Describing Differences between Databases

Heiko Müller
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
hmueller@informatik.hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
freytag@informatik.hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

We study the novel problem of efficiently computing the update distance for a pair of relational databases. In analogy to the edit distance of strings, we define the update distance of two databases as the minimal number of insert, delete and modification operations necessary to transform one database into the other. In contrast to related approaches we consider set-oriented instead of one-tuple-at-a-time operations. We show how this distance can be computed by traversing a search space of database instances connected by update operations. This insight leads to a family of algorithms that compute the update distance or approximations of it. In our experiments we observed that a simple heuristic performs surprisingly well in most considered cases.

Our motivation for studying distance measures for databases stems from the field of scientific databases. There, replicas of a single database are often maintained at different sites, which typically leads to (accidental or planned) divergence of their content. To re-create a consistent view, these differences must be resolved. Such an effort requires an understanding of the process that produced them. We found that minimal update sequences of set-oriented update operations are a proper and concise representation of systematic errors, thus giving valuable clues to domain experts responsible for conflict resolution.

1. INTRODUCTION

Today, many scientific databases overlap in their sets of represented objects due to redundant data generation or data replication. For instance, in life science research it is common practice to distribute the same set of samples, such as clones, proteins, or patient's blood, to different laboratories to enhance the reliability of analysis results. Whenever overlapping data is generated or administered at different sites, there is a high probability of the occurrence of differences. These differences do not need to be accidental, but could be the result of different data production and processing workflows. For example, the three protein structure databases OpenMMS [5], MSD [6], and Columba [19] are all copies of the Protein Data Bank PDB [4]. However, due to different cleansing strategies, these copies vary substantially. Thus, a biologist is often faced with conflicting copies of the same set of real world objects and with the problem of solving these conflicts to produce a consistent view of the data.

Conference'06, Month 1–2, 2006, City, State, Country.

Many inconsistencies are highly systematic caused by the usage of different controlled vocabularies, different measurement units, different abbreviations, or by consistent bias during experimental analysis. Learning about the reasons that led to inconsistencies is valuable information when assessing the quality of and reasons for contradicting values to resolve conflicts. Unfortunately, in most cases only the final databases are accessible without any additional knowledge about the data generation or manipulation processes. Assuming that conflicts do not occur randomly, but follow specific (but unknown) regularities, patterns of the form "IF *condition* THEN *conflict"* provide valuable knowledge to facilitate their understanding. A domain expert could evaluate these patterns to assess the correctness of conflicting values and therefore for conflict resolution. In [16] we proposed an algorithm for finding such patterns using association rule mining.

In this paper, we develop a different approach for describing "regular differences" among contradicting databases: The detection of *minimal update sequences* that transform one database into the other. We consider SQL-like update operations because these are the fundamental operations for the manipulation of data stored in relational databases. Given a pair of contradicting databases, each operation may (i) directly represent an update operation that has been applied to one of the databases if they both evolved from a common ancestor (as in the PDB example), or (ii) describe systematic differences between the databases. Consider the example in Figure 1 showing the fictitious results of two groups investigating the same set of owls (identified by their IDs). For transforming DBOwl into OWLBase Latin species names must be replaced with common names. We also need to modify the color for all snowy owls, as DBOwl uses a finer grained distinction in colors. Therefore, regardless of how these databases were generated, there are update operations that allow an inference on possible reasons for the existing conflicts.

DBOwl

ID	Species	Color
1	Bubo Bubo	brown
2	Ath. Noctua	brown
3	N. Scandica	white & gray
4	N. Scandica	snow-white

OWLBase

ID	Species	Color
1	Eagle Owl	brown
2	Little Owl	brown
3	Snowy Owl	white
4	Snowy Owl	white

Figure 1: Contradicting databases resulting from different groups investigating the same set of objects.

We propose to use minimal sequences of update operations that turn one database into the other. We call this number the *update distance* between two databases. Each sequence of operations as long as the update distance is one of the simplest possible explanations for the observed differences. Following the "Occam's Razor" principle, we conclude that the simplest explanations are also the most likely. Minimal update sequences give valuable clues on what has happened to a database to make it different from its original state. Therefore, the update distance is a semantic distance measure, as it is inherently process-oriented in contrast to purely syntactic measures such as counting different values.

The update distance is defined analogous to the edit distance for strings [14], i.e., the minimal number of edit operations that transform one string into another. In contrast to editing strings, set-oriented database updates strongly influence each other thus giving importance to the order of operations. To give an idea of the complexity of the problem, consider the two databases of Figure 2. A first approach for transforming r_1 into r_2 solves each conflict individually (Figure 2a). The five resulting operations could be executed in any order. A greedy approach, solving as many conflicts as possible with each operation, would result in a total of four update operations (Figure 2b). However, the minimal update sequence for transforming r_1 into r_2 has three update operations (Figure 2c). Intuitively, it is often necessary to use operations that in first place introduce new conflicts, because these conflicts can be used as discriminating conditions in later update operations. The first operation in Figure 2c temporarily increases the total number of conflicts, but this is compensated in later operations. Clearly, the minimal update sequence can be determined by enumerating update sequences of increasing length until one sequence is found that implements all necessary changes. In this example, this would generate 294,998 intermediate states, as we show in Section 4.1.

r_1

A_1	A_2	A_3
1	1	1
2	1	1
3	1	1
4	2	1
5	3	1
6	4	1
7	5	1
8	6	1
9	1	0
10	1	0

r_2

A_1	A_2	A_3
1	1	1
2	1	1
3	1	1
4	2	0
5	3	0
6	4	0
7	5	0
8	6	0
9	1	0
10	1	0

a) UPDATE r_1 SET $A_3 = 0$ WHERE $A_1 = 4$
UPDATE r_1 SET $A_3 = 0$ WHERE $A_1 = 5$
UPDATE r_1 SET $A_3 = 0$ WHERE $A_1 = 6$
UPDATE r_1 SET $A_3 = 0$ WHERE $A_1 = 7$
UPDATE r_1 SET $A_3 = 0$ WHERE $A_1 = 8$

b) UPDATE r_1 SET $A_3 = 0$
UPDATE r_1 SET $A_3 = 1$ WHERE $A_1 = 1$
UPDATE r_1 SET $A_3 = 1$ WHERE $A_1 = 2$
UPDATE r_1 SET $A_3 = 1$ WHERE $A_1 = 3$

c) UPDATE r_1 SET $A_3 = 2$ WHERE $A_2 = 1$ AND $A_3 = 1$
UPDATE r_1 SET $A_3 = 0$ WHERE $A_3 = 1$
UPDATE r_1 SET $A_3 = 1$ WHERE $A_3 = 2$

Figure 2: A pair of contradicting databases with three different update sequences.

