Classification of Contradiction Patterns

Heiko Müller, Ulf Leser, and Johann-Christoph Freytag

Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany,
{hmueller, leser, freytag}@informatik.hu-berlin.de

Abstract. Solving conflicts between overlapping databases requires an understanding of the reasons that lead to the inconsistencies. Provided that conflicts do not occur randomly but follow certain regularities, patterns in the form of "IF *condition* THEN *conflict*" provide a valuable means to facilitate their understanding. In previous work, we adopt existing association rule mining algorithms to identify such patterns. Within this paper we discuss extensions to our initial approach aimed at identifying possible update operations that caused the conflicts between the databases. This is done by restricting the items used for pattern mining. We further propose a classification of patterns based on mappings between the contradicting values to represent special cases of conflict generating updates.

1 Conflicts in Overlapping Databases

Many databases exist with overlaps in their sets of represented real-world entities. There are different reasons for these overlaps, like:

- *replication of data sources* at different sites to improve the performance of web-services and the availability of the data,
- *independent production of data* representing a common set of entities or individuals by different groups or institutions, and
- *data integration* where data is copied from sources, possibly transformed and manipulated for data cleansing, and stored in an integrated data warehouse.

Whenever overlapping data is administered at different sites, there is a high probability of the occurrence of differences. Many of these inconsistencies are systematic, caused by the usage of different controlled vocabularies, different measurement units, different data modifications for data cleansing, or by consistent bias in experimental data analysis. When producing a consistent view of the data knowledge about such systematic deviations can be used to assess the individual quality of database copies for conflict resolution.

Assuming that conflicts do not occur randomly but follow specific (but unknown) regularities, patterns of the form "IF *condition* THEN *conflict*" provide a valuable means to facilitate the identification and understanding of systematic deviations. In Müller et al. (2004) we proposed the adaptation of existing data mining algorithms to find such contradiction patterns. Evaluated by a domain expert, these patterns can be utilized to assess the correctness of conflicting values and therefore for conflict resolution.

Within this paper we present a modified approach for mining contradictory data aimed at enhancing the expressiveness of the identified patterns. This approach is based on the assumption that conflicts result from modification of databases that initially where equal (see Figure 1). Conflicting values are introduced by applying different sequences of update operations, representing for example different data cleansing activities, to a common ancestor database. Given a pair of contradicting databases, each resulting from a different update sequence, we reproduce conflict generation to assist a domain expert in conflict resolution. We present an algorithm for identifying update operations that describe in retrospective the emergence of conflicts between the databases. These conflict generators are a special class of contradiction patterns. We further classify the patterns based on the mapping between contradicting values that they define. Each such class corresponds to a special type of conflict generating update operation. The classification further enhances the ability for pruning irrelevant patterns in the algorithm.

The reminder of this paper is structured as follows: In Section 2 we define conflict generators for databases pairs. Section 3 presents an algorithm for finding such conflict generators. We discuss related work in Section 4 and conclude in Section 5.

2 Reproducing Conflict Generation

Databases r_1 and r_2 from Figure 1 contain fictitious results of different research groups investigating a common set of individual owls. Identification of tuples representing the same individual is accomplished by the unique object identifier ID. The problem of assigning these object identifiers is not considered within this paper, i.e., we assume a preceding duplicate detection step (see for example Hernandez and Stolfo (1995)). Note that we are only interested in finding update operations that introduce conflicts between the overlapping parts of databases. Therefore, we also assume that all databases have equal sets of object identifiers.

Conflicting values are highlighted in Figure 1 and conflicts are systematic. The conflicts in attribute SPECIES are caused by the different usage of English and Latin vocabularies to denote species names, conflicts in attribute COLOR are due to a finer grained color description for male and female snowy owls (*Nyctea Scandica*) in database r_2, and the conflicts within attribute SIZE are caused by rounding or truncation errors for different species in database r_1.

