c}_k$ is *conflicting* with respect to reclassification, if there exists a category c_m such that $\mathsf{c}_m \nearrow^* \mathsf{c}_i$, there is an alternative path between c_m and c_k not including the edge (c_i,c_j), and a is reached by at least one element in c_m. □

As an illustration, category Region in Figure 1(a) is conflicting with respect to reclassification from AreaCode to Region or a reclassification from City to Region. Consider a reclassification from AreaCode to Region, in this case, c_i = AreaCode, $\mathsf{c}_j = \mathsf{c}_k$ = Region, and c_m = Number. The edges (Number,City) and (City,Region) form an alternative path from Number to Region not including edge (AreaCode,Region).

In practice, preventing reclassification is not an admissible solution (eg., if the operation is applied to correct erroneous data). Therefore, non-strict dimensions resulting from the operation must be allowed. Definition 3 captures this semantics.

Definition 3. [Reclassify] Given a dimension instance $\mathcal{D} = (\mathcal{M}, <)$ over a dimension schema $\mathcal{S} = (\mathcal{C}, \nearrow)$, a pair of categories c_i and c_j such that $\mathsf{c}_i \nearrow \mathsf{c}_j$, a pair of elements a, b with $\mathsf{Cat}(\mathsf{a}) = \mathsf{c}_i$ and $\mathsf{Cat}(\mathsf{b}) = \mathsf{c}_j$, operator $Reclassify(\mathcal{D}, \mathsf{c}_i, \mathsf{c}_j, \mathsf{a}, \mathsf{b})$ returns a new dimension instance $\mathcal{D}_u = (\mathcal{M}_u, <_u)$ with $\mathcal{M}_u = \mathcal{M}$, and $<_u = (< \setminus \{(\mathsf{a}, \mathsf{c}) \mid (\mathsf{a}, \mathsf{c}) \in <$ and $\mathsf{Cat}(c) = C_j\}) \cup \{(\mathsf{a},\mathsf{b})\}$. □

For example, the result of applying $Reclassify(\mathcal{D}, \mathsf{Number}, \mathsf{AreaCode}, \mathsf{N}_3, 45)$ on dimension $\mathcal{D}$ in Figure 1(b) is the non-strict dimension $\mathcal{D}$ in Figure 2. Since we *know* that the area code for N_3 is 45, we *must* perform the reclassification and, in a subsequent step, repair the updated dimension in order to comply with the strictness constraint.

Thus, if a reclassification according to Definition 3 yields a non-strict dimension, we must *repair* the dimension [8], i.e., perform the necessary changes in order to obtain dimension satisfying the constraints. In particular, from all possible repairs we are

interested in finding the *minimal* one. Intuitively, a minimal repair of a dimension is a new dimension that satisfies a given set of constraints, and is obtained by applying a minimum number of insertions and deletions to the original rollup relations. Although techniques to compute repairs with respect to a set of constraints are well-known in the field of relational databases [14], they cannot be applied in a DW setting, since it is not trivial to represent strictness or covering constraints using relational constraints like functional dependencies [8].

Defining a minimal repair requires a notion of *distance* between dimensions: Let $\mathcal{D} = (\mathcal{M}, <_{\mathcal{D}})$ and $\mathcal{D}' = (\mathcal{M}, <_{\mathcal{D}'})$ be two dimensions over the same schema $\mathcal{S} = (\mathcal{C}, \nearrow)$. The distance between $\mathcal{D}$ and $\mathcal{D}'$ is defined as $dist(\mathcal{D}, \mathcal{D}') = |(<_{\mathcal{D}'} \setminus <_{\mathcal{D}}) \cup (<_{\mathcal{D}} \setminus <_{\mathcal{D}'})|$, i.e. the cardinality of the symmetric difference between the two roll-up relations [8]. Based on this, the definition of repair is as follows [8]: Given a dimension $\mathcal{D} = (\mathcal{M}, <)$ over a schema $\mathcal{S} = (\mathcal{C}, \nearrow)$, and a set of constraints $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$, a repair of $\mathcal{D}$ with respect to Σ is a dimension $\mathcal{D}' = (\mathcal{M}', <')$ over $\mathcal{S}$, such that $\mathcal{D}'$ satisfies Σ, and $\mathcal{M}' = \mathcal{M}$. A *minimal repair* of $\mathcal{D}$ is a repair $\mathcal{D}'$ such that $dist(\mathcal{D}, \mathcal{D}')$ is minimal among all the repairs of $\mathcal{D}$.