In this paper, we present exact and approximate algorithms for computing the update distance and for finding minimal sequences of update operations for a pair of databases. Even though we consider only a restricted form of updates (namely those where attribute values are set to constants), our algorithms for computing the exact solution require exponential space and time. However, we also present greedy strategies that lead to good results in all examples we considered. The rest of this paper is structured as follows: In the next section we discuss related work. Section 3 gives our definition of update distance and derives an upper and lower bound, which are important for optimization. In Section 4, algorithms for finding minimal sequences are presented. Section 5 discusses fast approximate algorithms. In Section 6 we present experimental results, and Section 7 concludes our paper. There is also a longer version of this paper available for further details [17].

2. RELATED WORK

To the best of our knowledge the problem of finding minimal sequences of set-oriented update operations for relational databases has not been considered before. The main areas of related work are: definitions of distance for databases, representing differences of databases, and finding patterns in conflicting data.

There are other definitions of database distance based on the modification of databases. One comes from the area of computing consistent query answers for inconsistent databases [1]. Here, a repair for a database r violating a set of integrity constraints IC is defined as a database r' that satisfies IC and has minimal distance to r. In this context, the distance is defined as the number of tuples that are contained in one database, but not in the other. Thus, the distance definition is very different from ours, as we need to find the actual update sequences for transforming one database into the other. Bohannon et al. presented an algorithm for finding a repair that is the cheapest under a cost model for modifications and updates, also leading to a quite different notion of distance [7]. While their approach needs to consider the order of their operations they do not consider set-oriented update operations.

The latter is also true for other approaches that represent differences between databases. Chawathe et al. and Labioet et al. use sequences of insert, delete, and update operations to represent differences between database snapshots [13] or hierarchically structured data [8]. Both approaches consider only operations that affect a single tuple. Since databases are manipulated with (set-oriented) SQL commands, we consider our problem as more natural than a tuple-at-a-time approach. Several other applications use so called "update deltas" to represent differences between databases. In database versioning they are used as memory effective representation of different database versions [10]. However, versioning collects the actual operations during execution instead of having to rediscover them from two given versions.

Other methods for finding patterns in contradictory data to support conflict resolution are presented in [16] and [11]. In [11], the authors distinguish between context dependent and context independent conflicts. Context dependent conflicts represent systematic disparities that are consequences of conflicting assumptions or interpretations. Context independent conflicts are idiosyncratic in nature and are consequences of random events, human errors, or imperfect instrumentation. Accordingly, we consider only context dependent conflicts. However, in contrast to [11], we do not

consider complex data conversion rules for conflict resolution. Instead we always use one of the conflicting values as solution. We consider the discovery of conflict conversion rules as future work in Section 7. However, we do consider the conflict causing context to be identifiable as data patterns. In [16], we use a frequent itemset mining algorithm to find conflicts frequently occurring together with certain patterns in the data. These patterns can only provide a static view on the differences and do not take into account possible dependencies between conflicts. In the approach presented here, such dependencies are implicitly captured by the order of update operations within a (minimal) update sequence.

A prerequisite of our approach is the ability to identify identical real-world objects in different databases. In general, this is a difficult problem with a long history of research [12, 15, 22] that we consider as orthogonal to our problem. Throughout this paper, we assume the existence of a source-spanning object identifier. These identifiers may have been assigned to tuples by a preceding duplicate detection step.

3. UPDATE DISTANCE

In this section we formally define the update distance, i.e., the minimal update sequence, between two relational databases. Each database consists only of a single relation r, following the same schema $R(A_1, \ldots, A_n)$. The domain of each attribute A is denoted by *dom*(A). Without loss of generality we assume *dom*(A) = ℕ for all $A \in R$ and that A_1 being the primary key of each r. We will use ID as a synonym for A_1. Tuples are denoted by t and attribute values of a tuple are denoted by t[A]. We use t{j} to refer to the tuple with primary key value j, j = 1, …, m. As stated in the previous section, we do not consider the problem of assigning unique key values to tuples that allow the association of tuples representing the same real-world objects in different databases.

In the following, we introduce the necessary concepts to define update sequences. Section 3.1 defines matches and conflicts between two databases. Section 3.2 introduces the types of basic operations we assume as possible updates and assembles them into update sequences. These types naturally lead to the notion of shortest update sequences. In Section 3.3, we define a first upper and lower bound for update distances which we use later for pruning.

Our basic update operations essentially are SQL updates that set attributes selected by conjunctions of equality patterns to constants. Even with this simple model, computing minimal update distances is not trivial, as we shall see in Section 4. Thus, we leave extensions to databases consisting of multiple relations and to more powerful updates operations for future work.

3.1 Contradicting Databases

A pair of tuples from databases r_1 and r_2 is called a *matching pair* if they posses identical primary key values. The *set of all matching pairs* between databases (i.e., relations) r_1 and r_2 is denoted by $M(r_1, r_2)$. Let $m = (t_1, t_2)$ be a matching pair from $M(r_1, r_2)$. The different tuples from m are denoted by $tup_1(m)$ and $tup_2(m)$. The equal primary key value of both tuples is denoted by *id*(m). A pair of databases r_1 and r_2 is called *overlapping* if $M(r_1, r_2) \neq \emptyset$. Note that we are only interested in finding regularities between the overlapping parts of databases. We do not consider finding regularities within the sets of tuples missing in either of the databases. Therefore, we assume $|r_1| = |r_2| = |M(r_1, r_2)|$.

Within a matching pair several conflicts may occur. We represent each conflict by the matching pair m and the attribute A in which the conflict occurs.

DEFINITION 1 (SET OF CONFLICTS): The *set of conflicts* between a pair of databases r_1 and r_2, denoted by $C(r_1, r_2)$, is the set of all tuples (m, A) where a conflict in attribute A of pair m exists, i.e.,

$$C(r_1, r_2) = \{(m, A) \mid (m, A) \in M(r_1, r_2) \times R \wedge tup_1(m)[A] \neq tup_2(m)[A]\}.$$

A pair of databases r_1 and r_2 is called *contradicting*, if there exists at least one conflict between them, i.e., $C(r_1, r_2) \neq \emptyset$. As an example, consider the contradicting databases of Figure 2. The set of matching pairs contains ten elements, and there are five conflicts, i.e., $C(r_1, r_2) = \{((t_1\{4\}, t_2\{4\}), A_3), ((t_1\{5\}, t_2\{5\}), A_3), ((t_1\{6\}, t_2\{6\}), A_3), ((t_1\{7\}, t_2\{7\}), A_3), ((t_1\{8\}, t_2\{8\}), A_3)\}$.

3.2 Update Sequences

Update operations are used to modify existing databases. They can be considered as functions that map databases onto each other. Let ℜ(R) denote the infinite set of databases following schema R that satisfy the primary key constraint. An *update operation* is a mapping $\psi: \Re(R) \rightarrow \Re(R)$. For relational databases there are three types of basic update operations, namely insert, delete, and modify [20]. For space limitations, we consider only modify operations and not insert and delete operations for this paper. This restriction also has a practical reason: Consider an object being stored (with conflicts) in two databases r_1 and r_2 as tuples t_1 and t_2, respectively. In this case, we want to find the modifications that are necessary to unify both descriptions. We are not interested in an update sequence that, for transforming r_1 into r_2, first deletes t_1 and then inserts t_2 with new values. However, our algorithms can be extended to include insert and delete operations, as discussed in [17].

