On the Distance of Databases

Heiko Müller, Johann-Christoph Freytag, Ulf Leser
Humboldt-Universität zu Berlin zu Berlin,
10099 Berlin, Germany
{hmueller, freytag, 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 set-oriented insert, delete and modification operations necessary to transform one database into the other. 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 are a proper representation of systematic errors, thus giving valuable clues to domain experts responsible for conflict resolution.

Contents

1 EDIT DISTANCE OF DATABASES 3

2 TRANSFORMERS FOR PAIRS OF DATABASES 7

2.1 CONTRADICTING DATABASES 7
2.2 UPDATE OPERATIONS 8
2.3 MINIMAL SEQUENCES OF UPDATE OPERATIONS 9

3 DISTANCE MEASURES FOR DATABASES 9

3.1 THE RESOLUTION DISTANCE 10
3.2 THE UPDATE DISTANCE 10

4 TRANSIT - MINIMAL TRANSFORMERS FOR DATABASES 12

4.1 SEARCH SPACE EXPLORATION 12
4.1.1 TRANSITION GRAPH 13
4.1.2 DUPLICATE DETECTION 14
4.1.3 REPRESENTATION OF DATABASES 16
4.1.4 PRUNING 16
4.1.5 BREADTH-FIRST ALGORITHM 16
4.1.6 DEPTH-FIRST ALGORITHM 18
4.2 ENUMERATION OF VALID MODIFICATION OPERATIONS 19
4.2.1 THE SET OF VALID MODIFICATION TERMS 19
4.2.2 THE SET OF VALID SELECTION PATTERNS 20
4.2.3 FILTERING VALID MODIFICATION OPERATIONS 21
4.3 HANDLING DELETE AND INSERT OPERATIONS 22
4.3.1 DELETE OPERATIONS AS SPECIAL CASE OF MODIFICATIONS 22
4.3.2 HANDLING ARBITRARY SEQUENCES OF UPDATE OPERATIONS 22

5 HEURISTICS AND PROBLEM VARIATIONS 22

5.1 A CLASSIFICATION OF MODIFICATION OPERATIONS 23
5.2 GREEDY TRANSIT 24
5.3 APPROXIMATION OF THE UPDATE DISTANCE 25

6 EXPERIMENTAL RESULTS 27

6.1 ALGORITHMS FOR FINDING EXACT SOLUTIONS 27
6.2 ACCURACY OF THE HEURISTIC APPROACHES 30
6.2.1 DIFFERENT OPERATION CLASSES 30
6.2.2 TRANSIT-DFS (GS) AND TRANSIT-BFS (GS) 32
6.2.3 GREEDY-TRANSIT AND TRANSIT-APPROX 33

7 FURTHER DISTANCE MEASURES 35

8 RELATED WORK 38

9 CONCLUSIONS & OUTLOOK 39

LITERATURE 41

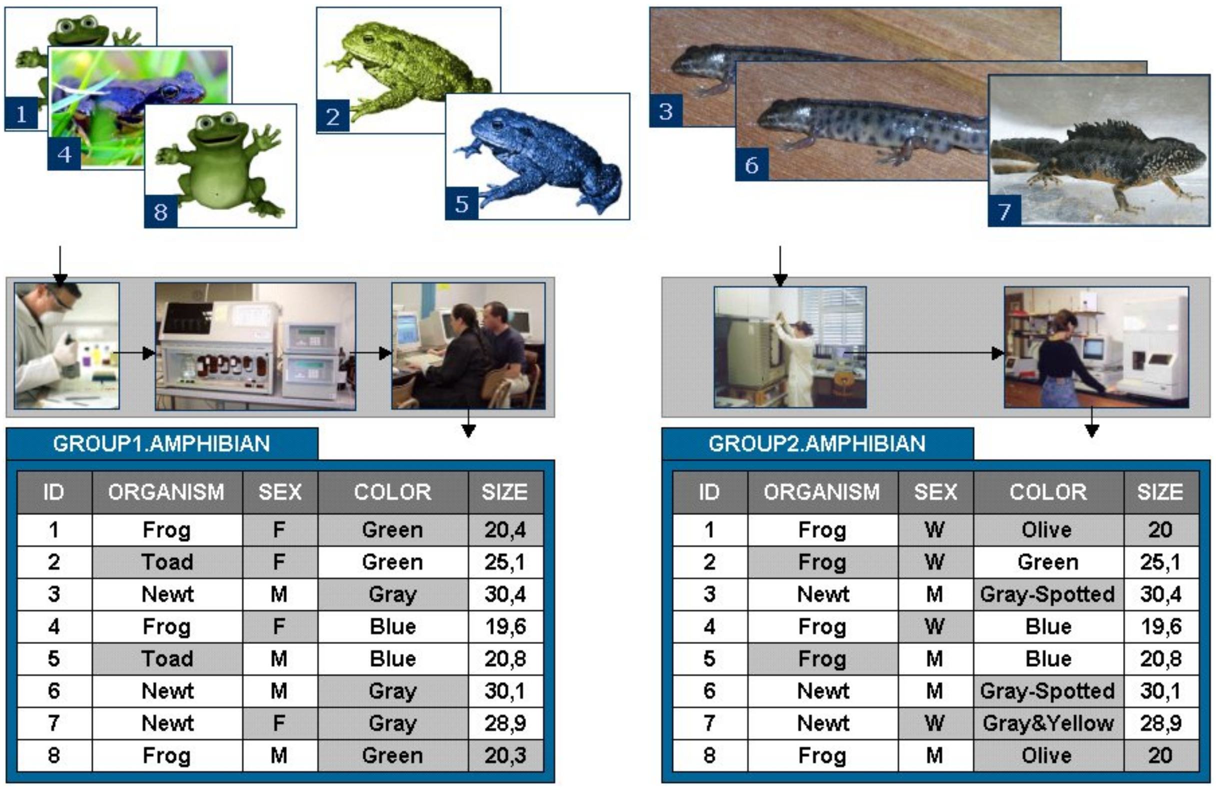
1 Edit Distance of Databases

Today, many databases are generated with overlaps in their sets of represented real-world entities. There are various reasons for these overlaps:

- ***Replication of data sources***: In e-commerce, for example, many business-critical services demand high availability and a low latency. Therefore, web services and their data are replicated at geographically distributed sites to improve the performance of these services [GDN+03]. A common example from life science research are the three databases GenBank, EMBL, and DDBJ within the International Nucleotide Sequence Database Collaboration (INSDC) [INSDC]. These databases all manage the same set of DNA sequences, but share the burden of submission handling and query answering. Replication of data sources is also common for mobile devices to gain data access independently of the availability of a network connection.
- ***Independent production of data***: Data representing a common set of entities or individuals is often collected and maintained independently by different groups or institutions. For example, within a company the accounting department and the personnel department maintain overlapping lists of the employees in the company. There may also exists overlaps within the customer databases of different companies. In the area of scientific 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.
- ***Data integration and data warehousing***: There is a plethora of data integration and data warehousing projects world-wide (see [Ziegler] for a current listing). Within these projects data is copied from sources, possibly transformed and manipulated for data cleansing, and stored in an integrated data warehouse. Data integration results in overlaps between the originals and the integrated databases.

Whenever overlapping data is 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 at the different institutions. For example, the three protein structure databases OpenMMS [BBF+01], MSD [BDF+03], and Columba [RMT+04] are all copies of the Protein Data Bank PDB [BWF+00]. However, due to different cleansing strategies, these copies vary substantially. In OpenMMS the focus is on completing and updating literature references, while in MSD the focal point of data cleansing is the standardization of used vocabularies. Thus, a biologist is faced with conflicting copies of the same set of real world objects and the problem of solving these conflicts to produce a consistent view of the data.

Learning about the reasons that led to inconsistencies is a valuable means in the task of conflict resolution. Many inconsistencies are highly systematic, caused by the usage of different controlled vocabularies, different measurement units, different abbreviations, or by misinterpretations during experimental analysis. Knowledge about such systematic deviations can be used to assess the individual quality of database copies for conflict resolution. Figure 1shows an example of overlapping databases representing fictitious results of two research groups examining the same set of amphibians with differing data production workflows. The contradicting values are highlighted by shaded cells. Unfortunately, usually only the databases are visible without any additional knowledge about the data generation or manipulation process that lead to the occurring differences.



GROUP1.AMPHIBIAN

ID	ORGANISM	SEX	COLOR	SIZE
1	Frog	F	Green	20,4
2	Toad	F	Green	25,1
3	Newt	M	Gray	30,4
4	Frog	F	Blue	19,6
5	Toad	M	Blue	20,8
6	Newt	M	Gray	30,1
7	Newt	F	Gray	28,9
8	Frog	M	Green	20,3

GROUP2.AMPHIBIAN

ID	ORGANISM	SEX	COLOR	SIZE
1	Frog	W	Olive	20
2	Frog	W	Green	25,1
3	Newt	M	Gray-Spotted	30,4
4	Frog	W	Blue	19,6
5	Frog	M	Blue	20,8
6	Newt	M	Gray-Spotted	30,1
7	Newt	W	Gray&Yellow	28,9
8	Frog	M	Olive	20

Figure 1: Contradicting data sources resulting from different data production workflows while examining the same set of amphibians**.**

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 their understanding. Evaluated by a domain expert, the patterns can be utilized to assess the correctness of conflicting values and therefore for conflict resolution. In [MLF04] we proposed the adaptation of existing data mining algorithms to find such patterns.

In this paper, we develop a different approach for finding regularities in contradicting databases: The detection of minimal update sequences. Figure 2 shows such a minimal sequence of six update operations (using SQL syntax) transforming the first data source of Figure 1 into the second. Each operation may act as a description of potential systematic difference in data production that lead to the occurring conflicts.

```
(1)   UPDATE GROUP1.AMPHIBIAN
      SET COLOR = ‘Olive’
      WHERE ORGANISM = ‘Frog’
      AND COLOR = ‘Green’

(2)   UPDATE GROUP1.AMPHIBIAN
      SET SIZE = 20
      WHERE COLOR = ‘Olive’

(3)   UPDATE GROUP1.AMPHIBIAN
      SET COLOR = ‘Grey-Spotted’
      WHERE ORGANISM = ‘Newt’
      AND SEX = ‘M’
      AND COLOR = ‘Grey’

(4)   UPDATE GROUP1.AMPHIBIAN
      SET COLOR = ‘Grey&Yellow’
      WHERE ORGANISM = ‘Newt’
      AND SEX = ‘F’
      AND COLOR = ‘Grey’

(5)   UPDATE GROUP1.AMPHIBIAN
      SET ORGANISM = ‘Frog’
      WHERE ORGANISM = ‘Toad’

(6)   UPDATE GROUP1.AMPHIBIAN
      SET SEX = ‘W’
      WHERE SEX = ‘F’
```

Figure 2: A sequence of update operations transforming a given data source into another one.

For example, Group2 does not differentiate between organisms toads and frogs (5), both groups use different representation for the gender female (6) and Group2 uses variations of color gray for male and female newts (3) and (4).

Our idea of using minimal update sequences as descriptions for database differences is best explained by analogy to the usage of the string edit distance [Lev65] in biological sequence analysis (see Figure 3). The DNA sequence of a gene is a string over a four letter alphabet. To learn about the function of a specific gene in a specific species, biologists search for evolutionary related genes of known function in other species. This evolutionary relatedness (or distance) is proportional to the number of evolutionary events that have occurred to the sequence of a common ancestor, deriving the observed sequences, which in turn is proportional to the number of evolutionary events that would be necessary to turn one gene into another. Using a simple model of evolution encompassing only changes, deletions, and insertions of single bases (i.e., characters of the sequence), the number of evolutionary events is measured by the edit distance between two gene sequences, i.e., the minimal number of edit operations (or evolutionary events) that transform one string into the other.

Similarly, we consider updates, insertions, and deletions of tuples as the fundamental operations for the manipulation of data stored in relational databases. Thus, to assess the "evolutionary relationship" of two databases, we propose to use the minimal number of such operations that turn one databases 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 (see for example Figure 2). Following the "Occam's Razor" principle, we conclude that the simplest explanations are also the most likely. Minimal update sequences therefore give valuable clues on what has happened to a databases to make it different from its original state. The update distance is a semantic distance measure, as it is inherently process-oriented in contrast to purely syntactic measures such as counting differences.

In this paper, we present several 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 the 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 convincing results in all examples we considered.

To give an idea of the complexity of the problem, consider the databases of Figure 4. Clearly, their update distance can be determined by enumerating update sequences of increasing length until one sequence is found that implements all necessary changes. This would generate 294,998 intermediate states.

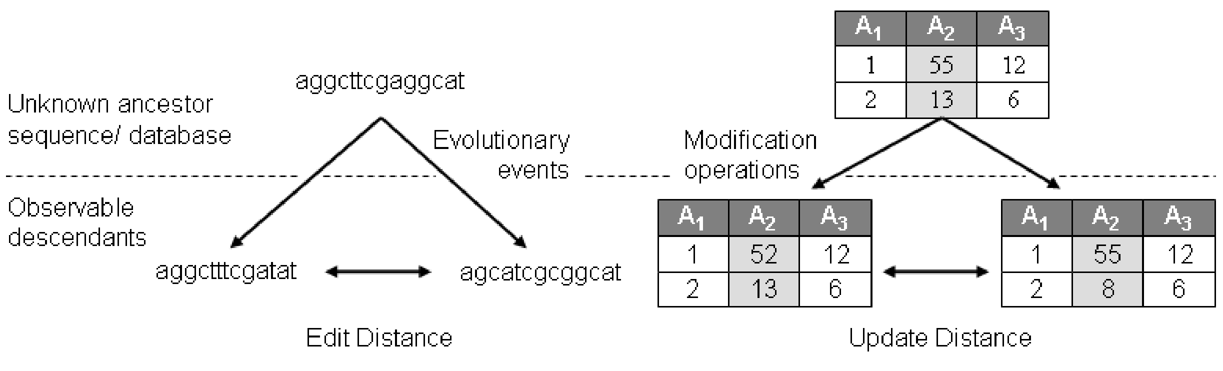


Figure 3: Edit distance of biological sequences (left) versus update distance of databases (right).

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 $A_3 = 0$
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 $A_1 = 3$

b) 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 $A_1 = 8$

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 4: An example for the need to introduce conflicts in order to find an optimal solution.

An intuitive idea to prune the search space would be to use a greedy strategy, i.e., to select at each stage the operation that solves the most conflicts. This reduces the number of generated intermediate states to 42 for the databases of Figure 4. The shortest sequence found using such an approach has four elements (Figure 4a), although the update distance between databases r_1 and r_2 is only three (Figure 4c). Another pruning idea might be to avoid modification operations that introduce new conflicts. This results in only 32 generated intermediate states. However, using this heuristic worsens the result, as now the shortest sequence is of length five (Figure 4b). 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 4c temporarily increases the total number of conflicts, but this is compensated in later operations that are now able to solve more conflicts within one statement.

The paper is structured as follows: Section 2 defines minimal update sequences that transform a given database into another database. Also introduced is the necessary vocabulary, i.e., update operations and sequences of update operations, used throughout this paper. In Section 3 we give a formal definition of the update distance for overlapping databases and derive an upper and lower bound, which are important for optimization. Section 4 outlines algorithms for calculating minimal update sequences transforming a given database into another. In Section 5 we discuss heuristics and problem variations to improve the efficiency of the algorithm. Section 6 shows the results of experiments we have performed after implementing the described algorithms and heuristics. We define two additional distance measures in Section 7. Section 8 discusses related work. We conclude in Section 9.

2 Transformers for Pairs of Databases

In the following, we build the necessary vocabulary for the definition of the update distance in the next section. Section 2.1 defines matches, conflicts and uncertainties between two databases. Section 2.2 introduces the types of basic operations we assume as possible updates. Section 2.3 defines sequences of these operations in order to transform a given database into another.

2.1 Contradicting Databases

The data sources within this paper are relational databases as defined in [Cod70]. These databases consist of a single relation r and they all follow the relational schema $R(A_1, \ldots, A_n)$. Each attribute $A \in R$ is associated with a domain of possible values, denoted by *dom*(A). Without loss of generality we assume $dom(A) = \mathbb{N}$ for all attributes $A \in R$. Tuples are denoted by t and values corresponding to attribute A by t[A]. We assume the existence of a primary key constraint for schema R. Without loss of generality we assume A_1 to be the primary key attribute. We will use ID as synonym for attribute A_1. The primary key represents the unique object identifier for finding duplicate tuples between databases. A single database is therefore free of duplicates. We use t{j} to refer to the tuple with primary key value j, $j \in \mathbb{N}$.