Minimal repairs according to [8] may result in dimensions where the reclassification is undone. If we assume that a reclassification always represents information that is certain, this semantics is not appropriate. For example, the minimal repairs (as defined in [8]) for dimension $\mathcal{D}$ in Figure 2 are dimensions $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$. All of them were obtained from $\mathcal{D}$ by performing insertions and/or deletions of edges. For example, $\mathcal{D}_1$ is generated by deleting edge (CCP,VIII) and inserting (CCP,IX). The distances between the repairs and $\mathcal{D}$ are: (a) $dist(\mathcal{D}, \mathcal{D}_1) = |$(CCP,IX), (CCP,VIII)$| = 2$; (b) $dist(\mathcal{D}, \mathcal{D}_2) =$ |(N_3,41), (N_3,45)$| = 2$; (c) $dist(\mathcal{D}, \mathcal{D}_3) =$ |(N_3,TEM), (N_3,CCP)$| = 2$. Note that dimension $\mathcal{D}_2$ has undone the reclassification, and should not be considered an appropriate repair.

In light of the above, we next study the problem of finding repairs that do not undo the reclassification. We denote these repairs *r-repairs*.

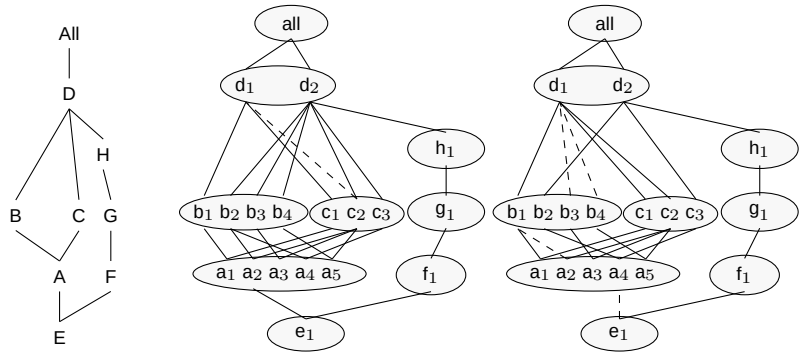
3.1 r-repairs

Given a dimension $\mathcal{D}$, a set $\mathcal{R}$ of reclassify operations is called *valid* if the following holds: for every $Reclassify(\mathcal{D}, \mathsf{c}_i, \mathsf{c}_j, \mathsf{a}, \mathsf{b}) \in \mathcal{R}$ there is no $Reclassify(\mathcal{D}, \mathsf{c}_i, \mathsf{c}_j, \mathsf{a}, \mathsf{c}) \in \mathcal{R}$ with $\mathsf{c} \neq \mathsf{b}$. In what follows we consider only valid reclassification.

Definition 4. [r-repairs] Given a dimension $\mathcal{D} = (\mathcal{M}, <)$ defined over a schema $\mathcal{S} = (\mathcal{C}, \nearrow)$ that is consistent with respect to $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$, and a valid set of reclassifications $\mathcal{R}$, an *r-repair* for $\mathcal{D}$ under $\mathcal{R}$ is a dimension $\mathcal{D}' = (\mathcal{M}', <')$ defined over $\mathcal{S}$ such that: (i) $\mathcal{D}'$ satisfies Σ; (ii) $\mathcal{M}' = \mathcal{M}$; and (iii) for every $Reclassify(\mathcal{D}, \mathsf{c}_i, \mathsf{c}_j, \mathsf{a}, \mathsf{b}) \in \mathcal{R}$, the pair $(\mathsf{a}, \mathsf{b}) \in <'$. Let $r\text{-}repair(\mathcal{D}, \mathcal{R})$ be the set of *r-repairs* for $\mathcal{D}$ under $\mathcal{R}$. A *minimal r-repair* $\mathcal{D}'$ for a dimension $\mathcal{D}$ and a set of reclassifications $\mathcal{R}$ is an *r-repair* such that $dist(\mathcal{D}, \mathcal{D}')$ is minimal among all the *r-repairs* in $r\text{-}repair(\mathcal{D}, \mathcal{R})$. □