Before defining sequences of update operations, we first fix the expressiveness of our basic update operations.

DEFINITION 2 (TERM): A *term* τ over schema R is a pair (A, x), with attribute $A \in R$ and value $x \in dom(A)$. We define $attr(\tau) = A$ and $value(\tau) = x$.◆

A term can be interpreted as a Boolean-function on tuples. A tuple t satisfies τ iff $t[attr(\tau)] = value(\tau)$. By τ(r) we denote the set of tuples from r satisfying τ. We say that the tuples in τ(r) are *selected by* τ. The number of tuples selected by τ is called the *support* of τ in r. Terms are combined to patterns selecting the set of tuples affected by an update.

DEFINITION 3 (PATTERN): A *pattern* ρ over schema R is a set of terms over R, where all terms are defined over different attributes, i.e.,

$$\forall\, \tau_i, \tau_j \in \rho : attr(\tau_i) = attr(\tau_j) \Leftrightarrow \tau_i = \tau_j.$$◆

A tuple t satisfies ρ if it satisfies each term within ρ. A pattern is therefore a conjunction of terms. The empty pattern is satisfied by each tuple of a database. Analogous to the definitions above, ρ(r) denotes the set of tuples selected by ρ. A pattern describes which tuples should be changed by a modification operation.

DEFINITION 4 (MODIFICATION OPERATION): A *modification operation* $\psi_\mu = (\tau, \rho)$ is a term-pattern pair with $attr(\tau) \in R / ID$.◆

The definition of a modification operation (i) excludes the primary key attribute from being modified, and (ii) allows only one attribute to be modified. When applied to a database r, a modification operation modifies all tuples that satisfy ρ. For these tuples, the value for attribute *attr*(τ) is set to *value*(τ). Note, that there does not necessarily exist a reverse operation for each modification operation. For example, the operation $\psi = ((A_2, 7), \{(A_3, 1)\})$ sets the value for attribute A_2 to 7 for the tuples t{1}, …, t{8}, when applied to database r_1 of Figure 2. We need at least six modification operations to undo this single operation.

DEFINITION 5 (UPDATE SEQUENCE): A list of update operations $\Psi = <\psi_1, \ldots, \psi_k>$ is called an *update sequence*. When applied to database r_1, an update sequence generates (or derives) database $\Psi(r_1)$ by applying the update operations in the given order: $\Psi(r_1) = \psi_k(\ldots\psi_2(\psi_1(r_1))\ldots)$.♦

Obviously, the order of operations within an update sequence is important. For example, update sequences $\Psi_1 = <\psi_{\mu1}, \psi_{\mu2}>$ and $\Psi_2 = <\psi_{\mu2}, \psi_{\mu1}>$ with $\psi_{\mu1} = ((A_2, 7), \{(A_3, 1)\})$ and $\psi_{\mu2} = ((A_3, 7), \{(A_3, 1)\})$ have different results when applied to database r_1 of Figure 2. The first sequence results in a database, where the value for attribute A_2 and A_3 is 7 in tuples t{1}, …, t{8}. In the second sequence the operation $\psi_{\mu1}$ has no effect, as the pattern is no longer satisfied by any of the tuples after applying operation $\psi_{\mu2}$.

We call Ψ a *transformer* for r_1 and r_2, iff $\Psi(r_1) = r_2$. The number of update operations within a sequence is called its length and is denoted by |Ψ|. Figure 2 lists three update sequences (a-c) of different length, which are transformers for the databases shown.

DEFINITION 6 (MINIMAL TRANSFORMER): For a pair of databases r_1 and r_2, an update sequence Ψ with $\Psi(r_1) = r_2$ is called a *minimal transformer* for r_1 and r_2 if there does not exist another transformer Ψ' with $\Psi'(r_1) = r_2$ and $|\Psi'| < |\Psi|$.♦

Clearly, minimal transformers for a pair of databases are not unique. The set of all minimal transformers for r_1 and r_2 is denoted as $T(r_1, r_2)$.

DEFINITION 7 (UPDATE DISTANCE): For a pair of databases r_1 and r_2, the *update distance* $\Delta_U(r_1, r_2)$ is defined as the length of any minimal transformer for r_1 and r_2.♦

Note that the update distance is not a metric as it is not a symmetric relation. Based on our definitions, we are ready to define the main problem considered in this paper.

PROBLEM STATEMENT: Given a pair of databases r_1 and r_2, find all minimal transformers from r_1 to r_2.

3.3 Upper and Lower Bounds

This subsection defines upper and lower bounds for the update distance. They are later utilized as pruning criteria.

DEFINITION 8 (RESOLUTION DISTANCE): For a pair of databases r_1 and r_2, the *resolution distance* $\Delta_R(r_1, r_2)$ is defined as the number of conflicts between them, i.e., $\Delta_R(r_1, r_2) = |C(r_1, r_2)|$.♦

Transforming a database r_1 into a database r_2 requires the conflicts between them to be solved by replacing conflicting values in r_1 with their according values from r_2. Since each tuple can be selected by a pattern using its ID as selection criteria and every modification operation can at least change one value, the resolution distance is an upper bound for the update distance.

LEMMA 1: For each pair of databases r_1 and r_2 there exists a transformer Ψ of length $\Delta_R(r_1, r_2)$.

PROOF: Omitted for space limitations.♦

To define a lower bound as well, we recognize that each update operation modifies only one attribute according to our definition. We subsume the conflicts that are potentially solvable using a single modification operation within a *conflict group*.

DEFINITION 9 (CONFLICT GROUP): Given a pair of databases r_1 and r_2. A *conflict group* κ is an attribute-value pair (A, x) with *attr*(κ) = A and *value*(κ) = x. κ represents the subset of conflicts (m, A) between r_1 and r_2 having the following property:

$$(m, A) \in C(r_1, r_2) \wedge attr(\kappa) = A \wedge value(\kappa) = tup_2(m)[A].$$♦

Thus, all conflicts represented by a conflict group κ occur in the same attribute A and have the same solution x. These conflicts are hence solvable using a modification operation with κ as the modification term. Let $K(r_1, r_2)$ be the set of all conflict groups between a pair of databases. According to this definition, the only conflict group for the databases in Figure 2 is $\kappa = (A_3, 0)$.

LEMMA 2: For each pair of databases r_1 and r_2, there cannot exist a transformer for r_1 and r_2 that is shorter than $|K(r_1, r_2)|$. We denote this lower bound as $LB(r_1, r_2)$.

PROOF: To transform r_1 into r_2 we need at least one modification ψ_μ operation per conflict group $\kappa \in K(r_1, r_2)$, with $\tau = \kappa$.♦

For the example in Figure 2 the update distance is three, as shown by the update sequence in c). The lower bound of the update distance is one and the upper bound is five.