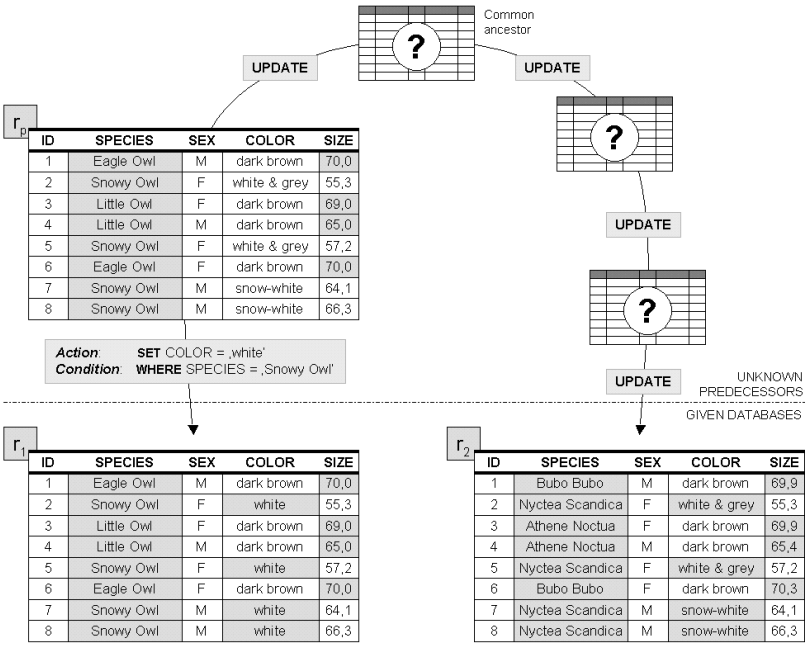

Fig. 1. A model for conflict emergence in overlapping databases

Reproducing conflict generation requires the identification of possible predecessors of the given databases. We consider exactly one predecessor for each of the databases r_1 and r_2 and each non-key attribute A. A predecessor is used to describe conflict generation within attribute A by identifying update operations that modify the predecessor resulting in conflicts between r_1 and r_2. Figure 1 shows the predecessor r_p for database r_1 and attribute COLOR. Also shown is an update operation that describes the generation of conflicts by replacing the finer grained original color specifications in r_p with the more generic term '*white*' in r_1. Currently, we only consider update operations that modify the values within one attribute. We start by defining the considered predecessors and then define a representation for conflict generators.

2.1 Preceding Databases

The databases within this paper are relational databases consisting of a single relation r, following the same schema $R(A_1, \ldots, A_n)$. The domain of each attribute $A \in R$ is denoted by $dom(A)$. Database tuples are denoted by t and attribute values of a tuple are denoted by $t[A]$. There exists a primary key $PK \in R$ for object identification. The primary key attribute is excluded from any modification.

Given a pair of databases r_1 and r_2, the set of potential predecessors is infinite. We restrict this set by allowing the values in the common ancestor to

be modified at most once by any conflict generator. This restriction enables the definition of exactly one predecessor for each of the databases and each non-key attribute.

In the reminder of this paper we consider only conflicts within a fixed attribute $B \in R/\{PK\}$. Let r_p be the predecessor for database r_1. Database r_p equals r_1 in all attributes that are different from B. These values will not be affected by an update operation modifying attribute B. The values for r_p in attribute B are equal to the corresponding values in the contradicting database r_2. These are the values that are changed to generate conflicts between r_1 and r_2:

$$r_p = \{t \mid t \in dom(A_1) \times \cdots \times dom(A_n) \wedge \\ \exists\, t_1 \in r_1,\ t_2 \in r_2\ : \forall\, A \in R\ :\ t[A] = \begin{cases} t_1[A], & \text{if } A \neq B \\ t_2[A], & \text{else} \end{cases} \}$$

2.2 Conflict Generators

A conflict generator is a (condition, action)-pair, where the condition defines the tuples that are modified and the action describes the modification itself. Conditions are represented by closed patterns as defined in the following. The action is reflected by the mapping of values between predecessor and resulting database in the modified attribute. For example, the action of the update operation shown in Figure 1 results in a mapping of values *white & grey* and *snow-white* to value *white*. We classify conflict generators based on the properties of the mapping they define.

Tuples are represented using terms $\tau : (A,\ x)$ that are (attribute, value)-pairs, with $A \in R$ and $x \in dom(A)$. Let $terms(t)$ denote the set of terms for a tuple t. For each attribute $A \in R$ there exists a term $(A, t[A]) \in terms(t)$. A pattern ρ is a set of terms, i.e., $\rho \subseteq \bigcup_{t \in r} terms(t)$. A tuple t satisfies a pattern ρ if $\rho \subseteq terms(t)$. The empty pattern is satisfied by any pattern. The set of tuples from r that satisfy ρ is denoted by $\rho(r)$. We call $|\rho(r)|$ the support of the pattern.