Note that the distance between a dimension $\mathcal{D}$ and an *r-repair* $\mathcal{D}'$ is bounded by the size of $\mathcal{D}$. For the dimension in Figure 1(b) and the set $\mathcal{R} = \{Reclassify(\mathcal{D},$ Number, AreaCode, N_3, 45$)\}$, $r\text{-}repair(\mathcal{D}, \mathcal{R})$ contains dimensions $\mathcal{D}_1$ and $\mathcal{D}_3$ in Figure 2.

The existence of *r-repairs* cannot be guaranteed if the reclassification set contains more than one reclassify operation. However, a positive result can be obtained for a set $\mathcal{R}$ containing only one reclassification.



(a) Dimension schema (b) Dimension $\mathcal{D}$ (c) *r-repair* of $\mathcal{D}$

Fig. 3: Heuristics

Theorem 1. Let $\mathcal{D}$ be a dimension instance over a schema $\mathcal{S}$ with constraints $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$, and a reclassification set $\mathcal{R}$ applied over $\mathcal{D}$, producing a new dimension $\mathcal{D}_u$; the problem of deciding if there exists an r-repair $\mathcal{D}'$ of $\mathcal{D}_u$ with respect to Σ, such that $dist(\mathcal{D}_u, \mathcal{D}') \leq k$ is NP-complete. □

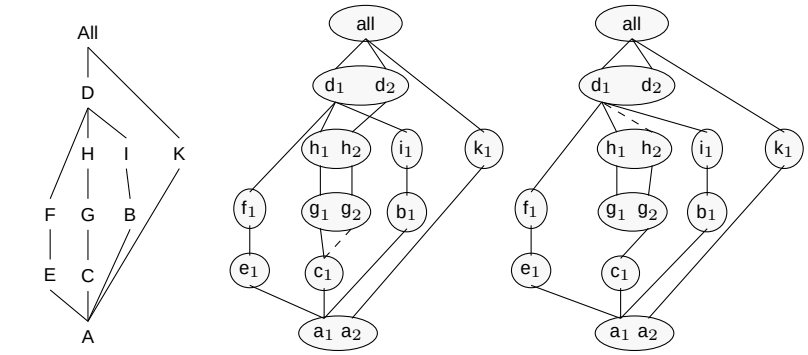
Theorem 2. Given a dimension $\mathcal{D}$ over a schema $\mathcal{S}$ with constraints $\Sigma = \Sigma_s(\mathcal{S}) \cup \Sigma_c(\mathcal{S})$, and a set of reclassification $\mathcal{R}$, the problem of deciding if $\mathcal{D}'$ is a minimal *r-repair* of $\mathcal{D}$ with respect to Σ is co-NP-complete. □

The proof of Theorem 1 proceeds by reduction from the set covering problem. Theorem 2 follows from considering the complement of the problem, that is, deciding wether or not the *r-repair* is minimal with respect to Σ and $\mathcal{R}$ which, from Theorem 1 is NP-complete.

4 Algorithms for *r-repairs*

We study *r-repairs* for dimensions containing *at most one conflicting level* (Definition 2) that becomes inconsistent with respect to strictness after a reclassify operation. Solving this problem efficiently we cover a wide range of real-life situations [9,10]. We present two heuristics and algorithms that find an *r-repair* for a dimension under reclassification in polynomial time. These heuristics are such that the distance between the *r-repair* obtained and the minimal *r-repair* is bound. In what follows we assume that after a reclassification a dimension becomes inconsistent with respect to strictness. Thus, we can identify the paths that make the dimension non-strict. We denote these paths *non-strict*.