A pair of tuples from databases r_1 and r_2 is called a *matching pair* if they possess identical primary key values. The *set of all matching pairs* between databases (i.e., relations) r_1 and r_2 is denoted by $M(r_1, r_2)$, i.e.,

$$M(r_1, r_2) = \{(t_1, t_2) \mid (t_1, t_2) \in r_1 \times r_2 \wedge t_1[ID] = t_2[ID]\}$$

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 $M(r_1, r_2)$. There might also be tuples in r_1 and r_2 without a matching partner in the other database. These tuples are called *unmatched*. The set of tuples from database r_1 that are unmatched by tuples from r_2 is denoted by $U(r_1, r_2)$, i.e.,

$$U(r_1, r_2) = \{t_1 \mid t_1 \in r_1 \wedge \neg\exists\, (t_2 \in r_2 \wedge t_1[ID] = t_2[ID])\}$$

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 s*et 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.,

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

A pair of databases r_1 and r_2 is called *contradicting*, if there exists at least one conflict between them, i.e., $C(r_1, r_2) \neq \emptyset$. We call the databases *different* if they are contradicting or there exist unmatched tuples between them, i.e., $C(r_1, r_2) \neq \emptyset \vee U(r_1, r_2) \neq \emptyset \vee U(r_1, r_2) \neq \emptyset$. As an example, consider the databases from Figure 4. The set of matching pairs contains ten elements. There are no unmatched tuples in either one of them and there are five conflicts, i.e., $C(r_1, r_2) = \{((t_1\{4\}, t_2\{4\}), A_3), ((t_1\{5\}, t_2\{5\}), A_3), ((t_1\{6\}, t_2\{6\}), A_3), ((t_1\{7\}, t_2\{7\}), A_3), ((t_1\{8\}, t_2\{8\}), A_3)\}$.

2.2 Update Operations

Update operations are used to modify existing databases. They can be considered as functions that map databases onto each other. Let $\Re(R)$ denote the infinite set of databases following schema R that satisfy the primary key constraint. An update operations ψ is then defined as a mapping ψ: $\Re(R) \rightarrow \Re(R)$. For relational databases there are three types of basic update operations, namely insert, delete, and modify [Vos91]. An insert operation creates a new tuple. A delete operation removes a set of tuples satisfying a given selection criteria. A modification operation changes the value for an attribute within a set of tuples satisfying a given selection criteria. Before we define the update operations and sequences in detail we introduce the following concepts to fix the expressiveness of the operations.

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

A term can be interpreted as a Boolean-function on tuples. A tuple t satisfies τ, denoted by τ(t) = *true*, if t[*attr*(τ)] = *value*(τ). By τ(r) we denote the set of tuples from r which satisfy τ. We say that the tuples in τ(r) are selected by τ. Terms are combined to patterns acting as selection criteria in update operations.

DEFINITION 3 (PATTERN): A *pattern* ρ over schema R is a set of terms over schema R. We only consider patterns which do not contain different terms with equal attributes, i.e.,

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

A tuple t satisfies ρ, denoted by ρ(t) = *true*, if it satisfies each term within ρ. A pattern is therefore a conjunction of terms. An empty pattern is satisfied by each tuple of a database. Similar to the definitions above, ρ(r) denotes the set of tuples satisfying ρ. We say that ρ selects the set of tuples ρ(r) from the database r.

DEFINITION 4 (UPDATE OPERATION): An *update operation* ψ over schema R is a mapping $\psi : \Re(R) \rightarrow \Re(R)$. We differ between three types of update operations:

- The *insert operation*, denoted by ψ_ι, is a n-tuple $(\tau_1, \ldots, \tau_n)$. It contains exactly one term for each of the attributes A_i from R. It adds a new tuple t_{new} to r, with $t_{new}[A_i]$ = value(τ_i) for $1 \leq i \leq n$. The insert operation is therefore also denoted by $\psi_\iota(t_{new})$. If there already exists a tuple t in r with t[ID] = t_{new}[ID], the database remains unchanged. Otherwise, the result of ψ_ι(r) is $r \cup \{t_{new}\}$.
- The *delete operation*, denoted by ψ_δ, is defined by a single pattern ρ. It removes all tuples from a relation, that satisfy the pattern ρ, i.e., $\psi_\delta(r) = r / \rho(r)$.
- The *modification operation*, denoted by ψ_μ, is a term-pattern pair (τ, ρ). We exclude key attributes from being modified. Therefore, *attr*(τ) is element of R / ID. A modification operation modifies all tuples within a relation, which satisfy ρ. For these tuples, the value for attribute *attr*(τ) is set to by *value*(τ).♦

Given a modification operation $\psi_\mu = (\tau, \rho)$, we refer to τ as the *modification term*, to *value*(τ) as the *modification value*, to *attr*(τ) as the *modified attribute*, and to ρ as the *modification pattern*. Note that there not necessarily exists a reverse operation for each modification operation. For example, the operation $\psi_\mu = ((A_2, 7), \{(A_3, 1)\})$ sets the value for attribute A_2 to 7 for the tuples t{1}, …, t{8} when applied to database r_1 from Figure 4. We need at least six modification operations to undo this single operation. There is also no single reverse operation for delete operations that delete more than one tuple.

2.3 Minimal Sequences of Update Operations

We now have all the tools at hand to define minimal sequences of update operations.

Definition 5 (Update Sequence): An *update sequence* $\Psi = <\psi_1, \ldots, \psi_k>$ is an ordered list of update operations. Applied on a database r_1, an update sequence *generates* (or *derives*) a database $r_2 = \Psi(r_1)$ by executing the update operations in given order on r_1, i.e., $\Psi(r_1) = \psi_k(\ldots(\psi_1(r_1))\ldots)$.♦

The databases which are generated by the update operations of an update sequence while transforming r_1 into r_2 are called *intermediate states*. Obviously, the order of operations within an update sequence is important. For example, the 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 4. 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 databases 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 $|\Psi|$. Figure 4 lists three update sequences of different length, which are transformers for the databases r_1 and r_2.

Definition 6 (Minimal Transformer): An update sequence Ψ is called a *minimal transformer* for a pair of databases r_1 and r_2, if $\Psi(r_1) = r_2$ and there does not exists another transformer Ψ' with $\Psi'(r_1) = r_2$ and $|\Psi'| < |\Psi|$.♦

There may be several minimal transformers for a pair of databases r_1 and r_2. The set of all minimal transformers for r_1 and r_2 is denoted as $T(r_1, r_2)$. Figure 5 shows two minimal transformers (in SQL-like notation) that transform r_1 into r_2 and r_2 into r_1, respectively.

r_1	A_1	A_2	A_3	A_4
	1	2	1	1
	2	1	2	1
	3	1	2	0
	4	1	2	1

r_2	A_1	A_2	A_3	A_4
	1	2	1	3
	2	2	2	3
	3	2	2	3
	4	2	2	3

$\Psi(r_1) = r_2$	$\Psi(r_2) = r_1$

UPDATE SET $A_4 = 3$
UPDATE SET $A_2 = 2$ WHERE $A_2 = 1$
	UPDATE SET $A_2 = 1$ WHERE $A_3 = 2$
	UPDATE SET $A_4 = 1$
	UPDATE SET $A_4 = 0$ WHERE $A_1 = 3$

UPDATE SET $A_2 = 2$
UPDATE SET $A_4 = 3$
	UPDATE SET $A_4 = 0$ WHERE $A_1 = 3$
	UPDATE SET $A_4 = 1$ WHERE $A_4= 3$
	UPDATE SET $A_2 = 1$ WHERE $A_3 = 2$

Figure 5: The set of minimal transformers for a pair of databases.

3 Distance Measures for Databases

Within this section we define distance measures for databases to quantify their similarity. Such a measure is represented by a distance function, which assigns a non-negative value to a pair of databases, with a smaller value, i.e., a shorter distance, reflecting a greater similarity. Similar to existing distance measures for strings, which rely on the edit operations insert, delete, and replace, we use sequences of update operations in our definitions.

3.1 The Resolution Distance

An obvious distance measure for a pair of databases is the total number of differences between them. This number is given by the sum of unmatched tuples and conflicts between these databases. Such a distance measure reflects the maximal number of necessary update operations for transforming one of the databases into each other.

DEFINITION 7 (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 sum of the number of unmatched tuples in either database and the number of conflicts between the databases, i.e.,

$$\Delta_R(r_1, r_2) = |U(r_1, r_2)| + |U(r_2, r_1)| + |C(r_1, r_2)|.\blacklozenge$$

For the databases r_1 and r_2 in Figure 5, the resolution distance is 7 and it equals the number of conflicts between the databases r_1 and r_2. It follows, that $\Delta_R(r_1, r_2) = \Delta_R(r_2, r_1)$, as $C(r_1, r_2)$ equals $C(r_2, r_1)$. The resolution distance is not a metric, as the triangle inequality $\Delta_R(r_1, r_2) + \Delta_R(r_2, r_3) \geq \Delta_R(r_1, r_3)$ odes not hold. For example, a tuple occurring within r_1 and r_3 but not in r_2 counts only twice on the left side of the inequality but potentially $(|R| - 1)$-times on the right side, because there may occur a conflict within every non-key attribute of the corresponding matching pair.

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: In order to transform r_1 into r_2 we have to (i) remove the tuples from r_1 without a matching partner in r_2, (ii) solve the conflicts within the matching pairs, and (iii) insert those tuples which exist in r_2 but not r_1. Due to the primary key property every single tuple t from database r is individually selectable by a pattern $\rho = \{(ID, t[ID])\}$. As the primary key is unchangeable this is always true for any existing tuple. The deletions are accomplished using a single delete operation for every unmatched tuple in r_1, i.e., for every tuple in $U(r_1, r_2)$. The conflicts are solved using a single modification operation for every element (A, m) from $C(r_1, r_2)$, with the modification term $\tau = (A, tup_2(m)[A])$ and the pattern $\rho = \{(ID, id(m))\}$. The inserts are performed by executing an insert operation on r_1 for every tuple unmatched tuple from r_2, i.e., for every tuple in $U(r_2, r_1)$. Overall, this requires $|U(r_1, r_2)|$ delete operations, $|C(r_1, r_2)|$ modification operations, and $|U(r_2, r_1)|$ insert operations. Any sequence of these operations is a transformer for r_1 and r_2.$\blacklozenge$

3.2 The Update Distance

The described transformers not necessarily reflect the optimal solution regarding the number of update operations needed to transform one database into another. Often, there is the possibility to solve more than one conflict using a single modification operation. The same is true for multiple deletes in order to minimize the overall number of necessary operations. This is reflected in the definition of the following distance measure, which considers update operations that affect an arbitrary number of tuples.

DEFINITION 8 (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.$\blacklozenge$

Note that the update distance also is not a metric as it is not a symmetric relation, i.e., $\Delta_U(r_1, r_2)$ not necessarily equals $\Delta_U(r_2, r_1)$. We consider the minimal transformers as explanations for observed differences between two databases. In order to avoid meaningless (or trivial) update sequences like (1) *delete all tuples in r_1*, and then (2) *for each tuple in r_2 perform* *an insert operation*, we further restrict the valid update operations within the transformers. For any intermediate state r_i in the process of transforming r_1 into r_2 an operation ψ is *valid*, if ψ is active and:

a) ψ is an insert operation, with $t_{new} \in r_2 / (r_2 \ltimes_{ID} r_1)$, ,

b) ψ is a delete operation, where $\rho_\delta(r_i) \subseteq r_1 / (r_1 \ltimes_{ID} r_2)$, or

c) ψ is a modification operation.

We thereby allow inserts only for tuples from r_2 without a matching partner in r_1 and deletes for tuples in r_1 without a matching partner in r_2. Modification operations are unrestricted. For the databases in Figure 5 the listed transformers are minimal. This results in an update distance $\Delta_U(r_1, r_2)$ of two and an update distance $\Delta_U(r_2, r_1)$ of three. The resolution distance and the update distance both describe minimal sequences of update operations for transforming one database into the other. They differ, however in the set of utilized update operations. In Section 4.3.2 we describe briefly how to extend the approaches for unrestricted update operations.

Unfortunately, there does not exist an easy formula for calculating of the update distance as there is one for the resolution distance. An algorithm to determine the update distance and the set of all minimal transformers for a given pair of databases is described in Section 4. While the calculation of the update distance is non-trivial, we can define upper and lower bounds.

LEMMA 2: An upper bound for the update distance between a pair of databases r_1 and r_2, denoted by $UB(r_1, r_2)$, is given by the resolution distance $\Delta_R(r_1, r_2)$.

PROOF: Due to LEMMA 1, there exists a transformer of length $\Delta_R(r_1, r_2)$ for r_1 and r_2. Any transformer of length greater than the resolution distance is therefore not minimal.♦

To also define a lower bound, we make use of the fact that according to our definition each modification operation modifies only one attribute. We subsume the conflicts that are potentially solvable using a single modification operation within a conflict group.

DEFINITION 9 (SOLUTION): Given a pair of databases r_1 and r_2 and a matching pair $m \in M(r_1, r_2)$. The *solution* of an existing conflict $(m, A) \in C(r_1, r_2)$ is given by the value $tup_2(m)[A]$ that has to by used as modification value in a modification operation to solve the conflict when transforming r_1 into r_2.♦

DEFINITION 10 (CONFLICT GROUP): Given a pair of databases r_1 and r_2. A conflict group κ is an attribute-value pair (A, x) with $attr(\kappa) = A \in R$ and $value(\kappa) = x \in dom(A)$. κ represents the subset of conflicts (m, A) from $C(r_1, r_2)$ 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(r1, r2) be the set of all conflict groups between a pair of databases. There exist two conflict groups for the databases r_1 and r_2 of Figure 5, namely $\kappa_1 = (A_2, 2)$ and $\kappa_2 = (A_4, 3)$. Note, that the set $K(r_2, r_1)$ for the databases of Figure 5 contains three conflict groups, i.e., $\kappa_1 = (A_2, 1)$, $\kappa_1 = (A_4, 0)$, and $\kappa_1 = (A_4, 1)$.

Due to the definition of the modification operations, we need at least one modification operation for solving the conflicts represented by a given conflict group.

LEMMA 3: The lower bound for the update distance between a pair of databases r_1 and r_2, denoted by $LB(r_1, r_2)$, is given by:

$$LB(r_1, r_2) = |U(r_2, r_1)| + |K(r_1, r_2)| + \begin{cases} 1, \text{ if } (U(r_1, r_2)) \neq \emptyset \\ 0, \text{ else} \end{cases}.$$

PROOF: In order to transform r_1 into r_2 with the restrictions for update operations as described above, we need exactly one insert operation for each tuple in $|U(r_2, r_1)|$, at least one modification operations for each conflict group in $K(r_1, r_2)$, and at least one delete operations if there are tuples to be deleted.♦

For the example in Figure 4 the update distance is three, as shown by the update sequences in c). The lower bound of the update distance is one and the upper bound is five. For the databases in Figure 5 the update distance $\Delta_U(r_1, r_2)$ is two, which is also the lower bound.

4 TRANSIT - Minimal Transformers for Databases

This section describes the TRANSIT algorithms to determine the set of minimal transformers for similar databases r_1 and r_2. Regarding the minimization of transformer length, an early execution of insert operations does not provide any benefit. Instead, early inserts bear the chance that following modification or delete operations affect the inserted tuples and cause additional contradictions. Inserts are therefore delayed until all other contradictions have been eliminated. Delete operations can be handled as special cases of conflict resolution with modification operations. We therefore omit the separated treatment of deletes and postpone this to Section 4.3. As a consequence, we only consider modification operations and restrict the presented algorithm to databases having a complete mutual overlap, i.e., $U(r_1, r_2) = U(r_2, r_1) = \emptyset$.