4. TRANSIT– COMPUTING MINIMAL TRANSFORMERS

This section describes the TRANSIT algorithms to determine the set of minimal TRANSformers for contradIcting daTabases r_1 and r_2. Both algorithms essentially enumerate the space of all databases reachable by applying sequences of modifications to r_1. Doing so efficiently poses several challenges for which we describe solutions. First, we introduce *transition graphs* as formalizations of the search problem. Since many update sequences lead to the same database state, duplicate detection is of outmost importance. We describe a hashing scheme for efficient duplicate checking. We show how we use the upper and lower bounds defined in Section 3.3 to prune the search space, leading to a branch and bound algorithm. We then describe a breadth-first strategy for traversing the search space and briefly sketch a depth-first strategy. In Section 4.2, we show how - given a database state - the set of all possible modification operations can be computed using a mining algorithm that computes closed frequent itemsets.

4.1 Search Space Exploration

Given a pair of databases r_o and r_t, called origin and target, our goal is to determine $T(r_o, r_t)$. Our approach essentially starts by determining all databases derivable from r_o by a single modification operation, called level-1 databases. Level-2 databases are computed by using all level-1 databases as starting point for another modification. This process continues until we reach r_t. The level at which the target is reached first reflects the minimal num-

ber of modification operations necessary to derive r_t from r_o, i.e., the update distance. To determine $T(r_o, r_t)$ the algorithm also needs to enumerate all other sequences that are of the same length. We maintain the sequence of modification operations with each database. Since multiple sequences may generate the same database, level-n databases may have an update distance that is actually shorter than n. We later treat the detection of duplicated databases. Since we enumerate all possible modifications at each level and for each database, we ensure that our first match with r_t defines the shortest possible sequence.

Transition Graph

We represent the search space using a directed labeled graph, called the *transition graph*. Vertices of this graph are databases connected by directed edges representing modification operations.

DEFINITION 10 (TRANSITION GRAPH): For two databases r_o and r_t, the transition graph $G_T = (V, E)$ with vertices V and edges E is defined as follows: V is the set of all databases derivable from r_o using an update sequence of length shorter than or equal to the update distance $\Delta_U(r_o, r_t)$. This implies that $r_t \in V$. E is the set of all edges $e = (r_1, r_2, \psi)$ for which $\psi(r_1) = r_2$, $r_1, r_2 \in V$. We call r_1 the source of e, denoted by *source*(e), and r_2 the target of e, denoted by *target*(e).♦

DEFINITION 11 (PATH): A *path* $\varphi = \langle e_1, \ldots, e_p \rangle$ within transition graph $G_T = (V, E)$ is a sequence of edges from E, with *source*(e_i) = *target*(e_{i-1}) for all $1 < i \leq p$. Two databases r_1 and r_2 are connected by φ if *source*(e_1) = r_1 and *target*(e_p) = r_2.♦

Each path between two databases r_1 and r_2 defines a transformer $\Psi(r_1) = r_2$. In accordance to DEFINITION 6 a path is minimal if no shorter path between the same two databases exists.

The TRANSIT-algorithms iteratively construct the transition graph starting with database r_o as the only vertex. Figure 3 shows an example. Levels are outlined by horizontal lines and derivable databases are only shown at the level of their update distance from r_o. Vertices and edges on the minimal paths between the origin and target are enclosed within a gray box.

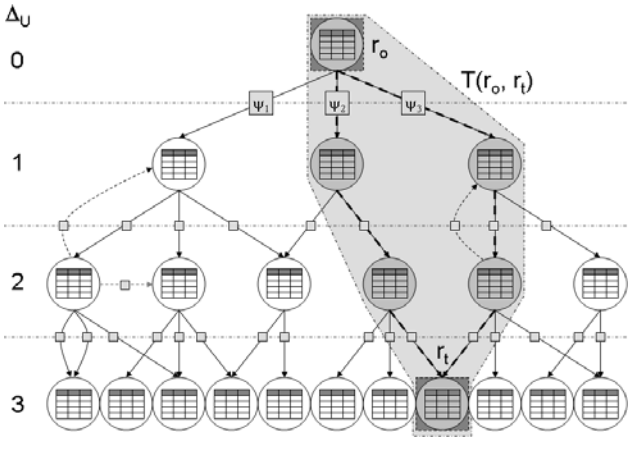

Figure 3: An exemplified transition graph as generated by the TRANSIT algorithm without performing any pruning of databases.

Duplicate Detection

Duplicates at different levels of the graph may introduce cycles. Since the corresponding edges - delineated by dotted lines for clarity in Figure 3 - cannot be part of a minimal transformer, they are not included in the graph. Intra-level duplicates may result in multiple edges between two vertices on adjacent distance levels. As we will show in Section 6, a large portion of all generated databases are duplicates. For example, the operations $\psi_1 = ((A_3, 0), \{(A_3, 1)\})$ and $\psi_2 = ((A_3, 0), \{\})$ derive the same result when applied to database r_1 of Figure 2. Also, many update sequences derive a database from itself. For example, the update sequence $\langle((A_2, 0), \{(A_1, 1)\}), ((A_2, 1), \{(A_1, 1)\})\rangle$ derives r_1 from r_1 using a 2-step update sequence. We must detect duplicates efficiently to avoid unnecessary explosion of the search space.

Duplicate detection requires comparison of entire databases. To reduce the number of duplicate checks, we compute a hash value for each database and maintain a hash table for generated databases. Complete database comparisons are only performed when the hash values of two databases are equal, which drastically reduces the number of (expensive) full database comparisons at the price of having to maintain the hash table.

We currently employ the following hash function for databases: Without a loss of generality we assume the IDs to be integers in the range 1, …, m. We number the attribute values of the particular tuples in the following order 0:t{1}[A_1], 1:t{1}[A_2], …, (m*n)-1:t{m}[A_n], called the cell index. With each database we maintain a list of the conflicting values with an order based on this cell index. We select k values from this list, having cell index values $c_1, \ldots, c_k$. The final hash value is an integer with k digits, where the i-th digit is the value of cell c_i modulo 10. We also tested a hash function based on a histogram of the attribute values occurring within a database, but we found the later scheme inferior in our experiments.

Pruning

The TRANSIT-algorithms try to avoid generating the complete transition graph. The number of vertices outside of the minimal transition graph in Figure 3 shows that many of the generated databases are not part of any minimal transformer. This observation is supported by the following example:

EXAMPLE 1: Enumerating the complete transition graph with duplicate detection for the databases shown in Figure 2 results in a graph with 294,998 vertices and 768,333 edges. However, only two minimal paths from r_1 to r_2 exist, which together contain only six vertices.♦

This large difference suggests that pruning is essential. In TRANSIT, pruning uses the upper and lower bounds for the update distance as defined in Section 3.3. Let β denote the current upper bound for the update distance between r_o and r_t. This bound is initialized following LEMMA 1 as $\Delta_R(r_o, r_t)$. Each generated database r with $LB(r, r_t) > (\beta - \Delta_U(r_o, r))$ is not included in the transition graph, because any path from r_o to r_t through r will have at least $\Delta_U(r_o, r) + LB(r, r_t) > \beta$ edges and is therefore not minimal. The update distance $\Delta_U(r_o, r)$ is maintained with each vertex in order to avoid recalculation.