The first heuristics we propose ensures that when choosing a repair operation *we do not generate new non-strict paths*. Let us consider the dimension schema and instance in Figure 3 (a)-(b). Element c_2 in category C is reclassified from d_2 to d_1 in category D as indicated in Figure 3(b) (dashed edge). The elements incident to c_2 are a_2, a_3, and a_5. The non-strict paths are: (i) $a_2 \rightarrow b_2 \rightarrow d_2$, $a_2 \rightarrow c_2 \rightarrow d_1$. (ii) $a_3 \rightarrow b_3 \rightarrow d_2$, $a_3 \rightarrow c_2 \rightarrow d_1$. (iii) $a_5 \rightarrow b_4 \rightarrow d_2$, $a_5 \rightarrow c_2 \rightarrow d_1$. Elements b_3 and b_4 in category B have no incident edges other than a_3 and a_5, respectively. Thus, a possible repair



(a) Dimension schema (b) Dimension $\mathcal{D}$ (c) *r-repair* of $\mathcal{D}$

Fig. 4: The dimension schema, instance and *r*-repair for the Algorithm

operation would delete edges (b_3, d_2) and (b_4, d_2), and insert edges (b_3, d_1) and (b_4, d_1) (dashed edges in Figure 3(c)). This repair operation *does not add new non-strict paths*. Conversely, b_2 has two incident edges (from a_2 and a_4), and a_4 rolls-up to d_2 via c_3; therefore, deleting (b_2, d_2) and inserting (b_2, d_1), would produce a violation of strictness, and the following new non-strict paths: $a_4 \rightarrow b_2 \rightarrow d_1$, $a_4 \rightarrow c_3 \rightarrow d_2$. A solution would be a repair that changes the parent of a_2 either to b_1 (as depicted in Figure 3 (c)), to b_3, or to b_4. In addition, all edges incident to a_2 must also be reclassified if they reach d_2 through a path not including the edge (A,B) in the dimension hierarchy. This is the case of e_1 that can be repaired by deleting edge (h_1, d_2) and inserting (h_1, d_1) or by moving e_1 to any element a_i reaching d_2, like a_4, as we show in Figure 3 (c).

The second heuristics is aimed at guaranteeing that at each step the algorithm chooses a repair operation that, accomplishing the "no new conflicts" heuristics, requires the least number of changes.

4.1 Computing the *r-repairs*

We illustrate the algorithms with the dimension schema and instance in Figure 4(a)-(b). The first algorithm *search_path()* reads the dimension schema from a table storing the child/parent relation between categories, obtains the conflicting level (CL) and stored it in variable *cat_cl*, and verifies that the schema has a unique such CL. It also computes the category from where the paths reaching the CL start, and stores it in the variable *cat_bottom*. For instance, for the child/parent relation of the categories in Figure 4(a), variable *cat_cl=D* since D is the CL, and the *cat_bottom* = *A*. Then, the algorithm computes the paths of categories that reach the CL, and stores them in the structure *cat_list_paths*. Any update operation involving categories in this structure may leave the dimension schema non-strict. For the schema in Figure 4, *cat_list_paths* contains: [1]: A $\rightarrow$ E $\rightarrow$ F $\rightarrow$ D. [2]: A $\rightarrow$ C $\rightarrow$ G $\rightarrow$ H $\rightarrow$ D. [3]: A $\rightarrow$ B $\rightarrow$ I $\rightarrow$ D. Then, Algorithm *search_repair*() in Figure 5 applies repair operations over categories in this structure.

Algorithm *search_repair*() (Figure 5) first verifies if an update operation leaves the dimension instance non-strict. For this, the list *list_inconsistent_paths* containing the non-strict paths is produced (line 2). If the dimension is non-strict, the algorithm re-

stores consistency of every non-strict path on the list. Let $Reclassify(\mathcal{D}, \mathtt{c}_1, \mathtt{c}_2, \mathtt{e}_u, \mathtt{p}_u)$ be the update operation that affects categories in *cat_list_paths* and leaves the dimension non-strict. The algorithm obtains, for every *cat_bottom* element in the inconsistent list, the number of paths reaching new_CL (new parent of $\mathtt{e}_u$ in CL) and old_CL (old parent in CL) (Lines 5 to 10). If these numbers are equal, it means that there are only two alternative paths for the corresponding *cat_bottom* element, and the algorithm tries to keep new_CL as the ancestor to these paths in the CL (lines 12 to 21). If not, it means that there are more paths reaching old_CL, and the algorithm tries to update the edge reaching new_CL, since this produces less changes (lines 25 to 26). If not, the algorithm assigns new_CL to all the paths reaching old_CL (lines 28 to 31).