Given a pair of databases r_o and r_t, called *origin* and *target*, the TRANSIT algorithms essentially enumerate the space of all databases reachable by applying sequences of modification operations to r_o. 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 outermost importance. We describe a hashing scheme for efficient duplicate checking. We show how we use upper and lower bounds defined in Section 3.2 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. Finally, Section 4.3 explains how to handle delete operations as a special case of modification operations and briefly discusses the usage of unrestricted sequences of update operation. Still, throughout this paper we mainly limit our explanations to modification operations.

4.1 Search Space Exploration

Given a pair of databases r_o and r_t 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. We call the resulting databases 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 number of modification operations necessary to derive r_t from r_o, i.e., the update distance $\Delta_U(r_o, r_t)$. 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. 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 databases, we ensure that our first match with r_t defines the shortest possible sequence.

4.1.1 Transition Graph

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

DEFINITION 11 (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 $\psi(r_1) = r_2$, $r_1, r_2 \in V$. The update operation represents the edge label, denoted by *label*(e). We call r_1 the source of e, denoted by *source*(e), and r_2 the target of e, denoted by *target*(e).♦

A path between two databases r_1 and r_2 within the transition graph is a sequences of edges that connect $r_1$with r_2.

DEFINITION 12 (PATH): A *path* $\varphi = <e_1, \ldots, e_p>$ 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 for r_1 and r_2. The path $\varphi = <e_1, \ldots, e_p>$ represents a transformer Ψ = <*label*(e_1), …, *label*(e_p)>, with Ψ(*source*(e_1)) = *target*(e_p). In accordance to DEFINITION 6 a path is minimal if no shorter path between the same two databases exists. Clearly, the set of minimal transformers for r_o and r_t is given by all minimal paths from r_o to r_t within the transition graph. For a transition graph $G_T = (V, E)$ the *minimal transition graph* $G_{T_{min}} = (V_{min}, E_{min})$ with $V_{min} \subseteq V$ and $E_{min} \subseteq E$ is the part of the transition graph G_T containing only those vertices and edges that are contained in the minimal paths between r_o and r_t.

Essentially, the TRANSIT-algorithms iteratively construct the transition graph – or a part of it containing at least the minimal transition graph – starting with r_o as the only vertex. Figure 6 shows an example of such a transition graph. The different levels are outlined by horizontal lines and derivable databases are only shown at the level of their update distance from r_o (as opposed to showing them on each level at which they are derived). Vertices and edges of the minimal transition graph are enclosed within a gray box.

Duplicate databases while constructing the transition graph occur whenever the same database is derived by different update sequences. We differ between *inter-level* and *intra-level* duplicates. Inter-level duplicates occur, if update sequences of different length derive the same database, i.e., the same databases is derived at different levels. Duplicates at different levels of the graph may introduce cycles. Since the corresponding edges – delineated by dotted lines for clarity in Figure 6 – cannot be part of a minimal transformer, they are not included in the graph. This ensures that the resulting graph is acyclic. Intra-level duplicates result from different update sequences of equal length that derive the same database. These duplicate databases result in multiple edges between two vertices on adjacent distance levels.

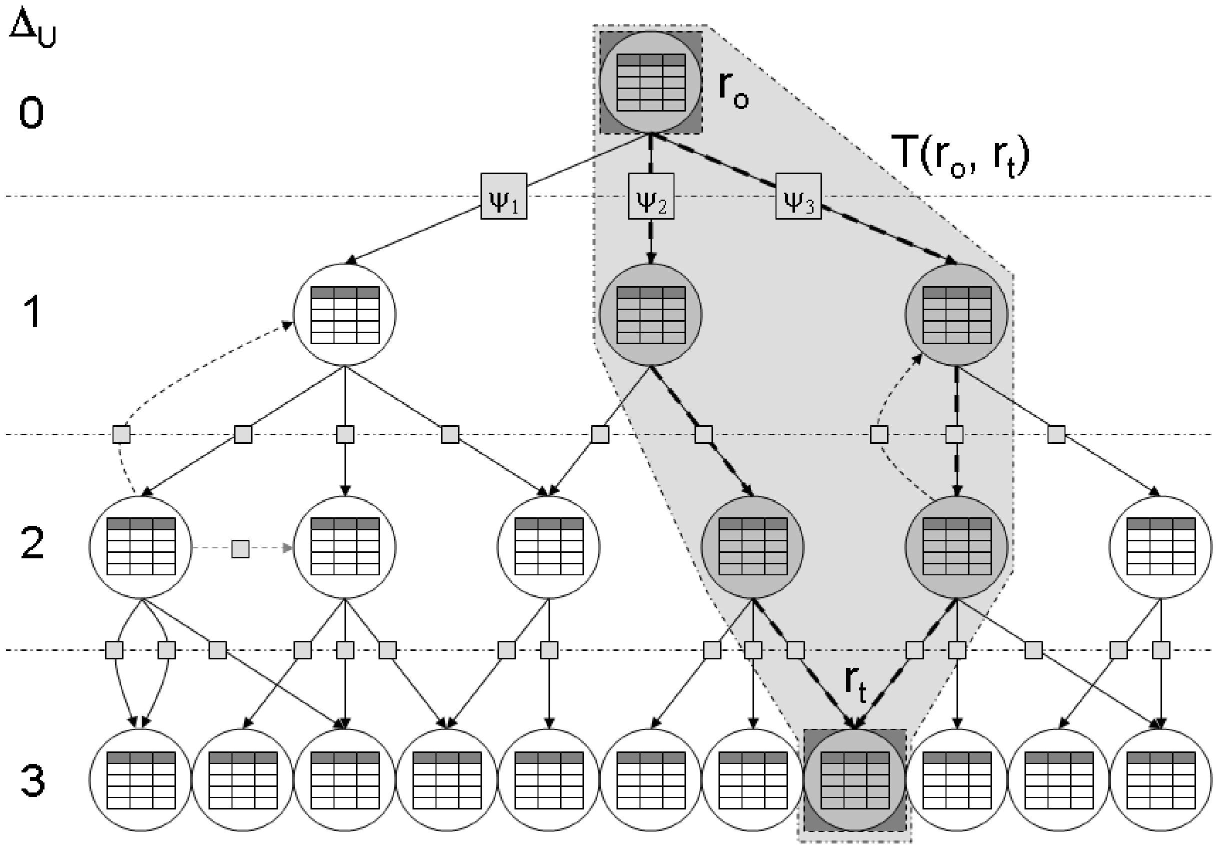


Figure 6: An exemplified transition graph as generated by the TRANSIT algorithm without pruning.

4.1.2 Duplicate Detection

A large portion of all generated databases are duplicates due to different update sequences deriving the same database. 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 4. Also, may update sequences derive the same database from itself. For example, the update sequence $<((A_2, 0), \{(A_1, 1)\}), ((A_2, 1), \{(A_1, 1)\})>$ 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.

Figure 7 shows a pair of databases having an update distance $\Delta_U(r_1, r_2)$ of four. Also shown in the figure are the number of newly generated databases at each level, as well as the number of duplicates, and the overall number of executed modification operations. There are a total of 24,586,604 executed modification operations while processing, i.e., generating the derivable databases, for 198,019 databases. The generated transition graph has a total of 4,823,538 vertices and 16,997,183 edges. The Figure also shows, that there is a very high rate of generated duplicates (ca. 80% of the generated databases), thus necessitating and justifying additional effort in order to detect and remove these duplicates. This number would even be higher if duplicate databases where not detected and removed from the graph.

Duplicate detection requires comparison of entire databases, i.e., the complete scan of two 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.

r_1	A_1	A_2	A_3	A_4
	1	1	2	3
	2	1	3	3
	3	1	2	1
	4	1	2	2
	5	1	2	7
	6	1	2	6
	7	2	2	5
	8	0	2	6

$\rightarrow$

r_2	A_1	A_2	A_3	A_4
	1	2	2	6
	2	2	3	6
	3	2	3	1
	4	2	3	2
	5	2	3	7
	6	2	3	6
	7	2	2	5
	8	0	2	6

	Generated Databases	Inter-level Duplicates	Intra-level Duplicates	Operations executed
$\Delta_U = 1$	111	0	11	122
$\Delta_U = 2$	5,761	1,905	5,585	13,251
$\Delta_U = 3$	192,146	165,303	340,871	698,320
$\Delta_U = 4$	4,625,519	7,422,214	11,827,178	23,874,911
$\sum$	4,823,537	7,589,422	12,173,645	24,586,604

Figure 7: The number of generated databases, duplicates, and modification operations executed using the naïve approach for a pair of similar databases of update distance four.

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], …, (n * m) - 1:t{m}[A_n], called the cell index (Figure 8). With each database we maintain a list of the conflicting values with an order based on this cell index (Figure 8 shows such a list for the conflicts between databases r_1 and r_2 from Figure 7). Starting at position 0, we select k values from this list, having cell index positions c_1, …, c_k, with c_i = (i – 1) * (*number of conflicts* / k) for 1 ≤ i ≤ 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. For the example in Figure 8 for k = 4 we use the values at position 0, 3, 6, and 9 with the resulting hash value being 2131 (the position of the digits being numbered from right to left).

We also tested a hash function based on a histogram of the attribute values occurring within a database. We thereby maintain for each occurring value from *dom*(A) the number of its occurrences within the databases (also shown in Figure 8). From this list we again take k values to compute the hash value. We found the later scheme to be inferior in our experiments.

r_1	A_1	A_2	A_3	A_4
	1	1	2	3
	2	1	3	3
	3	1	2	1
	4	1	2	2
	5	1	2	7
	6	1	2	6
	7	2	2	5
	8	0	2	6

Cell Index	A_1	A_2	A_3	A_4
	1	*2*	*3*	*4*
	5	*6*	*7*	*8*
	9	*10*	*11*	*12*
	13	*14*	*15*	*16*
	17	*18*	*19*	*20*
	21	*22*	*23*	*24*
	25	*26*	*27*	*28*
	29	*30*	*31*	*32*

List of Conflicting Values

Position	0	1	2	3	4	5	6	7	8	9	10	11
Index	*2*	*4*	*6*	*8*	*10*	*11*	*14*	*15*	*18*	*19*	*22*	*23*
Value	1	3	1	3	1	2	1	2	1	2	1	2

Histogram of Values

	0	1	2	3	4	5	6
Value	0	1	2	3	5	6	7
Occur.	1	7	9	3	1	2	1

Figure 8: Cell index an list of conflicting values as used for the hashing function.

4.1.3 Representation of Databases

The usage of a hash table greatly reduces the number of comparisons of complete databases. Still, in cases of equal hash values we must access the actual values in the database for comparison, i.e., we need access to the complete database in the vertices. We implemented two different representations of these databases.

In a first representation we maintain complete databases within the vertices. While this is memory consuming the access to the actual data values is fast. Alternatively, for each database r only a transformer $\Psi(r_o) = r$ is maintained within the corresponding vertex. This greatly reduces the memory requirement for transition graph maintenance. As a downside, we are now forced to re-derive the database r from the origin every time access to the actual tuples and attribute values is required. Therefore, this representation even further depends on the ability of the hash function to generate equally distributed hash values for the databases in order to avoid collisions.

4.1.4 Pruning

The TRANSIT-algorithms try to avoid generating the complete transition graph. The number of vertices outside of the minimal transition graph in Figure 6 shows that many of the generated databases are not part of any minimal transformer. This observation is supported by examining the minimal transition graph for the databases of Figure 7. The minimal transition graph contains 18 vertices and 36 edges which is far below the number of about 5,000,000 generated databases and 17,000,000 edges in the constructed transition graph.

This large difference suggest that pruning is essential. In TRANSIT, pruning uses the upper and lower bounds for the update distance as defined in Section 3.2. Let β denote the current upper bound for the update distance between r_o and r_t. This bound is initialized following LEMMA 2 as $UB(r_o, r_t)$. 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 length $|\Psi| = \Delta_U(r_o, r)$. Then LEMMA 1 guarantees the existence 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 4: 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: Transformer Ψ_3 is a concatenation of $\Psi_1 = <\psi_{11}, \ldots, \psi_{1k}>$ and $\Psi_2 = <\psi_{21}, \ldots, \psi_{2p}>$, i.e., $\Psi_3 = <\psi_{11}, \ldots, \psi_{2k}, \psi_{21}, \ldots, \psi_{2p}>$. The length of Ψ_3 is $|\Psi_1| + |\Psi_2|$ and the result of $\Psi_3(r_1)$ equals $\Psi_2(\Psi_1(r_1))$ which is r_3.♦

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

4.1.5 Breadth-First Algorithm

The described approach resembles a branch and bound behavior [LD60]. Therein, we can explore the search space either in breadth-first or in depth-first manner. We first describe a breadth-first algorithm.

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. Figure 9 shows the changes to the transition graph from Figure 6 when using a breadth-first approach. The fictitious upper and lower bounds of the databases are shown in white boxes on the right of every vertex. The order in which the databases are processed is given by the number in the dark gray circles attached to the left of the vertices. For example, the database on the left side of level 1 is pruned, because every path from r_o to r_t through this vertex is at least of length 5, while the current bound is 4 after generating all databases at level 1. Due to the pruning of databases in the bound step large portions of the originally shown transition graph not generated or tested in the breadth-first approach.

The corresponding algorithm TRANSIT-BFS is shown in Figure 10. Each database from the previous level, maintained in V_P, is processed while enumerating the current level (*lines* 9-27). The databases at the current level, maintained in V_C, afterwards become the candidates for the enumeration of the next level (*line* 28). We sort the candidates in ascending order of their lower and upper bounds. This is done with the intention of being able to decrease the current bound β as soon as possible, 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* 30). In Figure 9 the databases in the sets V_p and V_C are highlighted for the construction of level 2.

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

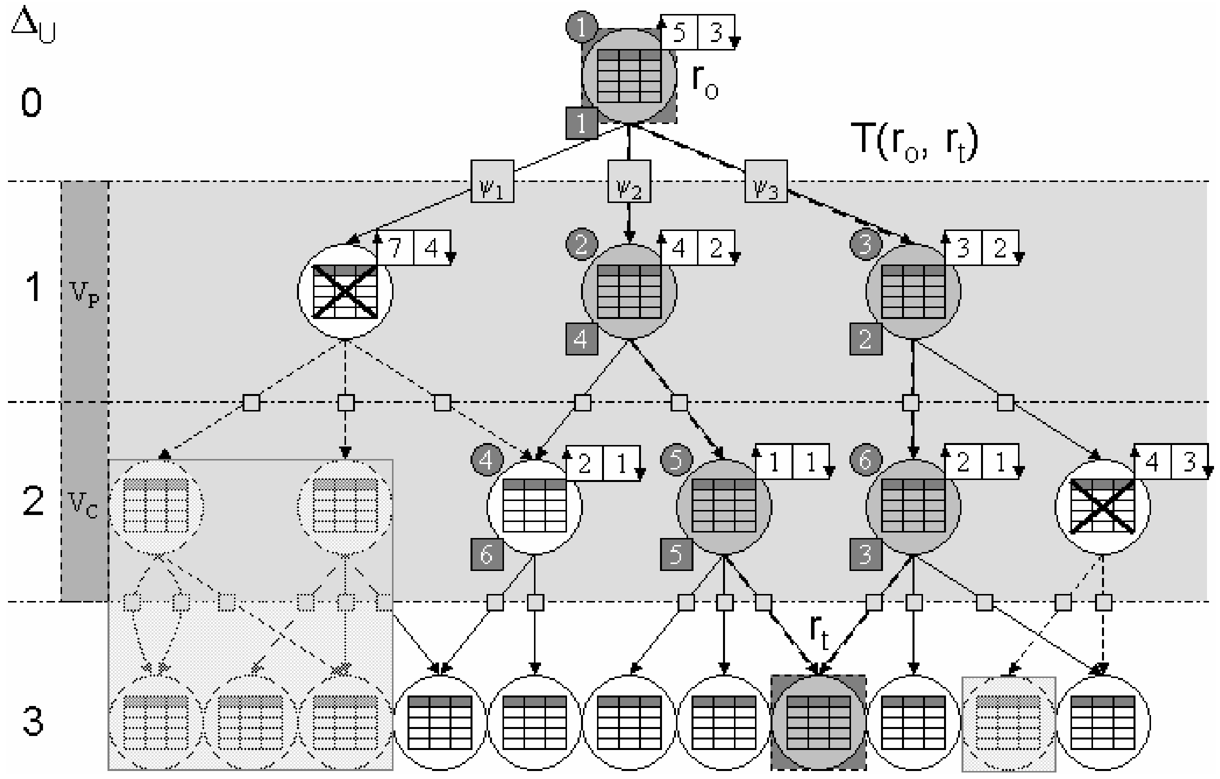


Figure 9: Exemplified transition graph construction when using a breadth-first approach

```
1   TRANSIT-BFS($r_o$, $r_t$) {
2     $G_T$ := ({$r_o$}, {});
3     $V_P$ := $V(G_T)$;
4     $\Delta_U$ = 0;
5     $\beta$ := $\Delta_R(r_o, r_t)$;
6     while($r_t \notin V_P$) {
7       $\Delta_U$ = $\Delta_U$ + 1;
8       $V_C$ := {};
9       for each $r_i \in V_P$ do {
10        MDF := modifier($r_i$, $r_t$);
11        for each $\psi \in$ MDF do {
12          $r_{new}$ := $\psi$(clone($r_i$));
13          if ((LB($r_{new}$, $r_t$) + $\Delta_U$) $\leq$ $\beta$) {
14            if ($r_{new} \notin V(G_T)$) {
15              $V(G_T)$ := $V(G_T) \cup \{r_{new}\}$;
16              $E(G_T)$ := $E(G_T) \cup \{(r_i, r_{new}, \psi)\}$;
17              $V_C$ := $V_C \cup \{r_{new}\}$;
18              if (($\Delta_R(r_{new}, r_t)$ + $\Delta_U$) < $\beta$) {
19                $\beta$ := $\Delta_R(r_{new}, r_t)$ + $\Delta_U$;
20                prune $V_p$, $V_C$, $G_T$, $\beta$;
21              }
22            } else if ($r_{new} \in V_C$) {
23              $E(G_T)$ := $E(G_T) \cup \{(r_i, r_{new}, \psi)\}$;
24            }
25          }
26        }
27      }
28      $V_P$ := sort($V_C$);
29    }
30    output min_paths($G_T$, $r_o$, $r_t$);
31  }
```

Figure 10: The breadth-first algorithm TRANSIT-BFS.