We decrease β whenever a database r is generated with $(\Delta_U(r_o, r) + \Delta_R(r, r_t)) < \beta$. For such a database there exists a transformer $\Psi(r_o) = r$ with $|\Psi| = \Delta_U(r_o, r)$. Then, LEMMA 1 guarantees the exis-

tence of a transformer $\Psi'(r) = r_t$ with length $|\Psi'| = \Delta_R(r, r_t)$. The following simple Lemma proofs the existence of a transformer $\Psi''(r_o) = r_t$ having length $|\Psi''| = \Delta_U(r_o, r) + \Delta_R(r, r_t)$.

LEMMA 3: Given transformers $\Psi_1(r_1) = r_2$ and $\Psi_2(r_2) = r_3$, there exists a transformer $\Psi_3(r_1) = r_3$ with length $|\Psi_3| = |\Psi_1| + |\Psi_2|$.

PROOF: Omitted for space limitations. ♦

Each time the current bound β is decreased we remove all databases r from the transition graph with insufficient bound, i.e., for which $\Delta_U(r_o, r) + LB(r, r_t) > \beta$.

Breadth-First Algorithm

The previous approach resembles a branch and bound behavior. Therein, we can explore the search space either in breadth-first or in depth-first manner. We first describe a breadth-first algorithm (see Figure 4).

The algorithm generates all databases derivable by update sequences of increasing length. Within the branch step a database is chosen for processing. We generate all databases that are derivable from this database by a single modification operation. Next, in the bound step the current bound is decreased if possible and databases are pruned as described. After finishing the processing of the current database we chose the next database for processing from the remaining, untested databases in the graph. We process all databases at the current level first before proceeding to databases at the next level. We continue until r_t is reached and no untested database remains.

```
1  TRANSIT-BFS(r_o, r_t) {
2    G_T := ({r_o}, {});
3    V_P := V(G_T);
4    Δ_U := 0;
5    β := Δ_R(r_o, r_t);
6    while (r_t ∉ V_P) {
7      Δ_U := Δ_U + 1;
8      V_C := {};
9      for each r_i ∈ V_P do {
10       MDF := modifier(r_i, r_t);
11       for each ψ ∈ MDF do {
12         r_new := ψ(clone(r_i));
13         if ((LB(r_new, r_t) + Δ_U) ≤ β) {
14           if (r_new ∉ V(G_T))
15             V(G_T) := V(G_T) ∪ {r_new};
16             E(G_T) := E(G_T) ∪ (r_i, r_new, ψ);
17             V_C := V_C ∪ {r_new};
18             if ((Δ_R(r_new, r_t) + Δ_U) < β) {
19               β := Δ_R(r_new, r_t) + Δ_U;
20               prune V_P, V_C, G_T, β;
21             }
22           } else if (r_new ∈ V_C) {
23             E(G_T) := E(G_T) ∪ (r_i, r_new, ψ);
24     }}}}
25     V_P := sort(V_C);
26   }
27   output min_paths(G_T, r_o, r_t);
28 }
```

Figure 4: The TRANSIT-BFS algorithm.

Figure 4 shows the corresponding algorithm TRANSIT-BFS. Each database from the previous level, maintained in V_P, is processed while enumerating the current level (*lines* 9-24). The databases at the current level, maintained in V_C, afterwards become the candidates for the enumeration of the next level (*line* 25). We sort the candidates in ascending order of their lower and upper bounds. This is done with the intention of decreasing the current bound β as soon as possible, therefore avoiding the unnecessary insertion of databases that are pruned afterwards. After reaching the destination the algorithm returns the set of minimal paths in the transition graph from the origin to the target (*line* 27).

If we are interested in calculating the update distance only, the algorithm can terminate immediately after r_t is derived for the first time (check for equality after *line* 12).

Processing a database starts by determining the set of possible modification operations (*line* 10 – see Section 4.2 for details). Each of the operations is applied to a copy of the database, as modification operations alter the given database (*line* 12). The resulting database is added to the transition graph and to V_C if it does not already occur within the graph (*lines* 14-17). Otherwise, the database is a duplicate. It is an intra-level duplicate, if it also occurs in V_C. In this case the database has been derived before at the current distance level. Intra-level duplicates add additional edges. Otherwise, no changes occur.

Depth-First Algorithm

The transition graph may also be constructed in depth-first manner. We refer to the corresponding algorithm as TRANSIT-DFS. Within this algorithm, we immediately proceed to the next distance level after finishing the processing of the current database, i.e., generating all databases derivable with a single modification operation. We chose that database from all generated ones with the smallest lower bound as the new current database. Pruning is performed as described above. The depth-first approach finds a first solution after processing fewer databases then the breadth-first approach. Although this solution is not necessarily optimal, it often helps to perform more pruning. After reaching the target database, TRANSIT-DFS needs to return to the previous databases and test them as candidates, again in a depth-first manner. The algorithm continues until all databases that have not been pruned by the bounding step have been tested.

Compared to TRANSIT-BFS, duplicate detection is more complicated because identical databases may be generated multiple times at decreasing distance levels. Every time a database is derived at a lower level once more, it must be considered as a candidate again and cannot be rejected as a duplicate.

The experiments of Subsection 6.1 show the advantage of using the branch and bound approaches over enumerating the complete transition graph.

4.2 Enumerating Modification Operations

According to DEFINITION 4, a modification operation is a pair consisting of a modification term τ and a modification pattern ρ. We are only interested in enumerating modification operations that change the database. We call these operations *valid*. Then, the set of possible modification operations for a database r is the Cartesian product of the set of valid modification terms and the set of patterns.

Valid Modification Terms

Terms (DEFINITION 2) are attribute-value pairs. Only non-key attributes are permitted within modification terms. The set of valid modification terms is the union of valid modification terms for each non-key attribute. For each attribute $A \in R / ID$ this set is $A \times dom(A)$. A problem is the infinite size of $dom(A)$ that leads to an infinite set of modification terms and therefore to an infinite set of modification operations. Consequently, the algorithm would not terminate, although almost all of the generated databases are isomorphic with respect to their ability to participate in a shortest update sequence.

We must therefore restrict the set of possible modification values by only using the values occurring within the current database r and the target r_t. In summary, we permit the following values in modification terms for attribute A:

- All values from the target that occur within attribute A, denoted by $r_t[A]$. Some of these values have to be used at least once as modification value for conflict solution. The remaining values are also contained in the following set.
- All values occurring for attribute A in the current database r, denoted by r[A]. In some situations increasing the selectivity of individual values enables to solve more conflicts with a single operation afterwards.
- Any of the remaining values from $dom(A)$ not contained in $r_t[A] \cup r[A]$ is a potentially necessary modification value, possibly to serve as a unique selection criterion in later stages of the algorithm. Thus, the actual value does not matter, as long as it is different from all other values currently used. We chose such a value using a random function. We call these values *Skolem constants*.