As an illustration, consider the reclassification $Reclassify(\mathcal{D}, \mathtt{c}, \mathtt{G}, \mathtt{c}_1, \mathtt{g}_2)$ applied over the dimension in Figure 4(b). The reclassification affects to the bottom element $\mathtt{a}_1$ and therefore *list_inconsistent_paths* contains the paths: [1]: $\mathtt{a}_1 \rightarrow \mathtt{e}_1 \rightarrow \mathtt{f}_1 \rightarrow \mathtt{d}_1$. [2]: $\mathtt{a}_1 \rightarrow \mathtt{c}_1 \rightarrow \mathtt{g}_2 \rightarrow \mathtt{h}_2 \rightarrow \mathtt{d}_2$. [3]: $\mathtt{a}_1 \rightarrow \mathtt{b}_1 \rightarrow \mathtt{i}_1 \rightarrow \mathtt{d}_1$. The old and new parent in CL for $\mathtt{a}_1$ are: $old\_CL = \mathtt{d}_1$, $new\_CL = \mathtt{d}_2$, and the number of paths reaching $\mathtt{d}_1$ and $\mathtt{d}_2$ are, respectively, 2 and 1. Since there are more paths reaching the old parent in CL, the algorithm tries to keep $\mathtt{d}_1$ as the ancestor in $\mathtt{D}$ for all the conflicting paths. This operation is possible given that element $\mathtt{h}_2$ does not have other child different from $\mathtt{g}_2$, and also the update is not performed over $\mathtt{h}_2$ (validations performed by function *check_change_to_new_CL*); thus, the algorithm deletes edge ($\mathtt{h}_2$, $\mathtt{d}_2$) and inserts ($\mathtt{h}_2$, $\mathtt{d}_1$) (function *change_new_CL*), producing the repair shown in Figure 4(c).

Proposition 1. Given a dimension $\mathcal{D}$ over an schema $\mathcal{S}$, and a set of reclassify operations $\mathcal{R}$ of size 1. (a) Algorithm *search_repair*() terminates in a finite number of steps. (b) Algorithm *search_repair*() finds an *r-repair* for dimension $\mathcal{D}$.

4.2 Complexity Analysis

Algorithm *search_path()*, that captures the hierarchy schema of a dimension, runs in $O(k)$ with k being the number of paths reaching the conflicting level and starting at the bottom category. For Algorithm *search_repair*(), the most expensive function is *check_Consistency*(), that finds out if the dimension instance becomes inconsistent after an update, and, in that case, generates the list of inconsistent paths. It needs to verify that every element at the bottom category reaches a unique element in the CL after an update. Let n be the number of elements at the bottom category, m the longest path from the bottom category to the CL, and k the number of alternative paths from the bottom category to the CL. Note that m, k and n are all independent values. Then, the function has to search k paths for n elements at the bottom category. Thus, the algorithm runs in $O(n * m * k)$ in a worst case scenario, which implies that all elements in the bottom category are in conflict after an update, which is quite unlikely. Consider also that, in general, the number of categories between the bottom and the CL category is small, as well as the number of alternative paths to the CL category. More than often the hierarchy schema is a tree (i.e., there is no CL), and in this case the algorithm runs in $O(n * \log\ m * k)$ (the longest path can be computed in $\log m$). The rest of the functions in the algorithm run in lineal time using the list of inconsistent paths, the list of categories in the hierarchy schema, and the rollup functions.