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* 9). 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.

4.1.6 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, after finishing the processing of the current database, i.e., generating all databases derivable with a single modification operation, we immediately proceed to the next distance level. From all generated database, we chose the one with the smallest lower bound as 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. This is continued until all databases that have not been pruned by the bounding step have been tested.

In TRANSIT-DFS we maintain the databases processed and generated on the current path on a stack to enable upward traversal. Compared to TRANSIT-BFS, duplicate detection is complicated by the problem that the identical databases may be generated multiple times at decreasing levels. Every time a database is repeatedly derived at a lower level, it has to be considered as a candidate again and cannot be rejected as a duplicate. Due to the depth-first proceeding we temporarily add databases to the transition graph having an update distance from r_o above the update distance $\Delta_U(r_o, r_t)$. This is contrary to our definition of the transition graph (DEFINITION 11), where we only consider databases having an update distance below or equal to $\Delta_U(r_o, r_t)$. Still, due to the performed pruning, we will remove databases r with an update distance $\Delta_U(r_o, r) > \Delta_U(r_o, r_t)$ from the final transition graph.

The advantage of using the branch and bound approaches is shown by comparing the following numbers with those from Figure 7. Using TRANSIT-BFS, the minimal transformers for the two databases are found after exploring only 255 databases. The algorithm thereby executes a total of 42,010 modification operations and generates 3,651 databases. When using TRANSIT-DFS 1,433 databases are processed, with 226,655 modification operations executed and 1,609 databases generated.

4.2 Enumeration of Valid Modification Operations

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

4.2.1 The Set of Valid Modification Terms

Terms (DEFINITION 2) are attribute-value pairs. Within modification terms only non-key attributes are permitted. 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 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 constrain the set of possible modification values. We accomplish this goal by using the values occurring within the current database r and the target r_t. In summary, we permit the following values for 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 using a single modification operation afterwards. An example is shown in Figure 11. Without allowing existing values to be used as modification values, we need at

least eight operation for solving the existing conflicts (four operations selecting on A_5 = 1 and four operations selecting on A_5 = 0). If we set the values in attribute A_5 for tuples t{7} and t{8} to 1, we are enabled to solve the conflicts in attributes A_2 to A_5 using 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 currently used values. We chose one value using a random function. We call these values *proxies*.[1]

The proxies are maintain within a separate list for each attribute, called *proxy*(A). Within the final modification sequence, the occurring proxies can be replaced by any valid subset of *dom*(A) of size |*proxy*(A)|, which is disjoint with $r_o[A] \cup r_t[A]$.

r_1	A_1	A_2	A_3	A_4	A_5
	1	3	7	7	3
	2	3	8	8	3
	3	3	9	9	3
	4	1	1	1	1
	5	2	2	2	1
	6	4	4	4	1
	7	5	5	5	0
	8	6	6	6	0
	9	7	3	10	3
	10	8	3	11	3
	11	9	3	12	3
	12	10	10	3	7
	13	11	11	3	8
	14	12	12	3	9

$\rightarrow$

r_2	A_1	A_2	A_3	A_4	A_5
	1	3	7	7	7
	2	3	8	8	8
	3	3	9	9	9
	4	3	3	3	3
	5	3	3	3	3
	6	3	3	3	3
	7	3	3	3	3
	8	3	3	3	3
	9	7	3	10	10
	10	8	3	11	11
	11	9	3	12	12
	12	10	10	3	13
	13	11	11	3	14
	14	12	12	3	15

Figure 11: An example for enhancing the selectivity of existing patterns in order to find an optimal solution.

4.2.2 The Set of Valid Selection 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 existing frequent itemset mining algorithms [AS94, HPY00]. Obviously this set very likely contains pairs of patterns ρ_1, ρ_2 with $\rho_1 \neq \rho_2$ and $\rho_1(r) = \rho_2(r)$. Using the patterns in P(r) when enumerating modification operations would result in operations with equal effect. Avoiding this redundancy is accomplished by restricting P(r) to the set of *closed patterns*, denoted by $P_C(r)$. The following definition is taken from [Bay98][PBTL99].

DEFINITION 13 (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.

[1] The shown numbers for the example in Figure 6 reflect a situation, where proxies are not allowed. If proxies are allowed, the number of operations executed (and database generated) for the first three levels are 164 (153), 25,051 (11,755), and 1,989,604 (624,659).

LEMMA 5: 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: We view the tuples as sets of terms. Let ρ_{common} denote the set of terms common to the tuples in $\rho(r)$. This set forms a closed pattern for $\rho(r)$. All tuples in $\rho(r)$ satisfy ρ_{common} and if we add a term to ρ_{common} the pattern will no longer be satisfied by all tuples in $\rho(r)$. Therefore, ρ_{common} equals $\rho' \in P_C(r)$ with $\rho(r) = \rho'(r)$.♦

Based on LEMMA 5 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. To determine the set of closed patterns, we start with a single scan of the database. Each tuple is a closed pattern due to the primary key constraint. While scanning the database we determine the set of terms for each attribute and maintain a list of tuples, in which these terms occur. We then prune all terms having a support, i.e., a tuple list size, of 1, since they only occur in the closed patterns already represented by single tuples. We than use any of the existing methods for mining closed itemsets like CHARM [ZH02], CLOSET+ [WHP03], or FARMER [CTX+04].Within our implementation we currently use CHARM [ZH02].

4.2.3 Filtering Valid Modification 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. The set of valid modification operations for database r_1 of Figure 5 is shown in Figure 12. There are 8 closed patterns (including the empty pattern). If we assume *proxy*(A_2) = {9}, *proxy*(A_3) = {9}, and *proxy*(A_4) = {9}, 10 modification terms. As a result we receive a total of 65 valid modification operations.

r_1	A_1	A_2	A_3	A_4
	1	2	1	1
	2	1	2	1
	3	1	2	0
	4	1	2	1

r_2	A_1	A_2	A_3	A_4
	1	2	1	3
	2	2	2	3
	3	2	2	3
	4	2	2	3

Patterns

ρ_1 = {	(A_1, 1)	(A_2, 2)	(A_3, 1)	(A_4, 1)	}
ρ_2 = {	(A_1, 2)	(A_2, 1)	(A_3, 2)	(A_4, 1)	}
ρ_3 = {	(A_1, 3)	(A_2, 1)	(A_3, 2)	(A_4, 0)	}
ρ_4 = {	(A_1, 4)	(A_2, 1)	(A_3, 2)	(A_4, 1)	}
ρ_5 = {		(A_2, 1)	(A_3, 2)		}
ρ_6 = {				(A_4, 1)	}
ρ_7 = {		(A_2, 1)	(A_3, 2)	(A_4, 1)	}
ρ_8 = {					}

Terms

A_2:	(A_2, 1)	(A_2, 2)	(A_2, 9)	
A_3:	(A_3, 1)	(A_3, 2)	(A_3, 9)	
A_4:	(A_4, 0)	(A_4, 1)	(A_4, 3)	(A_4, 9)

Operations

	ψ_{18} = ((A_2, 9), ρ_7)	ψ_{37} = ((A_3, 9), ρ_7)
	ψ_{19} = ((A_2, 9), ρ_8)	ψ_{38} = ((A_3, 9), ρ_8)
ψ_1 = ((A_2, 1), ρ_1)	ψ_{20} = ((A_3, 1), ρ_2)	ψ_{39} = ((A_4, 0), ρ_1)
ψ_2 = ((A_2, 1), ρ_6)	ψ_{21} = ((A_3, 1), ρ_3)	ψ_{40} = ((A_4, 0), ρ_2)
ψ_3 = ((A_2, 1), ρ_7)	ψ_{22} = ((A_3, 1), ρ_4)	ψ_{41} = ((A_4, 0), ρ_4)
ψ_4 = ((A_2, 1), ρ_8)	ψ_{23} = ((A_3, 1), ρ_5)	ψ_{42} = ((A_4, 0), ρ_5)
ψ_5 = ((A_2, 2), ρ_2)	ψ_{24} = ((A_3, 1), ρ_6)	ψ_{43} = ((A_4, 0), ρ_6)
ψ_6 = ((A_2, 2), ρ_3)	ψ_{25} = ((A_3, 1), ρ_7)	ψ_{44} = ((A_4, 0), ρ_7)
ψ_7 = ((A_2, 2), ρ_4)	ψ_{26} = ((A_3, 1), ρ_8)	ψ_{45} = ((A_4, 0), ρ_8)
ψ_8 = ((A_2, 2), ρ_5)	ψ_{27} = ((A_3, 2), ρ_1)	ψ_{46} = ((A_4, 1), ρ_3)
ψ_9 = ((A_2, 2), ρ_6)	ψ_{28} = ((A_3, 2), ρ_6)	ψ_{47} = ((A_4, 1), ρ_5)
ψ_{10} = ((A_2, 2), ρ_7)	ψ_{29} = ((A_3, 2), ρ_7)	ψ_{48} = ((A_4, 1), ρ_8)
ψ_{11} = ((A_2, 2), ρ_8)	ψ_{30} = ((A_3, 2), ρ_8)	ψ_{49} = ((A_4, 3), ρ_1)
ψ_{12} = ((A_2, 9), ρ_1)	ψ_{31} = ((A_3, 9), ρ_1)	ψ_{50} = ((A_4, 3), ρ_2)
ψ_{13} = ((A_2, 9), ρ_2)	ψ_{32} = ((A_3, 9), ρ_2)	ψ_{51} = ((A_4, 3), ρ_3)
ψ_{14} = ((A_2, 9), ρ_3)	ψ_{33} = ((A_3, 9), ρ_3)	ψ_{52} = ((A_4, 3), ρ_4)
ψ_{15} = ((A_2, 9), ρ_4)	ψ_{34} = ((A_3, 9), ρ_4)	ψ_{53} = ((A_4, 3), ρ_5)
ψ_{16} = ((A_2, 9), ρ_5)	ψ_{35} = ((A_3, 9), ρ_5)	ψ_{54} = ((A_4, 3), ρ_6)
ψ_{17} = ((A_2, 9), ρ_6)	ψ_{36} = ((A_3, 9), ρ_6)	…

Figure 12: The set of valid modification operations for database r_1.

4.3 Handling Delete and Insert Operations

To conclude the description of the TRANSIT-algorithms we now describe the handling of insert and delete operations. We start with the case of restricted operations as defined in Section 3.2 and then describe the extension for sequences of arbitrary update operations.

4.3.1 Delete Operations as Special Case of Modifications

In case of restricted update operations the execution of necessary insert operations is performed after solving existing conflicts. Delete operations are handled as special cases of modification operations. We therefore need to slightly alter given database. The set of tuples from r_o to be deleted is given by $U(r_o, r_t)$. We add a special attribute A_D to schema R and set $t[A_D] = 0$ for each $t \in U(r_o, r_t)$. We also insert the tuples from $U(r_o, r_t)$ to r_t changing the value of $t[A_D]$ to 1. For all other tuples in databases r_o and r_t attribute A_D is undefined.

The attribute A_D acts as a delete flag and the values are altered using modification operations. Terms for attribute A_D are excluded when enumerating valid selection patterns. However, A_D is allowed as attribute in modification terms. The only valid modification value in these terms is 1. A modification operation $\psi_\mu = (\tau, \rho)$ with $\tau = (A_D, 1)$ represents a delete operation $\psi_\delta = (\rho)$. A tuple t with $t[A_D] = 1$ then represents a deleted tuple. Following the definition of feasible update operations, in Section 3.2 we additionally have to restrict ρ to select only tuples from $U(r_o, r_t)$. In the resulting modification sequences the modification operations with $\tau = (A_D, 1)$ are replaced by the appropriate delete operations.

4.3.2 Handling Arbitrary Sequences of Update Operations

The described algorithm for enumerating valid modification operations is easily extended to allow arbitrary insert and delete operations within update sequences. When enumerating modification operations for a database r we (i) add an insert operation for every tuple in r_t / r, and (ii) add a delete operation for every valid modification pattern.

The TRANSIT-algorithms operate independently of the set of allowed update operations. Allowing insert and delete operations enables the application of TRANSIT on database pairs that do not completely overlap. However, including insert and delete implies changes to the definitions for the upper and lower bounds from Section 3.2. For every pair of databases r_1 and r_2 there always exists a trivial transformer $\Psi(r_1) = r_2$ that (i) deletes all tuples from r_1 and (ii) successively insert all tuples from r_2. Therefore, the upper bound is now given by $min(\Delta_R(r_1, r_2), |r_2| + 1)$. The calculation of the lower bound must consider the possibility of missing tuples in r_1. Furthermore, the number of tuples in r_2 may by lower than the number of conflict groups in $K(r_1, r_2)$. The lower bound is now defined by $min(K(r_1, r_2) + |r_2/r_1|, |r_2| + 1)$.

5 Heuristics and Problem Variations

The described TRANSIT-algorithms are only applicable for small databases. For larger databases the search space of derivable databases is enormous. Despite pruning over 95% of the generated databases immediately (see Section 6) processing the remaining databases is still to expensive. Within this section we describe heuristics which do not necessarily find the best (exact) solution, but instead are able to handle databases of almost arbitrary size. We analyze the quality of the computed results in Section 6.

5.1 A Classification of Modification Operations

A first approach is to reduce the number of valid modification operations. As Figure 12 suggests, the number of modification operations can become very large for databases even of mediocre size. We therefore restrict the valid modification operations based on the effect they have when applied on the given database. Again, we only consider databases r_1 and r_2 which with complete overlap.