During TRANSIT, all Skolem constants are maintained within a separate list for each attribute.

Valid Modification Patterns

Let P(r) denote the set of patterns ρ that select at least one tuple from r, i.e., $\rho(r) \neq \emptyset$. If we regard a tuple t as a set of terms (A, t[A]) with one term for each attribute $A \in R$, P(r) is efficiently computable using a frequent itemset mining algorithm [2]. Very likely, this set contains pairs of patterns ρ_1, ρ_2 with $\rho_1 \neq \rho_2$ and $\rho_1(r) = \rho_2(r)$. Using each pattern in P(r) for enumeration of modification operations would therefore result in operations with equal effect. We avoid such redundancy by restricting P(r) to the set of *closed patterns*, denoted by $P_C(r)$. The following definition is taken from [3, 18]:

DEFINITION 12 (CLOSED PATTERN): Given a database r, a pattern ρ with $\rho(r) \neq \emptyset$ is a *closed pattern* for r if there does not exist a pattern $\rho' \supset \rho$ with $\rho'(r) = \rho(r)$.♦

A closed pattern ρ represents exactly those terms that occur within every tuple of ρ(r) (when viewing the tuples as sets of terms as described above). Following this definition there are no two patterns $\rho_1, \rho_2 \in P_C(r)$, $\rho_1 \neq \rho_2$, that select equal subsets of r.

LEMMA 4: Given a database r. For each pattern $\rho \in P(r)$ there exists a pattern $\rho' \in P_C(r)$ with $\rho(r) = \rho'(r)$.

PROOF: Omitted for space limitations.♦

Based on LEMMA 4 it is sufficient to use $P_C(r)$ extended by the empty pattern instead of P(r) as the set of valid modification patterns. We add the empty pattern to $P_C(r)$ in order to allow modifications of the complete database at once. We are able to use existing methods for mining closed itemsets like CHARM [23], CLOSET+ [21], or FARMER [9]. Within our implementation we currently use CHARM [23].

Filtering Invalid Operations

A modification operation has no effect if the modification term τ also occurs within the modification pattern. In this case, all selected tuples already possess the new value in the modified attribute. We remove these operations.

5. HEURISTICS

The described TRANSIT-algorithms are only applicable for small databases. For larger databases the search space of derivable databases is enormous due to the large number of closed patterns for each database. The maximum number of closed pattern for a database r is $2^{|r|}$-1, i.e., each non-empty subset of r defines a different closed pattern. While this is the worst case, there are at least |r| closed patterns for each database, as each tuple forms a closed pattern by itself. The large number of closed patterns results in an even larger number of modification operations and therefore derivable databases. Despite eliminating duplicates and pruning over 95% of the generated databases immediately (see Section 6.1) processing the remaining databases is still too expensive. Within this section we describe heuristics which do not necessarily find the exact solution but are able to handle databases of almost arbitrary size. We analyze the quality of the computed results in Section 6.2.

5.1 Greedy TRANSIT

A first simple heuristic is applying a greedy algorithm, called GREEDY-TRANSIT. Given a pair of databases r_o and r_t, the greedy algorithm first determines all databases derivable from the origin by a single modification operation. From those, the algorithm chooses the database with the smallest upper bound, i.e., the least number of conflicts with the target, as starting point for the next level. This database is denoted by r_s. For database r_s again all databases derivable by a single modification operation are generated and the database with the smallest upper bound is chosen as the next r_s. This is continued until the target database is reached. The algorithm returns a single transformer.

The described procedure ensures that the database chosen as the next starting point always has fewer conflicts with r_t than any of the previous databases. Therefore, neither cycles nor duplicated databases at different levels can occur. If a database is derivable by more than one modification operation from the current database, only the first operation, depending on the order of their enumeration, is returned within the final transformer. Using the database with the smallest upper bound follows the assumption that this database has the potential of reaching the destination first. The example in Figure 2 shows that the assumption is not always correct, as the resulting transformer has a length of four.

The main challenge for the greedy algorithm is the enumeration of all derivable databases for r_s. Enumerating the complete set of modification operations is infeasible for large databases. However, it is also not necessary. We avoid enumerating all those

modification operations whose resulting database is not a candidate for having the smallest lower bound by interleaving the mining for closed pattern and the generation of derivable databases. The algorithm for determining the update operation resulting in the next r_s is outlined in Figure 5. This is actually an extension of the CHARM algorithm being represented here by the function *nextPattern*() (*line* 3). The parameter `minsup` specifies the minimal support the returned patterns have to have in r_s.

Let $sup(\tau, \kappa)$ denote the number of tuples selected by τ that contain a conflict belonging in conflict group κ, i.e.,

$$sup(\tau, \kappa) = |\{t_s \mid t_s \in \tau(r_s) \wedge (\exists t_t \in r_t : t_s[ID] = t_t[ID] \wedge t_s[attr(\kappa)] \neq value(\kappa) \wedge t_s[attr(\kappa)] = value(\kappa))\}|$$

This number can easily be derived while scanning the databases to determine the initial set of terms for closed pattern mining. We further define $sup(\rho, \kappa)$ as the minimum $sup(\tau, \kappa)$ for all the terms $\tau \in \rho$. This is the maximal number of conflicts the pattern can potentially solve from conflict group κ. Every time the pattern mining algorithm identifies a new closed pattern ρ, we enumerate all modification operations using ρ that are able to reduce the number of conflicts more than the currently best operation ψ_{max}. These are modification operations having a modification term that equals one of the current conflict groups from $\kappa \in K(r_s, r_t)$ with $sup(\rho, \kappa)$ being above the current maximum in reduction of the upper bound. Any other modification term does not have the potential for reducing the current number of conflicts. Whenever an operation is enumerated that performs better than ψ_{max} we are able to increase `minsup` and thereby avoid further enumeration of patterns that cannot solve more conflicts than the new ψ_{max}.

GREEDY-TRANSIT calls *greedyNext*() for each database r_s. The result of *greedyNext*() can by empty as we use two as the initial `minsup`. In this case we solve one of the existing conflicts randomly using the tuple where the conflict occurs as closed pattern.

5.2 Approximation of the Update Distance

Another heuristic is based on solving the conflicts within each conflict group independently. We use the sum of necessary operations for conflict solution of each conflict group as an approximation of the update distance. This is called the *group solution cost*. The approximation completely disregards the possible impact that the modification of values for some of the tuples may have on solving conflicts for other tuples.

```
1  greedyNext(r_s, r_t) {
2    ψ_max := ⊥; minsup := 2;
3    while (ρ = nextPattern(minsup)) {
4      for each κ ∈ K(r_s, r_t) {
5        if (sup(ρ, κ) ≥ minsup) {
6          r_c := (κ,ρ)(r);
7          if (Δ_R(r_s, r_t)- Δ_R(r_c, r_t) ≥ minsup) {
8            ψ_max := (κ,ρ);
9            minsup := (Δ_R(r_s, r_t)- Δ_R(r_c, r_t))+1;
10   }}}}
11   return ψ_max;
12 }
```

Figure 5: Determine the modification operation that results in a database with the smallest upper bound.