A pattern ρ is called a *closed pattern* if there does not exists a superset $\rho' \subset \rho$ with $\rho'(r) = \rho(r)$. We focus solely on closed patterns as conditions for conflict generators. The set of closed patterns is smaller in size than the set of patterns. Still, closed patterns are lossless in the sense that they uniquely determine the set of all patterns and their set of satisfied tuples (Zaki (2002)).

The Boolean function $conflict_B(t)$ indicates for each tuple $t_p \in r_p$ whether contradicting values exist for attribute B in the corresponding tuples $t_1 \in r_1$ and $t_2 \in r_2$ with $t_1[PK] = t_2[PK] = t_p[PK]$:

$$conflict_B(t) = \begin{cases} true, & \text{if } t_1[B] \neq t_2[B] \\ false, & \text{else} \end{cases}$$

We utilize the interestingness measures *conflict potential* and *conflict relevance* for contradiction patterns to enforce a close relationship between the

tuples affected by a conflict generator and the occurrence of actual conflicts as defined in Müller et al. (2004). The *conflict potential* of a pattern ρ is the probability that a tuple $t \in r_p$ satisfying ρ has a conflict in attribute B, i.e.,

$$pot(\rho) = \frac{|\{t \mid t \in \rho(r_p) \wedge conflict_B(t)\}|}{|\rho(r_p)|}$$

The *conflict relevance* of a pattern ρ is the probability that a tuple $t \in r_p$ with conflict in attribute B also satisfies ρ, i.e.,

$$rel(\rho) = \frac{|\{t \mid t \in \rho(r_p) \wedge conflict_B(t)\}|}{|\{t \mid t \in r_p \wedge conflict_B(t)\}|}$$

A pattern ρ is called a *conflict generator* for attribute B if it has conflict potential and conflict relevance above given thresholds min_{pot} and min_{rel}.

A pair of tuples from r_p and r_1 with identical primary key define a mapping that reflects the modification of values in attribute B as it occurred in the transition from the predecessor to the actual database. We denote this mapping by $t[M]$ for each $t \in r_p$. It follows:

$$t[M] = (x,\, y) \;\Leftrightarrow\; \exists\, t_1 \in r_1 \;:\; t[PK] = t_1[PK] \;\wedge\; t[B] = x \;\wedge\; t_1[B] = y$$

Each conflict generator ρ defines a mapping of values for the tuples that satisfy it, denoted by $map(\rho(r_p)) = \bigcup_{t \in \rho(r_p)} t[M]$. We call a conflict generator *functional* if $map(\rho(r_p))$ defines a function where each x relates exactly to one y. A functional conflict generator is called *injective* if different x values are always mapped to different y value. We call a functional conflict generator *constant* if all x values are mapped to the same y value. This results in four classes of conflict generators, denoted by F for functional, I for injective, C for constant, and $I\&C$ for injective and constant, with $F \supseteq I$, $F \supseteq C$, and $I\&C = I \cap C$.

Regarding the description of conflicts, the action of a functional conflict generator is represented by a function $f(B)$, e.g., the rounding of values. An injective conflict generator for example represents the translation of values between different vocabularies and a constant conflict generator may represent a generalization as in the example shown in Figure 1.

3 Mining Functional Conflict Generators

Mining conflict generators is accomplishable using closed pattern mining algorithms like CHARM (Zaki (2002)) or CARPENTER (Feng Pan et al. (2003)). If we are interested in finding functional conflict generators term enumeration approaches like CHARM have the drawback that there is no ability for pruning based on a given mapping class during pattern mining. Therefore, we

use tuple enumeration in our algorithm for REtrospective functional COnflict GeNeratIon (RECOGNIze) that is outlined in Figure 2.