Given a pair of databases r_1 and r_2 and a modification operation $\psi_\mu(\tau, \rho)$. When ψ_μ is applied on r_1 we divide the set of affected tuples, i.e., $\rho(r_1)$, into four disjunctive subsets, based on the effect the modification operation has on them:

- **NEUTRAL** (w_u): The set of selected tuples, which remain unchanged by the modification operation, as they already poses the new value in the modified attribute, i.e.,

  $$w_u = \{t \mid t \in \rho(r_1) \wedge t[\mathit{attr}(\tau)] = \mathit{value}(\tau)\}.$$

- **NEW** (w_n): The set of selected tuples currently not in conflict with their matching partner in the modified attribute *attr*(τ), that will contain a conflict in this attribute after execution of the modification operation, i.e.,

  $$w_n = \{t_1 \mid t_1 \in \rho(r_1) \wedge \exists\, t_2\ (t_2 \in r_2 \wedge t_1[ID] = t_2[ID] \wedge t_1[\mathit{attr}(\tau)] = t_2[\mathit{attr}(\tau)] \wedge t_1[\mathit{attr}(\tau)] \neq \mathit{value}(\tau))\}$$

- **CHANGED** (w_c): The set of selected tuples with an existing conflict between them and their matching partner in the modified attribute that are altered by the modification operation, i.e.,

  $$w_c = \{t_1 \mid t_1 \in \rho(r_1) \wedge \exists\, t_2\ (t_2 \in r_2 \wedge t_1[ID] = t_2[ID] \wedge t_1[\mathit{attr}(\tau)] \neq t_2[\mathit{attr}(\tau)] \neq \mathit{value}(\tau))\}$$

- **SOLVED** (w_s): The set of selected tuples containing a conflict in the modified attribute that will be solved after execution of the modification operation, i.e.,

  $$w_s = \{t_1 \mid t_1 \in \rho(r_1) \wedge \exists\, t_2\ (t_2 \in r_2 \wedge t_1[ID] = t_2[ID] \wedge t_1[\mathit{attr}(\tau)] \neq t_2[\mathit{attr}(\tau)] \wedge t_2[\mathit{attr}(\tau)] = \mathit{value}(\tau))\}.$$

Based on this classification we define four different classes of modification operations:

CLASS 0: Set of all valid modification operations.

CLASS 1: Set of modification operations that reduce the overall number of conflicts, i.e., $|w_s| > |w_n|$. We call these modification operations *conflict reducer*.

CLASS 2: Set of modification operations that reduce the overall number of conflicts and do not introduce any new conflicts, i.e., $w_s \neq \emptyset$ and $w_n = \emptyset$. We call these operations *conflict solver*.

CLASS 3: Set of modification operations that only solve conflicts or are neutral, i.e., $w_s \neq \emptyset$, $w_n = \emptyset$, and $w_c = \emptyset$. We call these operations *pure conflict solver*.

It follows that CLASS3 $\subseteq$ CLASS2 $\subseteq$ CLASS1 $\subseteq$ CLASS0. In order to reduce the number of possible modification operations we change the problem definition and only allow operations of a certain class within the process of determining the set of minimal transformers. A remaining problem is the determination of the class of a given modification operation. While distinguishing valid from invalid operations is fairly easy (CLASS 0), as described in Section 4.2.3, determining whether an operation is of CLASS i, i = 1, …, 3, requires more effort. We actually have to test each of the tuples affected, as well as their respective matching partner, i.e., we virtually have to execute the operation in the worst case.

5.2 Greedy TRANSIT

Another simple heuristic is applying a greedy algorithm. Given a pair of databases r_o and r_t, the greedy algorithm first determines the databases derivable from the origin by a single modification operation. A score is assigned to each of these databases. The database with the highest score is chosen as the starting point for the next level. For this database again all databases derivable by a single modification operation are generated and assigned with a score and so forth. This is continued until the target database is reached. Figure 13 shows the greedy algorithm GREEDY-TRANSIT. The algorithm returns a single transformer Ψ_T. r_s denotes the current starting point. For each database directly derivable from r_s the score, assigned by a function ω, is compared to the current maximum. If the score exceeds the current maximum the new database becomes the next starting point.

```
GREEDY-TRANSIT($r_o$, $r_t$) {
  $\Psi_T$ := <>;
  $r_s$ := $r_o$;
  while($r_s \neq r_t$) {
    $r_{next}$ := $r_s$;
    $\psi_{next}$;
    MDF := modifier($r_s$, $r_t$);
    for each $\psi \in$ MDF do {
      $r_{new}$ := $\psi$(clone($r_s$));
      if ($\omega(r_{new}) > \omega(r_{next})$) {
        $r_{next}$ := $r_{new}$;
        $\psi_{next}$ := $\psi$;
      }
    }
    $r_s$ := $r_{next}$;
    append($\Psi_T$, $\psi_{next}$);
  }
  return $\Psi_T$;
}
```

Figure 13: A greedy algorithm to calculate the update distance of a pair of databases.

The scoring function should assign the highest score to the database having the highest potential of reaching the target first. We tested two different scoring functions. The first assigns the highest score to the database with the smallest lower bound. For databases with equal lower bound the database with the smaller upper bound receives the higher score. We call the greedy TRANSIT-algorithm using this scoring function GREEDY-TRANSIT (LB). The second scoring function uses the upper and lower bounds in an opposite way, i.e., assigning the highest score to the database with the smallest upper bound, using the lower bound as a tie-breaker. We call the greedy TRANSIT-algorithm using this scoring function GREEDY-TRANSIT (UB). The scoring functions follow the assumption that either the database with the lowest number of conflicts or the lowest number of conflict groups has the potential of reaching the destination first. The example in Figure 4 shows that neither assumption is always correct, as the resulting transformer for each of the greedy approaches has a length of four.

Our scoring functions ensure 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 r_s, only the first operation, depending on the order in MDF, is returned within the final transformer Ψ_T.

5.3 Approximation of the Update Distance

Another heuristic is based on solving the conflicts within each conflict group independently. The sum of necessary operations for conflict solution of the individual conflict groups is an approximation of the update distance. The result is equal or above the lower bound, as we still need at least one modification operation per conflict group, and below or equal the upper bound, as we are still able to solve each conflict individually with a single modification operation. This approximation completely disregards the possible impact that the modification of values for some of the tuples may have on solving conflicts for other tuples.

Determining the minimal number of modification operations necessary to solve the conflicts within a conflict group individually still is expensive, as shown in Section 6.1. Therefore, we further restrict the set of valid modification operations for approximating the update distance in order to keep the computational cost in reasonable bounds. This restriction is done by considering only modification operations of CLASS 3. Therefore, for solving the conflicts represented by a conflict group κ, only operations having κ as modification term are valid. The modification patterns of these operations may only select tuples from r_o that are part of a conflict represented by κ or that already possess *value*(κ) for attribute *attr*(κ). The former is called *solution target set*, as these are the tuples that need to be modified for conflict solution, and the later is called *solution neutral set*, as these tuples are neutral regarding the described modification operations.

DEFINITION 14 (SOLUTION TARGET SET): Let $\kappa \in K(r_1, r_2)$ be a conflict group between a pair of databases r_1 and r_2. The *solution target set* of κ, denoted by $\xi(r_1, r_2, \kappa)$, is the set of tuples from r_1, that contain the conflicts represented by κ, i.e.,

$$\xi(r_1, r_2, \kappa) = \{t \mid t = tup_1(m) \wedge m \in M(r_1, r_2) \wedge tup_1(m)[attr(\kappa)] \neq tup_2(m)[attr(\kappa)] \wedge tup_2(m)[attr(\kappa)] = value(\kappa)\}.\blacklozenge$$

DEFINITION 15 (SOLUTION NEUTRAL SET): Let $\kappa \in K(r_1, r_2)$ be a conflict group between a pair of databases r_1 and r_2. The *solution neutral set* of κ, denoted by $\eta(r_1, r_2, \kappa)$, is the set of tuples from r_1 that are neutral regarding the solution of conflicts represented by κ, i.e.,

$$\eta(r_1, r_2, \kappa) = \{t \mid t \in r_1 \mid t[attr(\kappa)] = value(\kappa)\}.\blacklozenge$$

The cost for solving the conflicts represented by a conflict group κ is given by the minimal number of patterns that together select the group target set at least and the union of group target and neutral set at most. This cost forms the basis of our update distance approximation.

DEFINITION 16 (SOLUTION COST): Given a database r and two disjoint subsets s_t , $s_n \subseteq r$. The *solution cost*, denoted by $\theta(r, s_t, s_n)$, is the minimum number of patterns $\rho_1, \ldots, \rho_q$, that select s_t completely and $s_c \cup s_n$ at most, i.e., $s_c \subseteq \rho_1(r) \cup \ldots \cup \rho_q(r) \subseteq s_c \cup s_n$.♦

Definition 17 (Group Solution Cost): Given a pair of databases r_1 and r_2. The *group solution cost*, denoted by $\phi(r_1, r_2)$, is the sum of the solution cost for the conflict groups between the sources, i.e.,

$$\phi(r_1, r_2) = \sum_{\kappa \in K(r_1, r_2)} \theta(r_1, \xi(r_1, r_2, \kappa), \eta(r_1, r_2, \kappa)).\blacklozenge$$

The group solution cost $\phi(r_1, r_2)$ is used as an approximation of the update distance $\Delta_U(r_1, r_2)$ of databases r_1 and r_2. Note that there are cases, where this approximation is above the actual update distance or below. The first case occurs, whenever there are positive side effects of solving conflicts in one attribute for solving other conflicts. The later occurs, whenever the respective modification operations interfere with each other, i.e., after executing one of them, the other is no longer executable or has a different result. The group solution cost for the example in Figure 14 is 8.

r_1	A_1	A_2	A_3	A_4	A_5	A_6
t_1	1	1	2	3	1	1
t_2	2	1	3	3	1	0
t_3	3	1	2	1	0	0
t_4	4	1	2	2	1	0
t_5	5	1	2	7	1	1
t_6	6	1	2	6	1	1
t_7	7	2	2	5	1	1
t_8	8	0	2	6	1	1

$\rightarrow$

r_2	A_1	A_2	A_3	A_4	A_5	A_6
	1	2	2	6	1	1
	2	2	3	6	1	0
	3	2	3	1	0	0
	4	2	3	2	0	0
	5	2	3	7	0	1
	6	2	3	6	0	1
	7	2	2	5	1	1
	8	0	2	6	1	1

$\kappa_1 = (A_2, 2)$: $\theta(r_1, \xi(r_1, r_2, \kappa_1), \eta(r_1, r_2, \kappa_1)) = 1$

$\kappa_2 = (A_3, 3)$: $\theta(r_1, \xi(r_1, r_2, \kappa_2), \eta(r_1, r_2, \kappa_2)) = 3$

$\kappa_3 = (A_4, 6)$: $\theta(r_1, \xi(r_1, r_2, \kappa_3), \eta(r_1, r_2, \kappa_3)) = 3$

$\kappa_4 = (A_5, 0)$: $\theta(r_1, \xi(r_1, r_2, \kappa_4), \eta(r_1, r_2, \kappa_4)) = 1$

Figure 14: The group solution cost for a pair of databases.

The group solution cost may also be used as a replacement for the lower bound within the algorithms TRANSIT-BFS and TRANSIT-DFS. This may imply that the exact solution is missed. However, in all our experiments presented in Section 6.2.2 this heuristic computed the exact solution. The according algorithms are called TRANSIT-BFS (GS) and TRANSIT-DFS (GS), respectively. We can also use the group solution cost as a weight function in a greedy approach. We thereby enable the usage of proxies, which is omitted by the other weight functions. The corresponding algorithm is called GREEDY-TRANSIT (GS).

Computing the exact solution cost for a given pair of databases r_1 and r_2 and a given conflict group κ is expensive. We therefore implemented a greedy approach, shown in Figure 15. The calculation starts by determining P_{valid}, the subset of valid modification patterns that (i) only select tuples from the union $\xi(r_1, r_2, \kappa) \cup \eta(r_1, r_2, \kappa)$, and (ii) select at least one tuple from $\xi(r_1, r_2, \kappa)$. Let ρ_e denote the empty pattern and s_t, s_n denote $\xi(r_1, r_2, \kappa)$, $\eta(r_1, r_2, \kappa)$, respectively. We then choose repeatedly the pattern that selects the largest subset from s_t (*line* 10). This pattern is removed from P_{valid}. We also remove from s_t those tuples that satisfy this pattern. With each chosen pattern the solution cost is incremented by one. The algorithm terminates when s_t is empty. The algorithm for computing the group solution cost for a pair of databases using the described solution cost algorithm is called TRANSIT-APPROX. Basically, this algorithm calls GREEDY-SOLUTION-COST(r_1, $\xi(r_1, r_2, \kappa)$, $\eta(r_1, r_2, \kappa)$) for each conflict group $\kappa \in K(r_1, r_2)$ and summates the results.

```
GREEDY-SOLUTION-COST(r, $s_t$, $s_n$) {
  scost := 0;
  $P_{valid} = P_C(r) \cup \{\rho_e\}$;
  for each $\rho \in P_{valid}$ do {
    if (($\rho(r) \subset s_n$) || ($\rho(r) / (s_t \cup s_n) \neq \emptyset$)) {
      $P_{valid} := P_{valid} / \{\rho\}$;
    }
  }
  while ($s_t \neq \emptyset$) {
    $\rho_{max}$ := *max_select*($P_{valid}$, $s_t$);
    $s_t := s_t / \rho_{max}(s_t)$
    $P_{valid} := P_{valid} / \{\rho_{max}\}$;
    scost++;
  }
  return scost;
}
```

Figure 15: A greedy algorithm for calculating the solution cost.

6 Experimental Results

Within this section we discuss some of the results of our experiments using the described algorithms on different pairs of databases. We therefore mainly utilize the databases of Figure 1, named `F1`, the databases of Figure 4, named `F4`, the databases of Figure 7, named `F7`, and the databases of Figure 14, named `F14`. We also performed experiments on larger databases in conjunction with the heuristics described in the previous section. We start by describing properties of the algorithms for finding exact solutions and compare them afterwards with the implemented non-optimal algorithms. We implemented all algorithms using Java™ J2SE 5.0. The experiments where performed on a Citrix MetaFrame™ Server containing two Intel Xenon 2,4 GHz processors and 4 GB main memory.

6.1 Algorithms for Finding Exact Solutions

The necessary effort to determine the set of minimal transformers for the four pairs of databases is shown in Figure 16 a). We allow modification operations of CLASS 0 and proxies. In the first two columns the number of databases processed and modification operations executed for building the transition graph are shown. Also listed are the overall number of databases added to the graph and the number of databases generated as duplicates. The final results, i.e., the size of the minimal transition graphs, the total number of minimal transformers (|T|), and the update distances (Δ_U), are shown in Figure 16 c). Figure 16 b) shows the number of valid modification operations that where executed when processing the origin database of all four database pairs. Note that all experiments determined only the set T(r_1, r_2) of minimal transformers (T(GROUP1.AMPHIBIAN, GROUP2.AMPHIBIAN) in the case of `F1`). Also shown in Figure 16 b) are the number of generated databases and edges when processing the origin database of the four database pairs.

a)

TRANSIT-BFS	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates
F4	279	38,006	3,026	2,832	968
F7	255	42,010	3,651	3,481	749
F14	32,695	12,524,800	317,076	686,454	269,075
F1	12,742	5,457,202	89,424	159,695	42,421

OT[1]	TG[2]	OG[3]
136.22	0.09	12.56
164.75	0.07	11.51
383.08	0.1	39.5
428.28	0.14	61.03

TRANSIT-DFS	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates
F4	4,275	603,971	4,204	4,417	4,483
F7	1,433	226,655	1,609	1,625	871
F14	5,131	1,909,040	6,055	7,238	6,535
F1	95	36,986	1,134	373	32

OT	TG	OG
141.28	1.02	143.67
158.17	0.89	140.87
372.06	0.85	315.28
389.33	0.08	32.62

[1] OT The number of operations executed per tested database
[2] TG Percentage of tested databases from those added to the graph
[3] OG The number of operations executed per added database in the graph

b)

Initial Database	Valid Operations	Databases Added	Edges Generated
F4	124	106	110
F7	164	149	159
F14	386	320	386
F1	451	398	441

c)

Final Graph	Vertices	Edges	|T|	Δ_U
F4	6	6	2	3
F7	18	36	30	4
F14	30	76	160	5
F1	84	288	1,500	6

Figure 16: The results and effort of applying the TRANSIT algorithms

The number of executed modification operations is directly related to the number of tested databases. It is the sum of the number of valid modification operations of all tested databases. Comparing the average number of executed modification operations per tested database, shown in column OT of Figure 16 a), with the number of valid operations for each of the origins of the four database pairs (shown in Figure 16b)) reveals, that this number is quite similar for each of the generated and tested databases. This implies an exponential growths of the number of executed modification operations if no pruning is performed. The large number of valid modification operation even for these small databases suggest, that for larger databases the number of valid operations is going to explode.