Determining the minimal number of modification operations necessary to solve the conflicts within a conflict group still is expensive, as shown in Section 6.1. Therefore, we restrict the set of valid modification operations for approximating the update distance by considering only operations that reduce the number of conflicts and that do not introduce new conflicts or alter conflicting values in existing conflicts. The entire algorithm, called TRANSIT-APPROX can be found in [17] but is omitted here due to space limitations.

The group solution cost may also be used as a replacement for the lower bound within the algorithms TRANSIT-BFS and TRANSIT-DFS. We thus might miss the exact solution. However, in all our experiments this heuristic computed the exact solution. The corresponding algorithms are called TRANSIT-BFS (GS) and TRANSIT-DFS (GS), respectively.

6. EXPERIMENTS

Within this section we discuss some of the results of our experiments applying the defined algorithms on different pairs of databases. We implemented all algorithms using Java™ J2SE 5.0. The code and experimental data is available from the authors.

6.1 Branch and Bound Algorithms

Computing minimal update sequences using the algorithms described in Section 4 is feasible only for small databases. We used several small synthetic databases pairs in our experiments. In this paper, we only present the result for the databases of Figure 2 to give an idea of the complexity of the algorithms. We refer the interested reader to [17] for further results.

TRANSIT-BFS and TRANSIT-DFS

The necessary computation to determine the set of minimal update sequences that transform r_1 into r_2 is shown in Table 1a. The first two columns list the number of databases processed and modification operations executed for building the transition graph. We also list are the overall number of databases added to the graph together with the number of databases generated as duplicates.

Table 1a shows a huge difference between the number of modification operations executed and the number of databases added to the graph or being identified as duplicates for both approaches. This observation indicates that the majority of the generated databases are pruned due to their upper and lower bounds. The number of pruned databases is between 80-95% for TRANSIT-BFS and above 95% for TRANSIT-DFS. Still, due to the enormous number of modification operations and databases derivable at each node of the transition graph execution time and memory requirements prevent an application of the algorithms for larger databases.

The depth-first approach is inferior for the databases of Figure 2. The optimal solution requires the insertion of conflicts at first. Thus, the depth-first approach generates a first solution that is longer that the actual update distance. It therefore tests several databases at distance levels above the actual update distance. This is reflected by comparing the number of databases added and tested in columns 1 and 3 of Figure 2. For TRANSIF-DFS more databases are tested than added to the graph, due to those databases that are added once but tested several times at decreasing distance levels.

Table 1: Experimental Results of the branch and bound algorithms

	DBs Tested	Ops Executed	DBs Added	Intra-Dupl.	Inter-Dupl.
a)	TRANSIT-BFS vs. TRANSIT-DFS				
BFS	279	38,006	3,026	2,832	968
DFS	4,275	603,971	4,204	4,417	4,483
b)	TRANSIT-BFS (GS) vs. TRANSIT-DFS (GS)				
BFS(GS)	4	499	27	2	0
DFS(GS)	3	402	23	2	0

Comparing the numbers of databases added and tested for TRANSIT-BFS shows the general problem of this approach: a large number of databases that are added to the graph are never processed afterwards. Our experiments show that pruning of databases once added to the graph is not very effective for both approaches. Therefore, the number of databases in the graph grows linearly with the number of added databases. We observe that in general the memory requirement for the breadth-first approach is higher than that for the depth-first approach.

TRANSIT-BFS (GS) and TRANSIT-DFS (GS)

Table 1b shows the necessary effort to determine the set of minimal transformers when using the group solution cost as the lower bound for both branch and bound algorithms. In our experiments, this heuristic always computes the correct update distance, but does not find all minimal update sequences.

Compared to the numbers in Table 1a, the effort regarding databases tested and added is significantly lower for TRANSIT-BFS (GS) and TRANSIT-DFS (GS). As a downside, the computation cost may increase due to the computation of the group solution cost Therefore, despite the extremely high accuracy the computation cost (and not the memory requirements) prevents us from applying this heuristic to larger databases. Finding an efficient method for group solution cost computation would yield in a significant runtime improvement. We therefore consider improving the efficiency of group selection cost computation as future work.

6.2 Accuracy of GREEDY-TRANSIT and TRANSIT-APPROX

While computing the exact solution is only possible for small databases the greedy algorithm and the approximation allow computation of update sequences and distance for large databases. In order to assess the accuracy of GREEDY-TRANSIT and TRANSIT-APPROX we used a database of 10 attributes and 100 tuples and modified it using arbitrary update sequences of length between 5 and 50. We then computed the update distance between the original and the resulting database using both algorithms. The results are shown in Figure 6. The shown values are averaged over ten runs. The dark area above the lower bound highlights the location of the exact solution between the lower bound and the length of the sequences that generated the databases. The greedy approach and the approximation are both surprisingly accurate for short update sequences. For longer update sequences the accuracy decreases but remains in reasonable bounds. Overall, the GREEDY-TRANSIT outperforms TRANSIT-APPROX in accuracy.

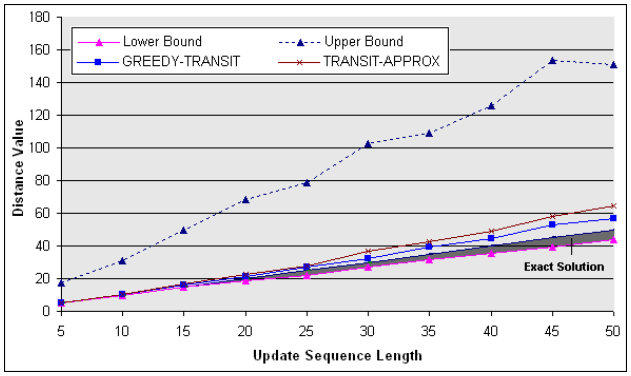

Figure 6: Comparing the accuracy of GREEDY-TRANSIT and TRANSIT-APPROX

The accuracy of GREEDY-TRANSIT decreases, as the number of conflicts increases. We used a second database of 20 attributes and about 800 tuples and modified it using update sequences as described above. Figure 7 shows the resulting greedy update distances when limiting modification operations within the sequences to such whose patterns select at least 10, 20, or 50 tuples each. The number of conflicts introduced by theses sequences increases as the selectivity of the patterns increases. This resulted in database pairs with over 3,500 conflicts between them. The decrease in accuracy shown by GREEDY-TRANSIT is even larger when using TRANSIT-APPROX.

Execution time of the branch and bound approaches lies between a few milliseconds and up to four minutes for the databases we considered. The execution time increases dramatically for larger databases. GREEDY-TRANSIT had execution times between two seconds for short sequences and one minute for long sequences in the experiments of Figure 7. We also applied this algorithm on the protein structure databases Columba and OpenMMS, having over 20,000 tuples each and nearly 10,000 conflicts between them. The resulting update sequence contained over 10,000 update operations and computation took more than 24 hours. The result shows the clear disadvantage of the greedy approach as more than 97% of the operations in the sequences solved less than ten conflicts. Still, by interpreting the operations at the start of the sequence, we discovered update operations that describe the commonly known differences between the databases like usage of different value representations or vocabularies in some of the attributes.