search_repair ()

```
Structure paths{String element, String category, *next, *below};
paths list_inconsistent_paths = NULL;
String new_CL, old_CL, e_u, p_u, parent_child_CL, child_CL1, child_CL2;
Int cost=0, cont_same_elements;
1: if check_Consitency() = 0 then
2:     list_inconsistent_paths= non_strict_paths();
3:     while (list_inconsistent_paths.below ≠ NULL) do
4:         i=0;
5:         cont_same_elements = find_number_paths(list_inconsistent_paths(i));
6:         new_CL = find_new_parent_CL(list_inconsistent_paths(i),e_u);
7:         old_CL = find_old_parent_CL(list_inconsistent_paths(i),new_CL);
8:         {the parents in the CL before and after the update};
9:         cont_1 =number_paths_reaching_element(list_inconsistent_paths(i),new_CL);
10:        cont_2 =number_paths_reaching_element(list_inconsistent_paths(i),old_CL);
11:        if (cont_1 = cont_2) then
12:            {Same # of paths reaching the old and new parent in CL, → try to keep the new parent};
13:            child_CL1 = find_child_CL(list_inconsistent_paths(i),old_CL);
14:            child_CL2 = find_child_CL(list_inconsistent_paths(i),new_CL);
15:            {it captures the element in the category that reach the old(new) parent in CL};
16:            if (check_change_to_new_CL(child_CL1)=1) then
17:                {It is possible to change to the new parent in CL};
18:                cost = cost + change_new_CL(list_inconsistent_paths(i),child_CL1, new_CL);
19:            else
20:                cost = cost + change_old_CL(list_inconsistent_paths(i),child_CL2, old_CL);
21:            end if
22:        else
23:            {# of paths reaching the old parent in CL is greater than the # of paths reaching the new parent in CL, → try
               to keep the old parent (second heuristics)};
24:            child_CL2 = find_child_CL(list_inconsistent_paths(i),new_CL);
25:            if (check_change_to_old_CL(child_CL2)=1) then
26:                cost = cost + change_old_CL(list_inconsistent_paths(i),child_CL2, old_CL);
27:            else
28:                for j = 1 TO cont_2 do
29:                    child_CL1 = find_child_CL(list_inconsistent_paths(i),old_CL);
30:                    cost = cost + change_new_CL(list_inconsistent_paths(i),child_CL1, new_CL);
31:                end for
32:            end if
33:        end if
34:        i = i + cont_same_elements;
35:        move(list_inconsistent_paths,i);
36:    end while
37: end if
```

Fig. 5: Generating an *r-repair*

5 Discussion and Conclusion

Most efforts addressing inconsistency issues in DWs focus in solving inconsistencies between operational and warehouse data [15,16,17]. No much work has been devoted to study the inconsistencies that may arise when dimension updates are applied. This is due probably to the fact that dimensions were assumed to be static. Hurtado et al. showed that dimensions need to be updated when, for instance, changes in the business rules that lead to the warehouse design occur, or data in the operational sources are updated [7,6]. In these works, dimension updates guarantee that the dimensions remain consistent after the updating operations are applied (if there is a risk of inconsistency, updates are prevented). A similar approach is adopted in [18]. Other approaches to dimension updates accept that the changes may leave the updated dimension inconsistent. Therefore, repairing techniques must be applied in order to guarantee summarizability

[4], which is crucial for OLAP operations. Pedersen et al. [12] presented a first approach to this problem, transforming non-strict into strict dimensions by means of insertion of artificial elements. Caniupán et al. present a logic programming approach to repair dimensions that are inconsistent with respect to a set of constraints [8,19]. Although important to gain insight into the problem of repairing inconsistent dimensions, and containing some interesting theoretical results, from a practical point of view, the approach presented in [8,19] would be computationally expensive in real-world cases. Besides, DW administrators and developers are not acquainted with logic programs. Moreover, for the specific case of reclassification, the work in [8,19] only deal with repairs that may undo the update. On the contrary, the *r-repairs* we present in this paper do not undo the reclassification. Finally, the minimal repairs obtained in [8,19] could lead to rollup functions that do not make sense in the real world (e.g., relating dimension members that are not actually related in any way). Following a different approach, we propose efficient algorithms that lead to consistent dimensions (although not necessarily minimal with respect to the distance function), and where undesired solutions could be prevented.