For each of the tested databases, the complete set of databases derivable by a single modification operation is generated. Each of these resulting databases is classified into one of four classes:

- **Rejected**: The resulting database has a lower bound, which disqualifies it as an intermediate state of any minimal transformer. These databases are rejected from further consideration.
- **Newly Added**: The database has a sufficient lower bound and is added to the evolving transition graph.
- **Inter-Duplicate**: The database has been generated before at a lower distance level. This database represents an inter-level duplicate and no changes to the graph occur.
- **Intra-Duplicate**: The database has already been derived at the current distance level. It therefore is an intra-level duplicate. This causes the generation of an additional edge within the graph.

The large difference between the number of executed modification operations and the number of newly added and duplicate databases reveals, that most of the results from executing modification operations are pruned. The distribution of the generated databases on the four classes is exemplarily shown in Figure 17 for the database pair F14. The distributions for the other database pairs in our experiments are fairly similar to this. The figure indicates that by far the largest portion of executed modification operations results in databases which are rejected. This portion is up to 90% when using the Breadth-First approach, and even up to 99% for the Depth-First approach. The figure also shows that a great portion of the not rejected databases are duplicates. The large number of nearly 70% duplicates, for the remaining databases, justifies the effort for detecting and removing duplicates while constructing the transition graph.

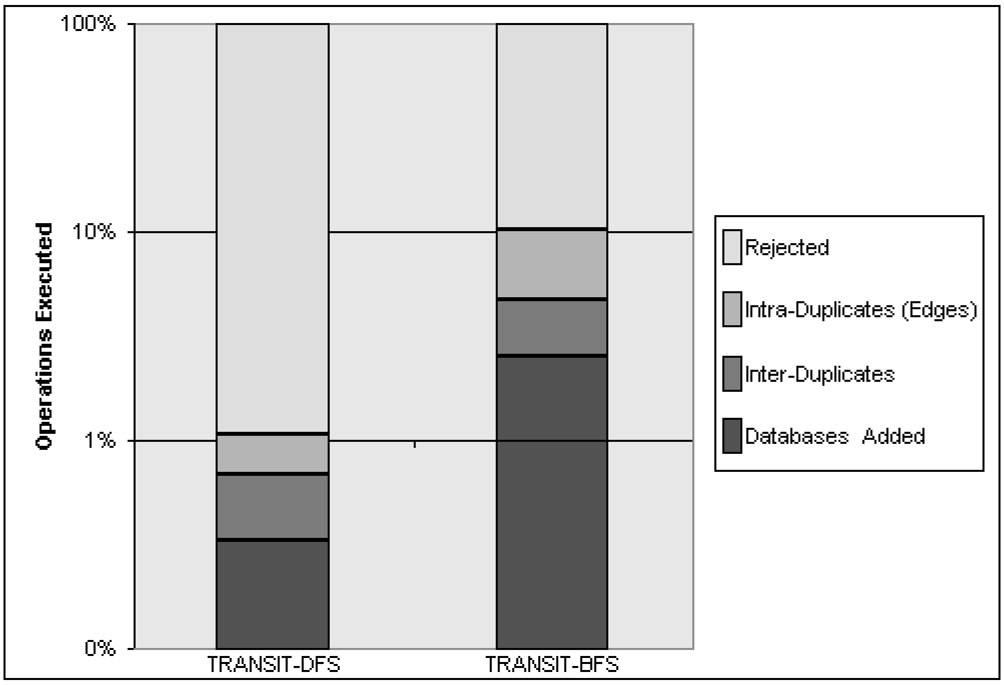


Figure 17: What happens to the results of executing the modification operations

The ability of pruning also is shown in Figure 18 for TRANSIT-DFS and F14. The total number of databases added to the transition graph is far below the number of executed modification operations. On the other hand, pruning of databases once added to the graph is not very effective. Therefore, the number of databases in the graph grows linearly with the number of added databases. This linear growth indicates that the number added databases is approximately equal for all tested databases. Thus, despite our quite effective pruning, the number of databases added to the generated transition graph remains large. This is especially true, if we compare this number with the number of databases in the final transition graph (shown in Figure 16 c)). This leads to the problem, that the memory requirement of the transition graph is very large, which makes it impossible even for database pairs of mediocre size to maintain the graph completely in main memory.

Comparing the two approaches shows, that each of them has their strength an weaknesses. The depth-first approach is inferior for F4, where the optimal solution requires the insertion of conflicts at first. In all other cases the depth-first performs better than the breadth-first approach with respect to the number of databases added and tested. The ratio of these numbers (shown in column TG of Figure 16 a)) shows that the depth-first usually processes over 80% of the added databases as candidates at the next distance level while the breadth-first approach adds numerous databases to the transition graph that are never considered as candidates afterwards. A special case is the numbers of added and tested databases by the depth-first approach for F4, where the ration is above 1. This is due to those databases that are added once but tested several times at different (decreasing) distance levels.

Figure 19 shows advantage of TRANSIT-DFS over TRANSIF-BFS. There we lists the number of added and tested databases for both approaches at each distance level for the pair of databases F14. The breadth-first approach tends to peak at lower distance levels due to the limited pruning ability of the overall bound at earlier stages of processing. On the other hand, due to finding a first solution at an early stage, the depth-first approach has a better ability of pruning databases at the lower distance levels. Still, the number of databases in the generated transition graph is far above the number of databases in the minimal transition graph. This is true for all experimental dataset as comparison of the respective values in Figure 16 a) and Figure 16 c) shows.

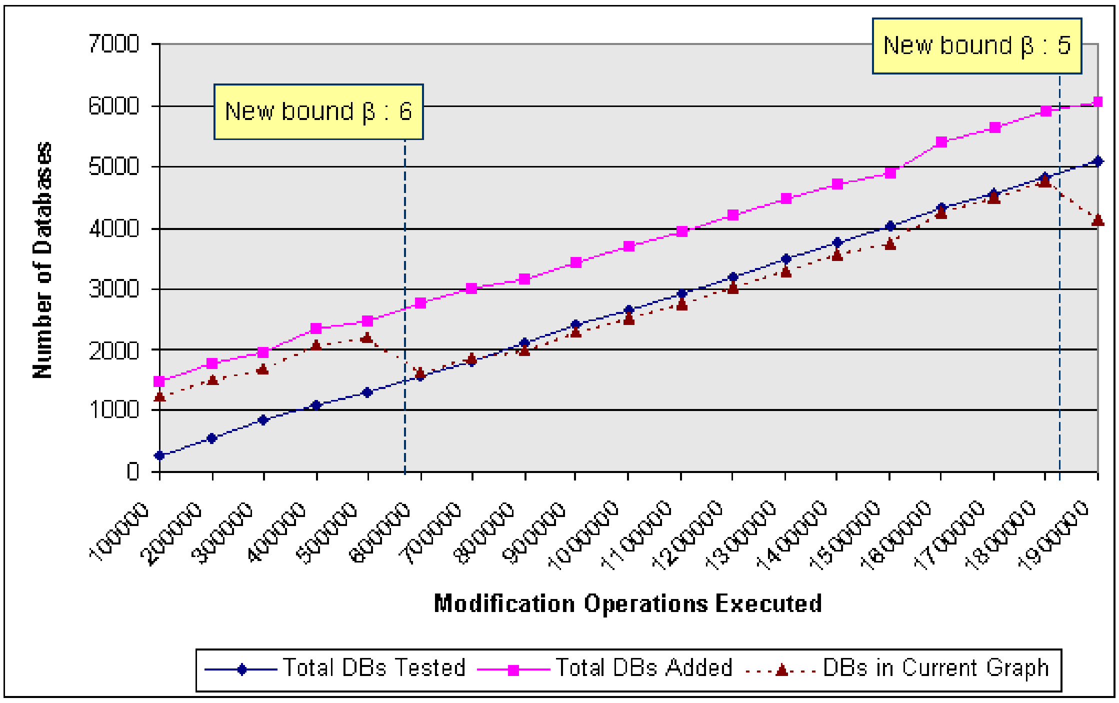


Figure 18: Development of the transition graph for TRANSIT-DFS on dataset F14.

6.2 Accuracy of the Heuristic Approaches

This subsection compares the effort and accuracy of the described heuristics with the exact algorithms. We start with restricting the set of valid modification operations.

6.2.1 Different Operation Classes

Figure 20 shows the change in effort for TRANSIT-DFS with different classes of modification operations. There is first drop-off in the number of databases tested and added when disabling the insertion of proxy values (CLASS 0 – Proxies). This restriction reduces for each attribute the number of valid modification operations by approximately the number of closed patterns for the database. The operation classes 1 to 3 do not consider proxies by definition. The largest improvement is gained by disallowing operations which increase the number of conflicts.

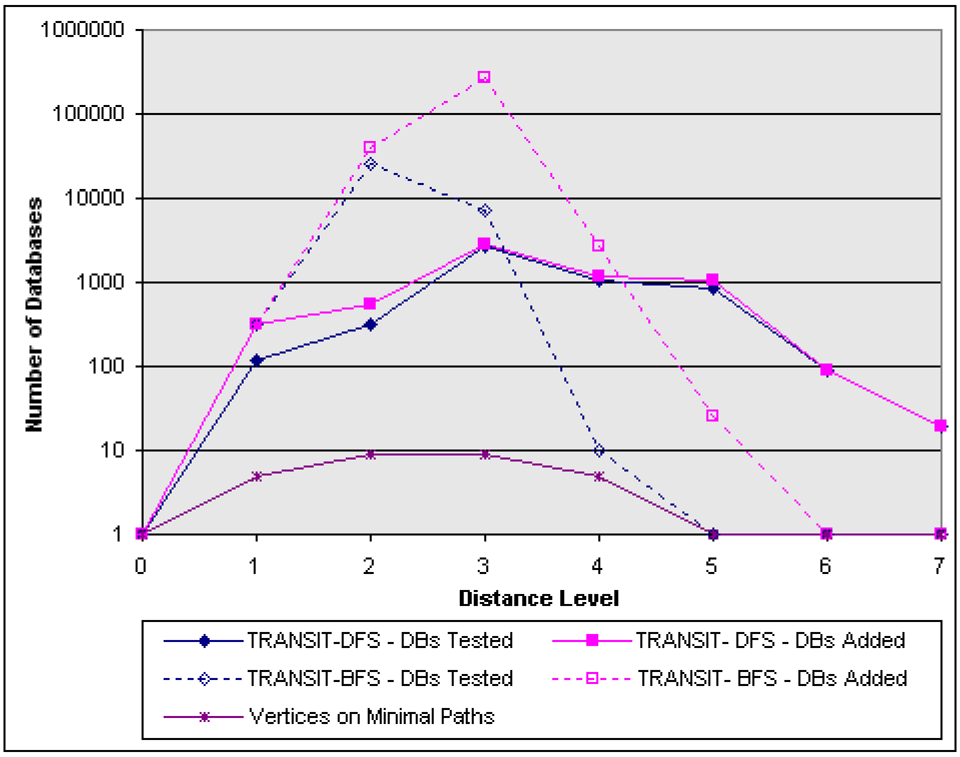


Figure 19: Comparing the exact algorithms for dataset F14.

TRANSIT-DFS F4	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0 + Proxies	4,275	603,971	4,204	4,417	4,483	3
CLASS 0 - Proxies	1,384	134,906	1,384	1,578	1,504	4
CLASS 1	36	104	36	38	10	4
CLASS 2	31	80	31	49	0	5
CLASS 3	31	80	31	49	0	5

TRANSIT-DFS F7	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0 + Proxies	1,433	226,655	1,609	1,625	871	4
CLASS 0 - Proxies	760	82,440	883	970	501	4
CLASS 1	105	1,012	115	102	73	4
CLASS 2	177	1,522	197	209	161	4
CLASS 3	177	1,522	197	209	161	4

TRANSIT-DFS F14	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0 + Proxies	5,131	1,909,040	6,055	7,238	6,535	5
CLASS 0 - Proxies	2,108	498,191	2,752	3,717	2,914	5
CLASS 1	458	7,960	498	631	359	5
CLASS 2	359	5,398	385	467	315	5
CLASS 3	359	5,398	385	467	315	5

TRANSIT-DFS F1	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U
CLASS 0 + Proxies	95	36,986	1,134	373	32	6
CLASS 0 - Proxies	95	29,574	955	368	28	6
CLASS 1	112	1,377	165	259	0	6
CLASS 2	95	1,068	129	241	0	6
CLASS 3	63	696	97	147	0	6

Figure 20: Using the depth-first approach with different classes of modification operations.

The gain of effort for the last two classes is not significant, as the number of valid modification operations is only marginally reduced compared to class 1. The figure also shows, that in some cases the banishment of valid modification operations may increase the effort, as some paths disappear, that reach the solution faster. The drop-off in accuracy (shown in the last column) is due to the fact, that the modification operations for an exact solution are no longer valid in some cases (for example in `F4`, where we need class 0 operations and proxies).

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

In Figure 21 we show the necessary effort to determining the set of minimal transformers, when allowing all operations and proxy values, but using the group solution cost as the lower bound. In our experiments, these heuristics always computed the correct update distance. The resulting transition graphs are in general smaller (the exception is `F1`, where the exact transition graph is found). The missing vertices and edges within the final transition graph result in missing some of the minimal transformers (the total number is shown in the last column of Figure 21).

TRANSIT-BFS (GS)	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U	|T|
F4	4	499	27	2	0	3	1
F7	22	3,685	172	31	12	4	24
F14	957	391,067	7,319	6,092	3,042	5	72
F1	5,049	2,232,558	31,956	54,843	22,657	6	1,500

TRANSIT-DFS (GS)	Databases Tested	Operations Executed	Databases Added	Intra-Duplicates	Inter-Duplicates	Δ_U	|T|
F4	3	402	23	2	0	3	1
F7	16	2,648	124	25	0	4	24
F14	18	7,226	803	221	34	5	72
F1	83	32,221	1,009	329	26	6	1,500

Figure 21: The necessary effort with group selection cost as lower bound.

Compared to the numbers in Figure 16 a) for the exact solution, the effort for the heuristic approach is lower than for the according exact approach. The improvement is especially significant for the first three database pairs, where we are able to reduce the number of databases tested and generated of up to 99%. The improvement is only marginal for the database pair `F1`, where the lower bound equals the actual update distance, while the initial approximation is greater. As a downside, the computation cost may increase due to the computation of the group solution cost. This is especially true for TRANSIT-BFS (GS), where the number of databases tested is larger than for TRANSIT-DFS (GS). Figure 22 compares the execution time of the two exact approaches and of TRANSIT-DFS (GS). Despite the extremely high accuracy, the computation cost (and not the memory requirements) prevents us from applying this heuristic to larger databases.

	TRANSIT-BFS	TRANSIT-DFS	TRANSIT-DFS (GS)
DBP1	366	4,311	89
DBP2	399	1,490	645
DBP3	236,596	12,747	3,621
DBP4	40,329	282	16,836

Figure 22: Execution time (in ms) for the exact algorithms and TRANSIT-DFS (GS).

6.2.3 GREEDY-TRANSIT and TRANSIT-APPROX

Figure 23 shows the resulting update distances for the four database pairs, when using the greedy approaches with different scoring functions. The group solution cost approach again determines the optimal update distance for each of the databases. This however is not always the case, as for larger databases it sometimes is inferior to the other two approaches.

Δ_{UG}	GREEDY-TRANSIT (UB)	GREEDY-TRANSIT (LB)	GREEDY-TRANSIT (GS)	TRANSIT-APPROX
F4	4	4	3	5
F7	5	5	4	6
F14	7	7	5	8
F1	7	7	6	6

Figure 23: The resulting update distance of the various greedy approaches.

In order to assess the accuracy of the greedy approaches and of the update distance approximation 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 the three algorithms GREEDY-TRANSIT (LB), GREEDY-TRANSIT (UB), and TRANSIT-APPROX. The results are shown in Figure 24. The shown values are averaged over ten runs. The results of the greedy approaches GREEDY-TRANSIT (LB) and GREEDY-TRANSIT (UB) are almost equal. Therefore, GREEDY-TRANSIT (LB) is omitted form Figure 24. 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 contradicting 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 approach outperforms the approximation in accuracy. On the other hand, the execution time for TRANSIT-APPROX is only a few milliseconds for the tested database while for the GREEDY-TRANSIT (UB) it is between 875 - 74,000 ms.

