Data Fusion in Three Steps: Resolving Schema, Tuple, and Value Inconsistencies

Felix Naumann[1] Alexander Bilke[2] Jens Bleiholder[1] Melanie Weis[1]
[1] Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{naumann|bleiho|mweis}@informatik.hu-berlin.de

[2] Technische Universität Berlin
Strasse des 17. Juni 135, 10623 Berlin, Germany
bilke@cs.tu-berlin.de

Abstract

Heterogeneous and dirty data is abundant. It is stored under different, often opaque schemata, it represents identical real-world objects multiple times, causing duplicates, and it has missing values and conflicting values. Without suitable techniques for integrating and fusing such data, the data quality of an integrated system remains low. We present a suite of methods, combined in a single tool, that allows ad-hoc, declarative fusion of such data by employing schema matching, duplicate detection and data fusion.

Guided by a SQL-like query against one or more tables, we proceed in three fully automated steps: First, instance-based schema matching bridges schematic heterogeneity of the tables by aligning corresponding attributes. Next, duplicate detection techniques find multiple representations of identical real-world objects. Finally, data fusion and conflict resolution merges each duplicate into a single, consistent, and clean representation.

1 Fusing Heterogeneous, Duplicate, and Conflicting Data

The task of integrating and fusing data involves the solution of many different problems, each one in itself formidable: Apart from the technical challenges of accessing remote data, heterogeneous schemata of different data sets must be aligned, multiple but differing representations of identical real-world objects (duplicates) must be discovered, and finally the duplicates must be fused to present a clean and consistent result to a user. In particular this final step is seldomly or inadequately addressed in the literature. Figure 1 shows the three steps and the inconsistencies they bridge.

Each of these tasks has been addressed in research individually at least to some extent: (i) Access to remote sources is now state of the art of most integrated information systems, using techniques such as JDBC, wrappers, Web Services etc. Such technical heterogeneities are not addressed in this article and we assume JDBC or file-based access to the relational data sources. (ii) Schematic heterogeneity

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

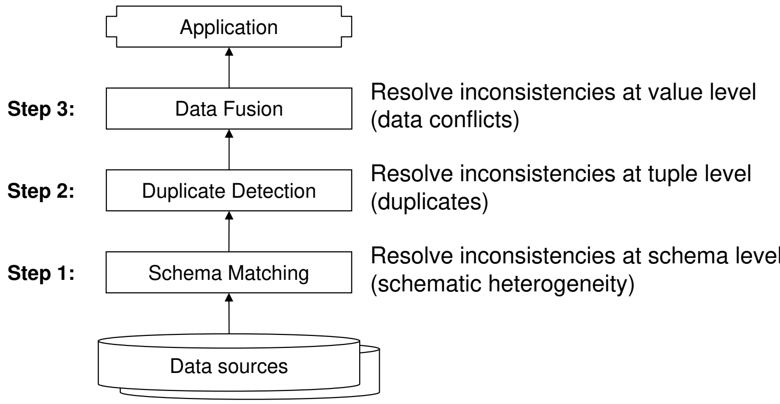

Figure 1: The three steps of data fusion

has been a research issue for at least two decades, first in the context of schema integration and then to automatically generate schema mappings. Especially recently, schema matching techniques have made great progress in automatically detecting correspondences among elements of different schemata. (iii) Duplicate detection is successful in certain domains, in particular in customer relationship management where duplicate customers and their contact information must be detected, and several research projects have presented well-suited domain-independent algorithms. Other research directions have developed domain-independent approaches. All are usually performed as an individual task, such as a separate cleansing step in an ETL procedure. Here we bed duplicate detection into a domain-independent ad-hoc querying environment. (iv) Data fusion, i.e., the step of actually merging multiple, duplicate tuples into a single representation of a real world object, has only marginally been dealt with in research and hardly at all in commercial products. The particular problem lies in resolving value-level contradictions among the different representations of a single real-world object.