We have shown that, in general, finding *r-repairs* for dimension instances is NP-complete. However, we also showed that in practice, computing *r-repairs* can be done in polynomial time when the set of updates contains only one reclassification, and the dimension schema has at most one conflicting level. We have explored algorithms to compute *r-repairs* for this class of dimension schemas, and discussed their computational complexity, being in a worst case scenario of order $O(n * m * k)$, where the key term is n, the number of elements in the bottom level affected by the inconsistencies. We would like to remark the fact that in the algorithms presented in this paper, for the sake of generality, we did not include the possibility of preventing rollups that could make no sense in practice. However, it is straightforward to enhance the *search-repair* algorithm to consider only repairs that are acceptable by the user. At least two approaches can be followed here: to prioritize the rollup functions (as, for example, is proposed in [20]), or even to define some rollups to be fixed (and therefore, not allowed to be changed). Of course, in the latter case, it may be the case where a minimal *r-repair* does not exist. We leave this discussion as future work, as well as the experimentation of the algorithms in real-world data warehouses.

Acknowledgements: This project was partially funded by FONDECYT, Chile grant number 11070186. Part of this research was done during visit of Alejandro Vaisman to University del Bío-Bío in 2010. Currently, Mónica Caniupán is funded by DIUBB 110115 2/R. A. Vaisman has been partially funded by LACCIR project LACR-FJR-R1210LAC004.

References

1. Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record **26** (1997) 65–74
2. Bertossi, L., Bravo, L., Caniupán, M.: Consistent query answering in data warehouses. In: AMW. (2009)
3. Hurtado, C., Gutierrez, C., Mendelzon, A.: Capturing Summarizability with Integrity Constraints in OLAP. ACM Transacations on Database Systems **30** (2005) 854–886
4. Lenz, H., Shoshani, A.: Summarizability in OLAP and Statistical Data Bases. In: SSDBM. (1997) 132–143

5. Rafanelli, M., Shoshani, A.: STORM: a Statistical Object Representation Model. In: SSDBM. (1990) 14–29
6. Hurtado, C., Mendelzon, A., Vaisman, A.: Maintaining Data Cubes under Dimension Updates. In: ICDE. (1999) 346–355
7. Hurtado, C., Mendelzon, A., Vaisman, A.: Updating OLAP Dimensions. In: DOLAP. (1999) 60–66
8. Caniupán, M., Bravo, L., Hurtado, C.: A logic programming approach for repairing inconsistent dimensions in data warehouses. Submitted to Data and Knowledge Engineering (2010)
9. Dodge, G., Gorman, T.: Essential Oracle8i Data Warehousing: Designing, Building, and Managing Oracle Data Warehouses (with Website). John Wiley & Sons, Inc. (2000)
10. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. John Wiley & Sons, Inc. (2002)
11. Hurtado, C., Mendelzon, A.: Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In: ICDT. (2001) 375–389
12. Pedersen, T., Jensen, C., Dyreson, C.: Extending Practical Pre-Aggregation in On-Line Analytical Processing. In: VLDB. (1999) 663–674
13. Vaisman, A.: Updates, View Maintenance and Materialized Views in Multidimnensional Databases. PhD thesis, Universidad de Buenos Aires (2001)
14. Bertossi, L.: Consistent query answering in databases. ACM Sigmod Record **35** (2006) 68–76
15. Zhuge, Y., Garcia-Molina, H., Wiener, J.L.: Multiple View Consistency for Data Warehousing. In: ICDE. (1997) 289–300
16. Gupta, H., Mumick, I.S.: Selection of Views to Materialize Under a Maintenance Cost Constraint. In: ICDT. (1999) 453–470
17. Schlesinger, L., Lehner, W.: Extending Data Warehouses by Semiconsistent Views. In: DMDW. (2002) 43–51
18. Letz, C., Henn, E.T., Vossen, G.: Consistency in Data Warehouse Dimensions. In: IDEAS. (2002) 224–232
19. Bravo, L., Caniupán, M., Hurtado, C.: Logic programs for repairing inconsistent dimensions in data warehouses. In: AMW. (2010)
20. Espil, M.M., Vaisman, A., Terribile, L.: Revising data cubes with exceptions: a rule-based perspective. In: DMDW. (2002) 72–81