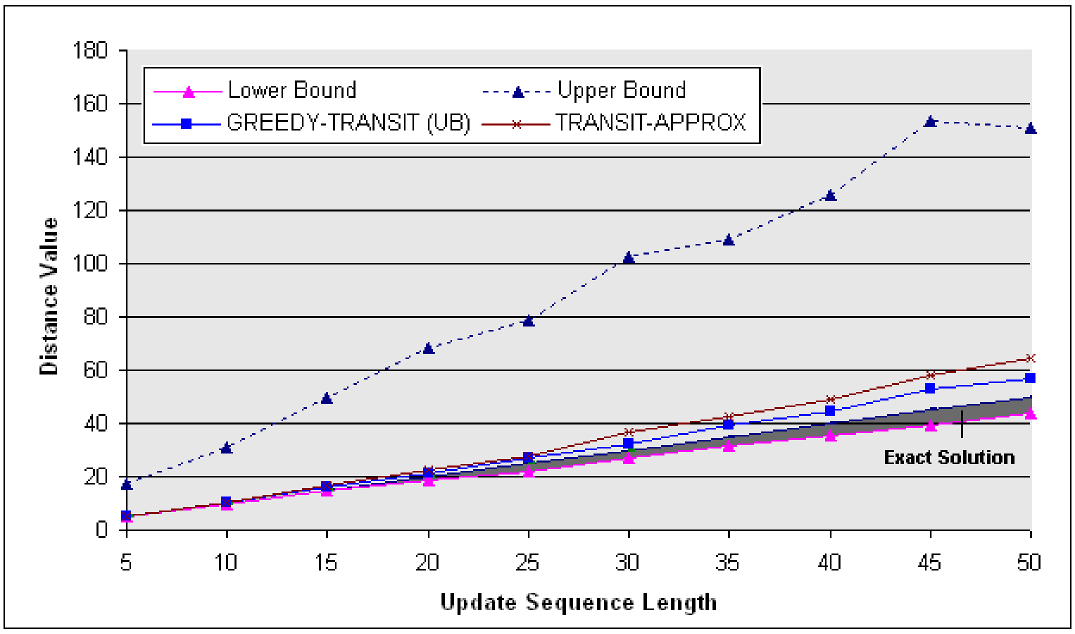


Figure 24: Comparing the accuracy of GREEDY-TRANSIT (UB) and TRANSIT-APPROX.

When generating the contradicting databases for the accuracy experiments we randomly chose one operation from the set of valid modification operations for the current database. The accuracy of GREEDY-TRANSIT (UB) and TRANSIT-APPROX decreases if we restrict the chosen modification operation to affect a minimum of n tuples.

Figure 25 shows the update distances computed by GREEDY-TRANSIT (UB) when allowing only modification operations whose patterns select at least 5, 10, or 20 tuples (results are denoted by GREEDY-TRANSIT (UB) (n)). Also shown is the resulting upper bound, i.e., number of conflicts, for the generated databases. Using patterns with higher selectivity increases the number of conflicts between the resulting databases without increasing the length of the generating sequences. While the accuracy decreases, the results are still closer to the actual update distance then the upper bound.

The decrease in accuracy is even larger for TRANSIT-APPROX. This is shown by Figure 26 where we list the computed distance values for GREEDY-TRANSIT (UB) and TRANSIT-APPROX for sequences using operations with selectivity of at least 2, 5, 10, and 20.

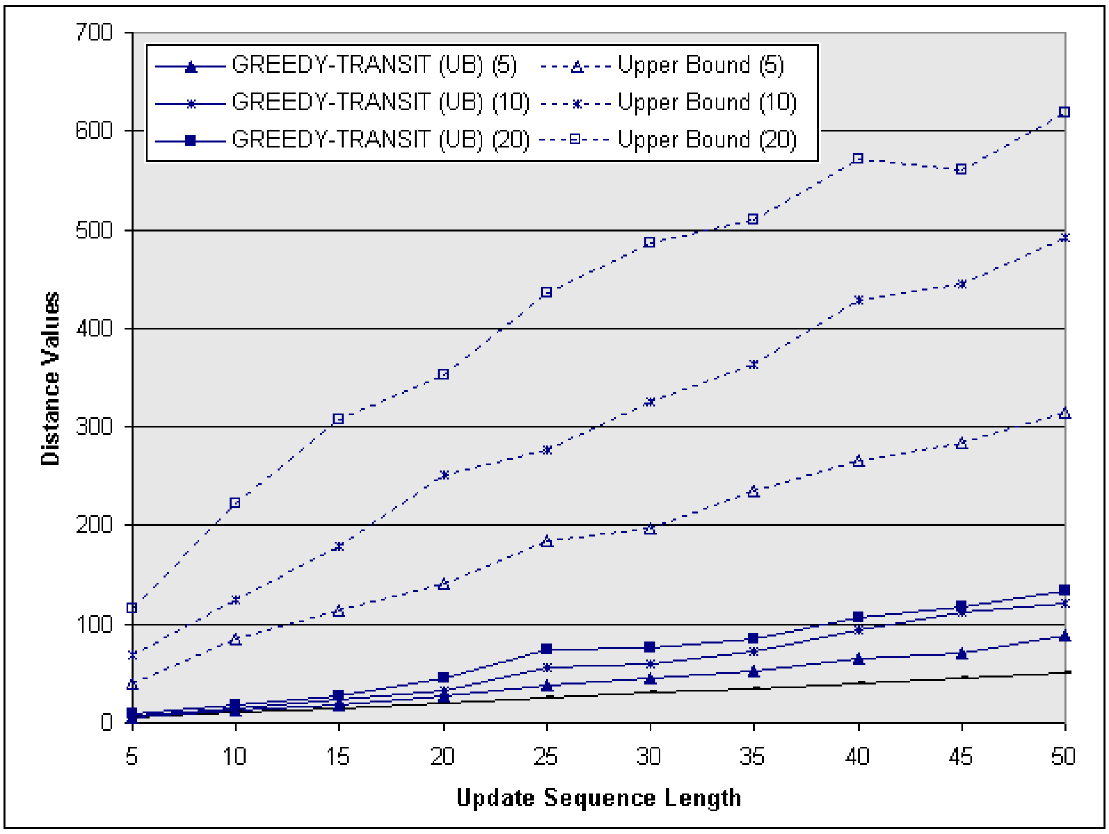


Figure 25: Accuracy of GREEDY-TRANSIT (UB) for update sequences with operations having different pattern selectivity

We also applied the greedy and the group solution cost to an artificial, randomly generated database of 20 attributes and 1,000 tuples in order to validate the applicability of the algorithms to larger databases. The accuracy is similar to the accuracy values shown in Figure 24, while the execution time is now between 3 and 9 minutes for update sequences of length 5 to 10. Application of the described algorithms on larger databases is limited by the currently used algorithm for determining the set of closed patterns. The number of patterns increases drastically as the number of tuples and especially attributes increases. We therefore need to employ a database-based mining algorithm.

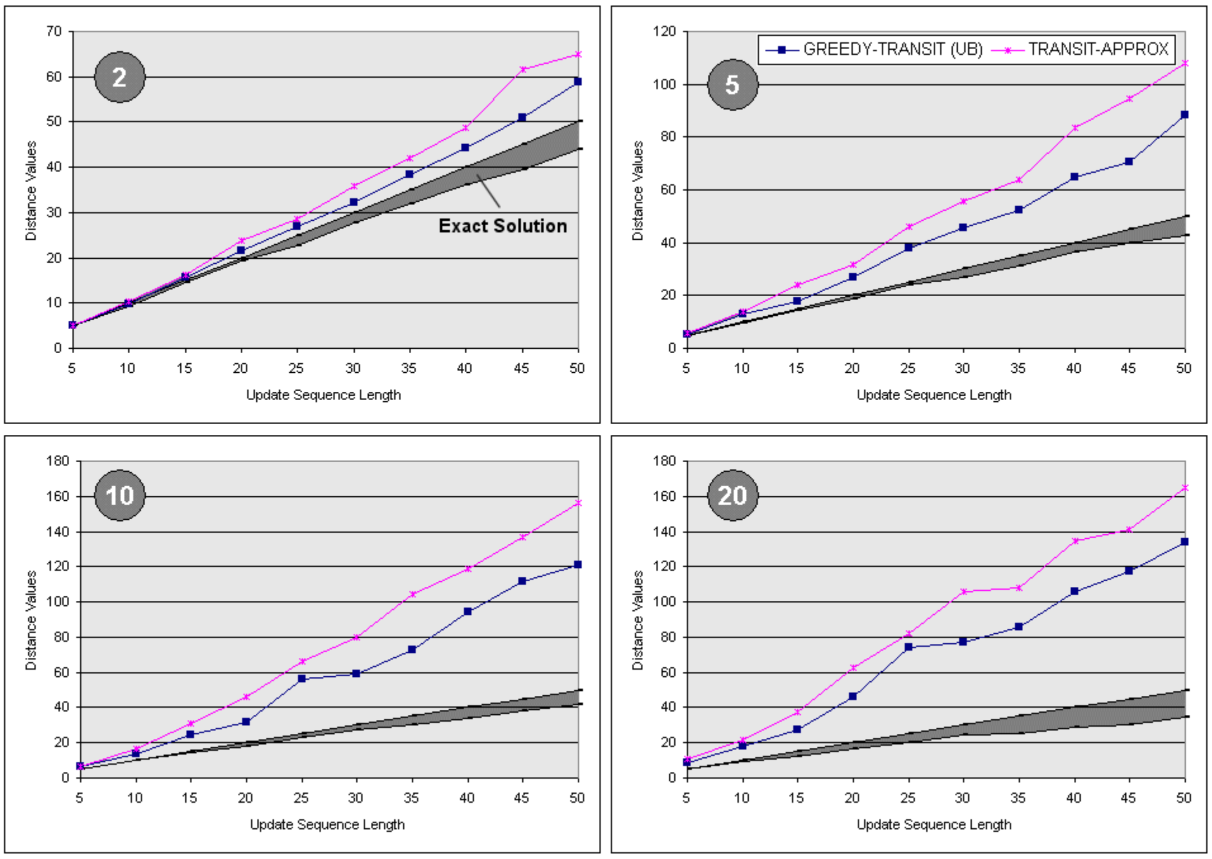


Figure 26: The accuracy of GREEDY-TRANSIT (UB) compared to TRANSIT-APPROX for sequences with operations having pattern selectivity 2, 5, 10, and 20.

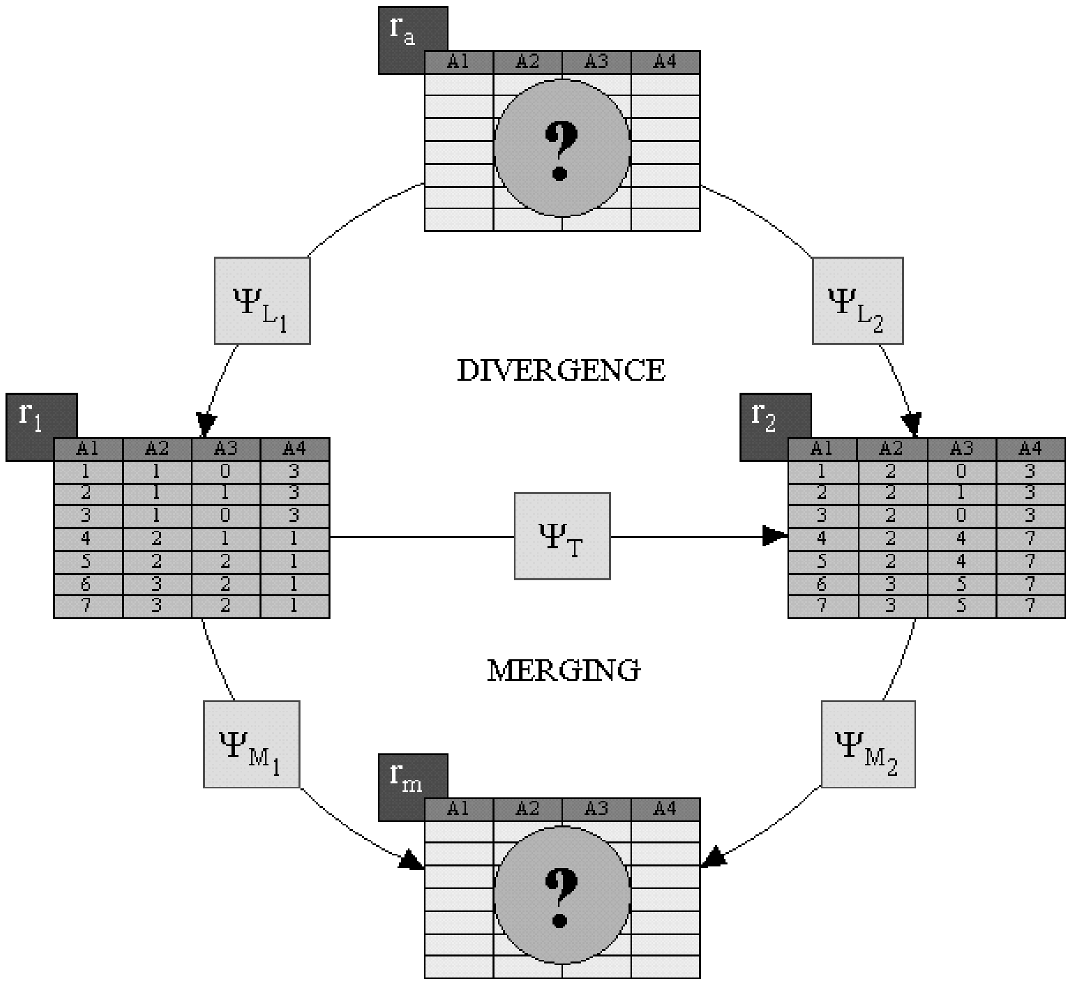
7 Further Distance Measures

We motivated our definition of an update distance between two databases by an analogous definition of the edit distance in sequence analysis. Based upon the update distance, we define two additional distance measures for pairs of contradicting databases. These definitions are motivated by the following two questions:

a) *How did a pair of databases evolve from a common ancestor?* This question is related to the phylogeny of organisms in biology.

b) *How can we transform a pair of databases into a common descendant?* This question is related to the problem of integrating a pair of databases.

The databases and modification processes surrounding these questions are depicted in Figure 27. The first question follows the assumption that a given pair of databases r_1 and r_2 evolved as modified copies of a common ancestor r_a. The modifications where performed by applying sequences of update operations Ψ_{L_1} and Ψ_{L_2} to copies of the ancestor r_a. This approach is related to the phylogeny of organisms, i.e., the evolution from a common ancestor by evolutionary events like the modification of the DNA sequence. Similar to this evolutionary process, we describe the *process of divergence* of r_1 and r_2 from r_a by the triple $(r_a, \Psi_{L_1}, \Psi_{L_2})$, with $\Psi_{L_1}(r_a) = r_1$ and $\Psi_{L_2}(r_a) = r_2$.



r_1

A1	A2	A3	A4
1	1	0	3
2	1	1	3
3	1	0	3
4	2	1	1
5	2	2	1
6	3	2	1
7	3	2	1

r_2

A1	A2	A3	A4
1	2	0	3
2	2	1	3
3	2	0	3
4	2	4	7
5	2	4	7
6	3	5	7
7	3	5	7

Figure 27: The evolution of a given pair of related data sources r_1 and r_2.

In [CWO+04] the phylogenetic distance between two organisms is defined as the total number of intermediate organisms along the lines of descent leading to their most recent common ancestor. For overlapping databases, the phylogenetic distance describes the minimal number of intermediate states for their divergence from a common, but probably unknown, ancestor. Based on LEMMA 1 any database r from $\Re(R)$, i.e., the infinite set of databases following schema R that satisfy the primary key constraint, is a common ancestor for a pair of databases, as there exists at least one transformer, which transforms r into any other database from $\Re(R)$. We again assume the simplest, i.e., shortest transformers to be the most likely explanations of the observed differences.