We have combined all these techniques under the umbrella of the Humboldt Merger (HumMer) – a one-stop solution for fusing data from heterogeneous sources [2]. A unique feature of HumMer is that all steps are performed in an ad-hoc fashion at run-time, initiated by a user query to the sources; in a sense, we perform *ad hoc, automatic, and virtual ETL*. Apart from the known advantages of virtual data integration (up-to-dateness, low storage requirement), this on-demand approach allows for maximum flexibility: New sources can be queried immediately, albeit at the price of not generating as perfect query results as if the integration process were defined by hand. To compensate, HumMer optionally visualizes each intermediate step of data fusion and allows users to interfere: The result of schema matching can be adjusted, tuples discovered as being border-line duplicates can be separated and vice versa, and finally, resolved data conflicts can be undone and resolved manually.

Ad-hoc and automatic data fusion is useful in many scenarios: Catalog integration is a typical one-time problem for companies that have merged, but it is also of interest for shopping agents collecting data about identical products offered at different sites. A customer shopping for CDs might want to supply only the different sites to search on. The entire integration process, from finding corresponding metadata, to detecting entries for identical CDs, and finally to fuse all conflicting data, possibly favoring the data of the cheapest store, is performed under the covers. In such a scenario, a schema matching component is of special importance, as many web sites use different labels for data fields or even no labels at all.

Another application made possible only by automatic data fusion systems like HumMer is the provision of online data cleansing services. Users of such a service simply submit sets of heterogeneous and dirty data and receive a consistent and clean data set in response. Such a service is useful for individuals trying to compare different data sets, but also for organizations not wanting to employ complex ETL procedures for all data sets.

Finally, an important application is disaster data management. In an area affected by a disaster, data about damages, missing persons, hospital treatments etc. is often collected multiple times (causing duplicates) at different levels of detail (causing schematic heterogeneity) and with different levels of accuracy (causing data conflicts). Fusing such data with the help of a graphical user interface can help speed up the recovery process and for instance expedite insurance pay-outs or detect insurance fraud.

2 Components for Data Fusion

HumMer combines several larger research projects under one umbrella. In the following sections we describe each project in some detail. Related work is references intermittently, but we point out that this article can be no means be a complete survey for each of the vast fields of schema matching, duplicate detection, and data fusion.

2.1 Schema Matching and Data Transformation

When integrating autonomous data sources, we must assume that they do not conform to the same schema. Thus, the first phase in the integration process is the resolution of schematic heterogeneity. This phase proceeds in two sub-steps: schema matching, i.e., the identification of semantically equivalent schema elements, and data transformation, i.e., the bringing the data under a single common schema.

Schema matching is the (semi-automatic) process of detecting attribute correspondences between two heterogeneous schemata. Various approaches that exploit different kinds of information [19], i.e., schema information [15], instances [17], or additional metadata [14], have been proposed. As we assume the databases to contain duplicates according to our scenarios, we apply the DUMAS schema matching algorithm [3]: First, the DUMAS efficiently detects a few duplicates in two (or more) unaligned databases and then derives a schema matching based on similar attribute values of duplicates. This key idea is shown in Figure 2 where two detected duplicate tuples from different sources are used to find a schema matching.

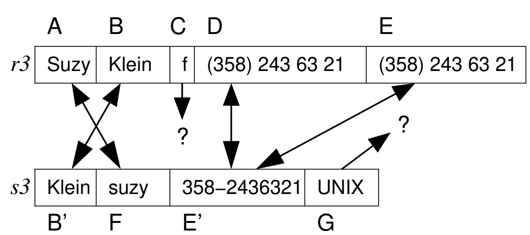

Figure 2: Schema matching using duplicates

Duplicate detection in unaligned databases is more difficult than in the usual setting, because attribute correspondences are missing, i.e., it is not known which attribute values to compare. However, the goal of this phase is not to detect all duplicates, but only as many as required for schema matching. Detecting *all* duplicates is left to the next HumMer component. DUMAS considers a tuple as a single string and applies a string similarity measure to extract the most similar tuple pairs. From the information retrieval field we adopt the well-known *TFIDF similarity* for comparing records. Experimental evaluation shows that the most similar tuples are in fact true duplicates.

These duplicates can be used for schema matching. If two duplicate tuples have the same or a sufficiently similar attribute value, we assume that these attributes correspond. Because two non-corresponding attributes might have a similar value by chance, we use several duplicates instead of only one. Two duplicates are compared field-wise using the *SoftTFIDF similarity measure* [6], resulting in a matrix containing similarity scores for each attribute combination. The matrices of each duplicate are averaged, and the maximum weight matching is computed, resulting in a set of 1:1 correspondences. Correspondences with a similarity score below a given threshold are pruned. HumMer allows users to manually add missing or delete false correspondences simultaneously across multiple data sources.