Mining conflict generators using tuple enumeration is based on the following property: Each set of tuples $s \subseteq r$ defines a pattern, denoted by ρ_s, that is the set of terms common to all tuples in s, i.e., $\rho_s = \bigcap_{t \in s} terms(t)$. If ρ_s is not empty it represents a closed pattern as we cannot add any term to ρ_s without changing the set of tuples that satisfy the pattern. Different tuple sets may define the same closed pattern. Therefore, algorithms like CARPENTER efficiently enumerate those tuple sets that define the complete set of closed patterns satisfying a given support threshold.

RECOGNIze takes as parameters database r_p, conflict potential and relevance thresholds, and the requested class of the returned conflict generators (*classMapping*), i.e., F, I, C, or $I\&C$. The algorithm adopts the CARPENTER algorithm and extends the pruning in subroutine *minePattern* to avoid enumeration of tuple sets that do not represent conflict generators of the requested class. Let s_B denote the subset of r_p containing those tuples that have a conflict in attribute B. The algorithm enumerates all subsets $s_b \subseteq s_B$ that (i) have sufficient size to satisfy the relevance threshold, i.e., $|s_b| \geq min_{rel} * |s_B|$, (ii) whose resulting pattern ρ_{s_b} satisfies the conflict potential threshold, and (iii) where $map(\rho_{s_b})$ represents a valid mapping based on the specified mapping class. The enumeration is done using subroutine *minePattern*. The pa-

`RECOGNIze(`r_p, min_{pot}, min_{rel}, $classMapping$`)`

1. `Initialize` $CG := \{\}$ `and` $s_B := \{t \mid t \in r_p \wedge conflict_B(t)\}$
2. `Minimal support of conflict generators` $tupsup := min_{rel} * |s_B|$
3. `Call` *minePattern*`(`$\{\}$, s_B`)`
4. `return` CG

minePattern`(`s_b`,` s'_B`)`

1. `Determine candidate tuples for extension of` s_b
   $s_u := \{t \mid t \in s'_B \wedge t[PK] > max(s_b) \wedge terms(t) \cap \rho_{s_b} \neq \{\} \wedge compatible(t, s_b)\}$
   `if` $|s_u| + |s_b| < tupsup$ `then` $return$
2. `Determine additional tuples that satisfy` ρ_{s_b}
   $s_y := \{t \mid t \in s_u \wedge terms(t) \supseteq \rho_{s_b}\}$
   `if` $\neg\, validMapping(s_y)$ `then` $return$
3. `if` $\rho_{s_b} \in CG$ `then` $return$
4. `Add` ρ_{s_b} `to the result if all constraints are satisfied`
   `if` $|s_b| + |s_y| \geq tupsup \wedge pot(\rho_{s_b}) \geq min_{pot} \wedge validMapping(\rho_{s_b}(r_p))$ `then`
   $GC := CG \cup \rho_{s_b}$
5. `Enumerate for the remaining candidates the tuple sets`
   `for each` $t \in (s_u - s_y)$ `do`
     minePattern`(`$s_b \cup \{t\}, (s_u - s_y) - \{t\}$`)`

Fig. 2. The RECOGNIze Algorithm

rameters of *minePattern* are the current tuple set s_b, and the set of tuples s'_B that are considered as possible extensions for s_b (other variables are global).

We assume that the elements in s_B are sorted in ascending order of their primary key. Tuple sets are enumerated in depth first order based on the primary key to avoid the enumeration of duplicate tuple sets. In the first step of subroutine *minePattern* the candidate tuples from s'_B for extending s_b are determined. These are the tuples that contain at least one of the terms in ρ_{s_b}. They also have to have a primary key that is greater than the maximum primary key (returned by max) of tuples in s_b to ensure depth first enumeration. In addition to CARPENTER we request that the mapping $t[M]$ is compatible with the mapping defined by ρ_{s_b}regarding the requested class (details below). If the sum of candidates and tuples in s_b is below the minimal tuple support we return, because this tuple set does not define a relevant conflict generator.

The second step determines the subset of candidates that satisfy ρ_{s_b}. It follows that $s_b \cup s_y$ defines the set of tuples from s_B that satisfy ρ_{s_b}. There is an additional ability for pruning, if the tuples in s_y do not define a mapping that is valid for the desired class. The tuples in s_y are not considered as further extensions of s_b as this would only generate identical closed patterns. Still, in Step 3 we have to check whether ρ_{s_b} is already contained in CG. Otherwise we add ρ_{s_b} to CG if all three constraints as listed above are satisfied. We then extend the current tuple set using the remaining candidates in $s_u - s_y$ and call *minePattern* recursively in order to build the complete tuple set enumeration tree.