DEFINITION 18 (PHYLOGENETIC DISTANCE): For a pair of databases r_1 and r_2, the *phylogenetic distance*, denoted by $\Delta_P(r_1, r_2)$, is defined as the minimal number of update operations necessary to derive r_1 and r_2 from any of the possible ancestors by independent application of a pair of update sequences, i.e.,

$$\Delta_P(r_1, r_2) = \forall\ r_a \in \Re(R) : \mathit{min}(\Delta_U(r_a, r_1) + \Delta_U(r_a, r_2)).\blacklozenge$$

The challenge with determining the phylogenetic distance is to find those databases from $\Re(R)$, for which the sum of the update distances is minimal. We leave algorithms for calculating the phylogenetic distance as well as finding the common ancestor for a pair of databases as future work.

The second question results from the problem of data integration. When integrating or merging two databases, we need to solve the conflicts between them. We thereby assume a proceeding where we derive an integrated database by retaining existing values from each of the original databases. Therefore, the resulting database contains within each tuple and each attribute one of the possibly two values for this attribute from the matching partners. Tuples without a matching partner are added to the merged database as they are.

DEFINITION 19 (MERGED DATABASE): For a pair of databases r_1 and r_2, a *merged database* r_m is defined as (i) the union of the tuples without a matching partner from either source and (ii) the overlapping part of r_1 and r_2 with conflicts solved by a set of resolution function F that chose context-dependently one of the conflicting values, i.e.,

$$r_m = U(r_1, r_2) \cup U(r_2, r_1) \cup F(C(r_1, r_2)).\blacklozenge$$

In general, a resolution function $f \in$ F takes two or more values from a certain domain and returns a single value of the same domain [NH02]. Examples are well-known aggregation functions like *min*(), *max*(), etc.. Any of these resolution functions completely solves the conflicts within an attribute when applied on the whole database. In this paper we focus on context dependent conflict resolution. Context dependent conflicts represent systematic differences, which are the consequence of conflicting assumptions or interpretations in data production [FLMC01]. We adopt the assumption from [MLF04], that the conflict causing context is represented by patterns derivable from the given databases. The resolution function which we consider here are modification operations as defined above. Therefore, for r_m it holds, that

1. Each tuple t_o contained in one of the databases r_1 and r_2 is also contained in r_m, i.e.,

$$\forall\ t_o \in r_1 \cup r_2\ \exists\ t_m \in r_m : t_m[ID] = t_o[ID].$$

2. The attribute values for tuples in r_m are derived from the values of the corresponding tuples in r_1 or r_2, i.e.,

$$\forall\ t_m \in r_m\ \forall\ A \in R\ \exists\ t_o \in r_1 \cup r_2 : t_m[ID] = t_o[ID] \wedge t_m[A] = t_o[A].$$

We describe the transformation of each of the databases r_1 and r_2 into r_m by update sequences. The process of *merging a pair of databases* r_1 and r_2 into r_m is defined by the triple (r_m, Ψ_{M_1}, Ψ_{M_2}), where r_m is a common descendant of r_1 and r_2 and Ψ_{M_1} and Ψ_{M_2} describe the transformation of r_1, respectively r_2, into r_m, i.e., $\Psi_{M_1}(r_1) = r_m$ and $\Psi_{M_2}(r_2) = r_m$.

Several databases from $\Re(R)$ fulfill the described constraints of a merged database for a pair of databases. We again regard the databases requiring the shortest sequences of update operations to describe the merging as the most likely ones. This results in the following definition.

DEFINITION 20 (INTEGRATION DISTANCE): The *integration or merge distance* of a pair of data sources r_1 and r_2, denoted by $\Delta_M(r_1, r_2)$, is defined as the minimal number of update operations necessary in order to transform the sources into a merged database. Let $\Re_{valid}(r_1, r_2)$ denote the set of databases fulfilling the constraints of a merged database of r_1 and r_2. The integration distance is then defined as

$$\Delta_M(r_1, r_2) = \forall\ r_m \in \Re_{valid}(r_1, r_2): \mathit{min}(\Delta_U(r_1, r_m) + \Delta_U(r_2, r_m)).\blacklozenge$$

The development of an algorithm for calculating the integration distance of a pair of databases and for determining the most likely merged database for them is also considered as future work.

8 Related Work

To the best of our knowledge the problem of finding minimal sequences of set-oriented operations for relational databases has not been considered before. There exist various distance measures for other objects, like the well-known Hamming distance [Ham50] or the Levenshtein distance [Lev65] for binary codes and strings. Our update distance follows the Levenshtein distance, defined as the minimum number of edit operations necessary to transform one string into another. There are three main areas of related work: consistent query answering for inconsistent databases, finding patterns in conflicting data, and representing differences of databases.

The only other distance measure for databases, which is related to our definition, is defined in [ABC99]. Here, the distance of two databases is defined as the number of tuples from each of the databases without a matching partner in the other database. This definition coincides with our definition of the resolution distance when disregarding existing conflicts and regarding only the existing uncertainties. This definition is used in the area of computing consistent query answers for inconsistent databases [ABC99][CM05][Wij03]. The problem here is, given a query Q, a set of integrity constraints IC, and a database r, which violates IC, determine the set of tuples that satisfy Q and are contained in each possible repair for database r. A repair for database r is defined as a database r', which satisfies IC and is minimal in distance to r in the class of all databases satisfying IC [ABC99]. While the approaches [ABC99][CM05] only allow insertion and deletion of tuples in order to find the repairs, [Wij03] also considers the modification of existing values. Opposed to these approaches, we do not rely no integrity constraints for the identification of contradicting values. Instead, in our model the repair is already given by the target database. We therefore are not interested in finding the nearest database in a plethora of possible repairs for an inconsistent database, but in identifying update sequences that transform a given database into another given database.

The manipulation of existing database values to satisfy a given set of integrity constraints is also considered in [BFFR05]. In this approach modification as well as insertion of tuples is allowed. A certain cost is assigned with each modification and insertion operation. For a given database and a set of integrity constraints, which are violated by the database, the problem then is to find a repair, i.e., a database satisfying a given set of constraints, with minimal cost. Again, in our approach we are not interested in determining the optimal value modifications in order to solve a set of conflicts, as the solutions of existing conflicts are predetermined by the target database. Our focus is rather on how to perform the (apriori known) necessary modifications with minimal effort in terms of the number of SQL-like update operations. All other approaches described so far do not consider this problem, as they implicitly expect to modify the values one at a time, after they determined a conflict solution.

Methods for finding patterns in contradictory data to support conflict solution are for instance presented in [FLMC01] and [MLF04]. In [FLMC01], the authors discern between context dependent and context independent conflicts. Context dependent conflicts represent systematic disparities, which 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. In this sense, we are considering context dependent conflicts. However, in contrast to [FLMC01], we do not consider complex data conversion rules for conflict resolution, but always use one of the conflicting values as the solution. Discovering conflict conversion rules is considered as future work in the following section. On the other hand, we do consider the conflict causing context to be identifiable as data patterns. Therefore, this work is a continuation of our work on mining patterns in contradictory data. In [MLF04] we adopt existing data mining methods to identify patterns in overlapping databases occurring in conjunction with conflicts between them, i.e., the context in which the conflicts occur. The update operations that transform a given data source into another can be understood as a different kind of difference explaining patterns. A determined sequence of update operations for a pair of data sources may also be used as retrospective documentation of modifications performed to cleanse, standardize, or transform one of the sources into the other. The lack of documentation often hinders the interpretation of existing differences and therefore the assessment of the quality of the resulting data source.

So called “update deltas” are used in several applications to represent differences between databases. In database versioning they are used as memory effective representation of different database version [DLW84]. However, versioning collects the actual operations during execution instead of having to reengineer them from two given versions. In [LGM96] sequences of insert, delete and update operations are used to represent differences between database snapshots. In contrast to our approach, only operations that affect a single tuple are considered. Since databases are manipulated with (set-oriented) SQL commands, we consider our problem as more natural than a tuple-at-a-time approach. The detection of minimal sequences of update operations is considered in [CGM97] for hierarchically structured data. The authors consider an extended set of update operations to meet the requirements of the manipulation of hierarchically structured data. The data is represented as a tree structure and there are operations that delete, copy, or move complete sub-trees. However, the corresponding update operation, i.e., to manipulate single data values, considered in [CGM97] is tuple (or node)-at-a-time.

A main prerequisite of our approach is the ability to identify entries within the databases that represent the same real-world entity. This is known as duplicate detection or record linkage (see for example [HS95][ME97][Win99]). We assume the existence of a source-spanning object identifier for duplicate identification (the ID attribute) and therefore do not considered this problem within this paper. This identifier may be assigned to the data entries by a preceding duplicate detection step.

9 Conclusions & Outlook

We defined a distance measure for contradicting databases, based on the concept of minimal sequences of SQL-like update operations that transform one database into the other. 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.

The experimental results show, that the calculation of update distances is only practical for smaller databases, as the number of databases maintained while determining the minimal transformers growth linear, thus requesting large amounts of memory. We therefore defined several heuristics, which give up on the claim of finding the exact solution, but in turn are able to process larger databases. We performed several experiments to evaluate the accuracy of these heuristics. We found that described heuristics have a reasonable accuracy to be used as an replacement of algorithms for determining exact solutions.

In our current research work we investigate several directions. A major challenge is to reduce the computational cost and the memory requirements of our algorithms. A considerable cost factor is the necessary computation of the complete set of closed patterns for each tested database. However, since database vary only very little from their predecessors, deriving the set of closed patterns from the set of closed patterns from the parent database using some incremental approach could be highly advantageous.

There are several approaches for reducing the memory requirement of the 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 but increases the execution time for duplicate checks. We therefore investigate the possibility of efficiently detecting duplicate databases based on comparing their generating transformers from a given origin. Another approach is geared towards enhancing the pruning ability by finding upper and lower bounds that are closer to the actual update distance. However, these bounds must be computable very efficiently, as they are calculated for very many databases.

A different problem concerning the memory requirement due to the overabundance of executed operations and generated databases comes with larger databases. It is well-known, that the number of closed patterns grows immensely, as the number of tuples and the number of attributes in a database growths. A number over a million valid modification operations is not uncommon for larger databases, which in turn generates an abundance of resulting databases at each level, even in the greedy approaches. In our current experiments we where unable to compute the complete set of closed patterns for databases having over 30 attributes and 10,000 tuples. We are able to limit the number of closed patterns to those which have a support above a certain threshold. This would allow only modification operations that select a large number of tuples. As a downside, this approach is not able for any pair of databases to transform them into each other, as the modifiability of single values is no longer guaranteed (i.e., unlike in LEMMA 1 there is no guaranteed transformer for a given pair of databases). We therefore have to include those closed patterns that select the individual tuples. Limiting the number of valid modification operations by support thresholds for closed patterns eliminates the ability of find the optimal solution in some cases.

In Section 5.3 we describe an approach for approximating the actual update distance. Using this approximated update distance in a branch and bound algorithm shows promising results in terms of the accuracy of the calculated distance. However, computing the approximation currently is too costly. Finding an efficient method for group solution cost computation would yield a significant runtime improvement.

There are other variations of the described greedy approaches that enhance the accuracy of the calculated update distance. In a so-called *top-k greedy approach* we chose more than a single database at each distance level as the starting point for the next level. Given a pair of databases r_o and r_t, and an integer k, with $k \geq 1$. We start with the origin as the solely starting point. After determining the set of valid modification operations, we chose those k different databases, which receive the highest score by the applied scoring function. These databases are the starting points for the next distance level. We then build the union of databases derivable by a single operation from these databases. From this union we again chose the k databases receiving the highest weights. This process is continued until the target is reached.

Enhancing the expressiveness of update operations, including modifications like `SET A =` f`(A)` as described in [FLMC01], would be very important; yet the cost of finding such functions is probably prohibitive. Another variation is to assign different weights to the edges in the transition graph. These weights for example reflect the number of tuples actually modified by the respective operation. Using only the number of modified tuples as a weight and determining those paths of minimal weight would always result in a update distance equal with the resolution distance. While such a sequence is minimal in the number of tuples affected, it is maximal in the number of update operations executed. We therefore need to add additional cost for the execution of an update operation. This for example could be the overall number of tuples in the database, which have to be scanned while executing the selection statement of the modification operations.

Literature

[ABC99] M. Arenas, L. Bertossi, J. Chomicki. *Consistent Query Answers in Inconsistent Databases*. Proc. ACM Symposium on Principles of Database Systems (PODS), Philadelphia, Pennsylvania, 1999.

[AS94] 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.

[Bay98] 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.

[BBF+01] T.N. Bhat, P. Bourne, Z. Feng, G. Gilliland, S. Jain, V. Ravichandran, B. Schneider, K. Schneider, N. Thanki, H. Weissig, J. Westbrook and H.M. Berman. *The PDB data uniformity project*, Nucleic Acid Research, Vol. 29(1), 2001, 214-218.

[BDF+03] H. Boutselakis, et al. *E-MSD: the European Bioinformatics Institute Macromolecular Structure Database*. Nucleic Acid Research, Vol. 31(1), 2003, 458-462.

[BFFR05] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. *A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modifications*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Baltimore, Maryland, United States, 2005.

[BWF+00] 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

[CTX+04] 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.

[CGM97] S. Chawathe, H. Garcia-Molina. *Meaningful change detection in structured data*. Proc. ACM SIGMOD Int. Conf. on Management of Data Tucson, Arizona, May 1997.

[CM05] J. Chomicki, J. Marcinkowski. *Minimal-change integrity maintenance using tuple deletions*. Information and Computation, Vol. 197, No. 1/2, pp. 90-121, 2005.

[Cod70] E.F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[CWO+04] S.S. Chow, C.O. Wilke, C. Ofria, R.E. Lenski, and C. Adami. *Adaptive Radiation from Resource Competition in Digital Organisms*, Science, Vol. 305, Issue 5680, 2004, pp. 84-86.

[DLW84] P. Dadam, V.Y. Lum, H.-D. Werner. *Integration of Time Versions into a Relational Database System*. In Proc. of 10th International Conference on Very Large Data Bases, Singapore, 1984, p.p. 509-522

[FLMC01] 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.

[GDN+03] L. Gao, M. Dahlin, A. Nayate, J. Zheng, A. Iyengar. *Application Specific Data Replication for Edge Services*. In Proc. of International World Wide Web Conference (WWW2003), Budapest, Hungary, 2003.

[Ham50] R. W. Hamming. *Error-detecting and error-correcting codes*, Bell System Technical Journal Vol. 29, No. 2, 1950, pp. 147-160.

[HPY00] J. Han, J. Pei, Y. Yin. *Mining frequent patterns without candidate generation*. Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, Texas, 2000

[HS95] 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.

[INSDC] International Nucleotide Sequence Database Collaboration, http://www.insdc.org

[Lev65] V. I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals*, Doklady Akademii Nauk SSSR, 163(4):845-848, 1965 (Russian). English translation in Soviet Physics Doklady, Vol. 10, No. 8, 1966, pp. 707-710.

[LD60] A.H. Land, A.G. Doig. *An automatic method of solving discrete programming problems*. In Econometrica 28, 1960, pp. 497-520.

[LGM96] 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

[ME97] A.E. Monge, C.P. Elkan. *An efficient domain-independent algorithm for detecting approximately duplicate database tuples*. Proceedings of the SIGMOD 1997 workshop on data mining and knowledge discovery, 1997

[MLF04] 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.

[NH02] F. Naumann and M. Häussler. *Declarative Data Merging with Conflict Resolution*, Proc Int. Conf. on Information Quality (IQ 2002), Cambridge, MA.

[PBTL99] 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.

[RMT+04] 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.

[Vos91] G. Vossen. *Data Models, Database Languages and Database Management Systems*. Addison-Wesley Publishers, ISBN 0-201-41604-2, 1991

[WHP03] 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.

[Win99] 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.

[Wij03] J. Wijsen. *Condensed representation of database repairs for consistent query answering*. In Proc. of the 9th Int. Conf. on Database Theory Siena, Italy, 8 - 10 January 2003.

[ZH02] M.J. Zaki and C.-J. Hsiao. *CHARM: An efficient algorithm for closed itemset mining*. In Proc. of the Second SIAM International Conference on Data Mining, Arlington, VA, 2002.

[Ziegler] P. Ziegler. A directory of data integration projects world-wide, accessible at http://www.ifi.unizh.ch/dbtg/Staff/Ziegler/IntegrationProjects.html.