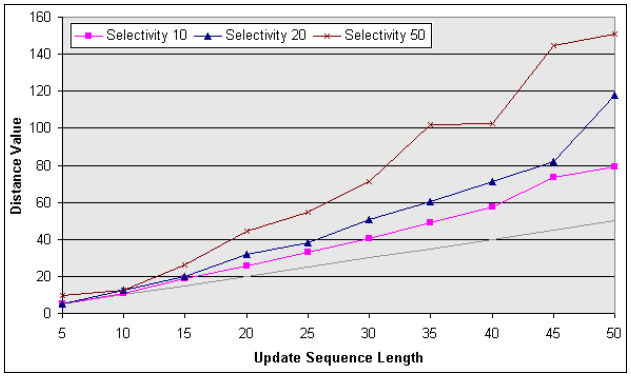

Figure 7: Accuracy of GREEDY-TRANSIT for update sequences with different pattern selectivity

7. CONCLUSION & OUTLOOK

We described several algorithms for determining update sequences of SQL-like update operations that transform one database into another one. If conflicts between two databases are due to systematic manipulation, the operations within update sequences are valuable to domain experts interested in solving the conflicts. Minimal sequences may also be used as retrospective documentation of manipulations performed on a given database. Due to the complexity of the problem, exact solutions are only feasible for very small databases. Therefore, we investigated several heuristics that, in our experiments, found near-optimal solutions and are applicable also to larger data sets.

In out current research work we investigate several directions. A major challenge is to reduce the memory requirements of our algorithms. For instance, instead of holding entire databases in main memory, one could represent a database by its generating operations plus the hash key. This reduces memory consumption while increasing the execution time for duplicate checks. Another considerable cost factor is the necessary computation of closed patterns to find all valid modifications. However, deriving the set of closed patterns for a database from the set of closed patterns from its predecessor database using some incremental approach could be highly advantageous since both database vary only very little.

We also consider several extensions to our approach. First, enhancing the expressiveness of update operations, including modifications like `SET A = f(A)` as described in [11], would be very important; yet the cost of finding such functions is probably prohibitive. Second, assuming that two contradicting databases have been derived from a single ancestor database, it is natural to ask the following question (studied in biology under the term phylogenetics): Given two database r_1 and r_2, compute the database r whose update distance to r_1 plus its update distance to r_2 is minimal. Thus, we could actually reconstruct the original database.

8. REFERENCES

[1] M. Arenas, L. Bertossi, J. Chomicki. *Consistent Query Answers in Inconsistent Databases*. Proc. ACM Symposium on Principles of Database Systems (PODS), Philadelphia, Pennsylvania, 1999.

[2] R. Agrawal and R. Srikant. *Fast Algorithms for Mining Association Rules*, Proc. Int. Conf. On Very Large Data Bases (VLDB), Santiago de Chile, Chile, 1994.

[3] J. Bayardo, Jr. *Efficiently mining long patterns from databases*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, Washington, United States, 1998, 85 – 89.

[4] H.M.Berman, J.Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, P.E. Bourne. *The Protein Data Bank*. Nucleic Acids Research, Vol. 28(1), 2000, 235-242

[5] T.N. Bhat, et al. *The PDB data uniformity project*, Nucleic Acid Research, Vol. 29(1), 2001, 214-218.

[6] H. Boutselakis, et al. *E-MSD: the European Bioinformatics Institute Macromolecular Structure Database*. Nucleic Acid Research, Vol. 31(1), 2003, 458-462.

[7] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. *A Cost-Based Model and Effective Heuristic for Repairung Constraints by Value Modifications*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Baltimore, Maryland, United States, 2005.

[8] S. Chawathe, H. Garcia-Molina. *Meaningful change detection in structured data*. Proc. ACM SIGMOD Int. Conf. on Management of Data Tucson, Arizona, May 1997.

[9] G. Cong, A.K.H. Tung, X. Xu, F. Pan, and J. Yang. *FARMER: finding interesting rule groups in microarray datasets*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Paris, France, 2004, 143 – 154.

[10] P. Dadam, V.Y. Lum, H.-D. Werner. *Integration of Time Versions into a Relational Database System*. Proc. Int. Conf. on Very Large Data Bases, Singapore, 1984, p.p. 509-522

[11] W. Fan, H. Lu, S.E. Madnick, and D. Cheung. *Discovering and reconciling value conflicts for numerical data integration*. Information Systems, Vol. 26, 2001, 635-656.

[12] M.A. Hernandez, S.J. Stolfo. *The merge/purge problem for large databases*. Proc of ACM SIGMOD Int. Conf. On Management of Data, San Jose, California, 1995

[13] W. J. Labio and H. Garcia-Molina. *Efficient Snapshot Differential Algorithms for Data Warehousing*. Proc. Int. Conf. On Very Large Data Bases (VLDB), Bombay, India, September 1996, pp. 63-74

[14] V.I. Levenshtein. *Binary codes capable of correcting insertions and reversals*. Sov. Phys. Dokl., 10:707-10, 1966.

[15] A.E. Monge, C.P. Elkan. *An efficient domain-independent algorithm for detecting approximately duplicate database tuples*. Proc. SIGMOD Workshop on Data Mining and Knowledge Discovery, 1997

[16] H. Müller, U. Leser, and J.-C. Freytag. *Mining for Patterns in Contradictory Data*, Proc. SIGMOD Int. Workshop on Information Quality for Information Systems (IQIS'04), Paris, France, 2004.

[17] H. Müller, J.-C. Freytag, and U. Leser. *On the Distance of Databases*, Technical Report, HUB-IB-199, March 2006.

[18] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. *Discovering Frequent Closed Itemsets for Association Rules*. Lecture Notes in Computer Science, Vol. 1540, 1999, 398-416.

[19] K. Rother, H. Müller, S. Trissl, I. Koch, T. Steinke, R. Preissner, C. Frömmel, U. Leser. *COLUMBA: Multidimensional Data Integration of Protein Annotations*, Int. Workshop on Data Integration in Life Sciences (DILS 2004), Leipzig, Germany, 2004.

[20] G. Vossen. *Data Models, Database Languages and Database Management Systems*. Addison-Wesley Publishers, ISBN 0-201-41604-2, 1991

[21] J. Wang, J. Han, and J. Pei. *CLOSET+: searching for the best strategies for mining frequent closed itemsets*. Proc. ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Washington, D.C., 2003, 236 – 245.

[22] W. Winkler. *The state of record linkage and current research problems*. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1999.

[23] M.J. Zaki and C.-J. Hsiao. *CHARM: An efficient algorithm for closed itemset mining*. In Proc. of the Second SIAM Int. Conference on Data Mining, Arlington, VA, 2002.