The subroutines *compatible* and *validMapping* check whether a mapping $map(s)$ is valid regarding *classMapping*. This check is trivial for either C or $I\&C$ where we test for equality of the y values (and the x values in case of $I\&C$) of the elements in $map(s)$. The case of *classMapping* being F or I is described in the following: For each element in $map(r_p)$, referred to as *mapping term* in the following, we maintain a list of incompatible mapping terms, denoted by *incomp*. A pair of mapping terms $(x_1,\ y_1), (x_2,\ y_2)$ is considered incompatible for conflict generators of class F if $x_1 = x_2$ and $y_1 \neq y_2$. The mapping terms are incompatible for class I if they are incompatible for class F or $x_1 \neq x_2$ and $y_1 = y_2$. In subroutine *compatible* we request $incomp(t[M])$ to be disjoint with $\bigcup_{t \in s_p} incomp(t[M])$, i.e., the mapping term is not incompatible with any of the mapping terms currently in s_b. For $validMapping(s)$ to be true we request that $\bigcap_{t \in s} incomp(t[M]) = \{\}$, i.e., there exists not incompatibilities between the mapping terms of the tuples in s.

For the example in Figure 1 there are three functional conflict generators for a relevance threshold of 50% representing conflicts for femal snowy owls, male snowy owls, and snowy owls in general. They all have conflict relevance of 100%. The first two conflict generators belong to class $I\&C$ with the other belonging to C. Experiments with data sets of protein structures as used in Müller et al. (2004) show that an average of 63% of the patterns for each attribute represent functional conflict generators, with C and $I\&C$ being the most frequently subclasses.

4 Related Work

Weiguo Fan et al. (2001) present a method for finding patterns in contradictory data to support conflict solution. Their focus is the identification of rules that describe the conversion of contradicting values. They do not request these rules to be associated with a descriptive condition as in our approach. We do not consider the identification of complex data conversion rules. However, the mappings defined by conflict generators could be used as input for the methods described in Weiguo Fan et al. (2001). There is also a large body of work on statistical data editing, i.e., the automatic correction of conflicting values, based on the Fellegi-Holt-Model (Fellegi and Holt (1976)). These approaches rely on edits (rules) for conflict detection and determine the minimal number of changes necessary for conflict elimination. In contrast, we use object identifiers for conflict identification and currently do not consider automatic value modification.

5 Conclusion

The classification of conflict generators in this paper allows to restrict contradiction pattern mining to those patterns that represent certain situations of conflict generation, e.g., the usage of different vocabularies. The presented algorithm has the advantage of being able to prune patterns not belonging to the requested class at an earlier stage of pattern enumeration. As future work we consider, based on the idea of statistical data edits, to determine the minimal set of update operations that have to be undone in order to derive the common ancestor of a given pair of databases.

References

H. MÜLLER, U. LESER and J.-C. FREYTAG (2004): Mining for Patterns in Contradictory Data. *Proc. SIGMOD Int. Workshop on Information Quality for Information Systems (IQIS'04), Paris, France.*

M.A. HERNANDEZ and S.J. STOLFO (1995): The merge/purge problem for large databases. *Proc. Int. Conf. Management of Data (SIGMOD), San Jose, California.*

M.J. ZAKI (2002): CHARM: An Efficient Algorithm for Closed Itemset Mining. *Proc. of the Second SIAM Int. Conf. on Data Mining, Arlington, VA.*

WEIGUO FAN, HONGJUN LU, S.E. MADNICK and D. CHEUNG (2001): Discovering and reconciling value conflicts for numerical data integration. *Information Systems, Vol. 26, pp. 635-656.*

FENG PAN, GAO CONG, A.K.H. TUNG, JIONG YANG and M.J. ZAKI (2003): CARPENTER: finding closed patterns in long biological datasets. *Proc. Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD), Washington DC.*

P. FELLEGI and D. HOLT (1976): A Systematic Approach to Automatic Edit and Imputation. *Journal of the American Statistical Association, 71, 17-35.*