Like most related approaches, the matcher currently incorporated into HumMer is restricted to 1:1 correspondences. However, we have also developed a matcher that is able to detect complex 1:n or m:n correspondences based on the detected duplicates. The underlying idea is to combine source or target attributes and merge their respective matrix rows or columns in the matching step. The algorithm searches through the space of similarity matrices using greedy pruning, i.e., if a merge does not improve the overall matching, that branch of the search tree is not further considered.

In addition to complex matchings, we have also devised a duplicate-based matcher for schemata consisting of multiple tables. The algorithm starts with a few correspondences and crawls though the schemata by joining neighboring tables. In each step, additional correspondences are detected using the DUMAS matcher, which are used in the following steps to add more tables.

Thus with the schema matching step, schematic inconsistencies are detected and “marked” with appropriate correspondences. In the next sub-step the inconsistencies are overcome by transforming the data so that it appears under a single common schema.

The following *transformation* phase is straightforward because we assume only union-type integration: Without loss of generality, we assume that one schema is the preferred schema, which determines the names of attributes that semantically appear in multiple sources. The attributes in the non-preferred schema that participate in a correspondence are renamed accordingly. All tables receive an additional *sourceID* attribute, which is required in later stages. Finally, the full outer union of all tables is computed.

If correspondences cross multiple relations of source or target schema joins are necessary and a Clio-style data transformation becomes necessary. In this paper we assume that integration is to be performed over relations talking about same types of objects. Only then does duplicate detection and conflict resolution as described in the next sections make sense. Any more complex transformations should be performed in beforehand.

2.2 Duplicate Detection

Duplicate detection is a research area with a long tradition. Beginning with early work on record linkage [10], among many others a prominent technique for domain-dependent duplicate detection is the sorted neighborhood method [13]. More recently, several approached have emerged that regard not only data in a single table, but also data in related tables (or XML elements) to improve accuracy [1, 9, 21].

In [24] we introduce an algorithm that detects duplicates in XML documents. More precisely, duplicate XML elements are detected by considering not only their text nodes, but also those of selected children, i.e., elements involved in a 1:N relationship with the currently considered element. We map this method to the relational world (similar to [1]) to detect duplicates in a table using not only its attribute values, but also “interesting” attributes called *descriptions*, from relations that have some relationship to the current table. In this section, we first describe how descriptions are selected. Then, we introduce the duplicate detection procedure that compares tuples based on their descriptions.

2.2.1 Description Selection

Generally, we consider attributes interesting for duplicate detection being attributes that are (i) related to the currently considered object, (ii) useable by our similarity measure, and (iii) likely to distinguish duplicates from non-duplicates. We developed several heuristics to select such attributes in [25], based on descendant depth, data type, content model, optionality of elements, etc. In the relational data integration scenario descriptions are determined as follows: The attributes related to the currently considered object are attributes of the integrated table and attributes of "children tables". We consider as children all tables that contain foreign keys referencing the tables matched in the previous step. For efficiency, we limit the children tables to direct children only, i.e., no descendants reached by following more than one reference are considered.

The default description selection proposed by HumMer is the set of corresponding attributes between the two schemas, i.e., those that include values for tuples from both sources, opposed to values of non-matching attributes, which are padded with NULL values. Indeed, NULL values do not distinguish duplicates from non duplicates and thus such attributes should not be included in the description.

The heuristic of selecting only matched attributes as descriptions applies both to the already matched tables and attributes from their children tables. Therefore, the children tables need to be matched and integrated as well. Currently, children tables are matched pairwisely and the transitive closure over these pairwise matches is the final result. More specifically, let tables T_1 and T_2 be the two matched tables, and let $\{T_{1,1}, \ldots, T_{1,k}\}$ and $\{T_{2,1}, \ldots, T_{2,m}\}$ be their respective children tables. Then, every pair of tables $(T_{1,i}, T_{2,j}), 1 \leq i \leq k, i \leq j \leq m$ is matched. A given threshold determines if a pair of children tables correspond at all as shown in Figure 3. If they do, their data in the matched attributes can be used for duplicate detection as well.

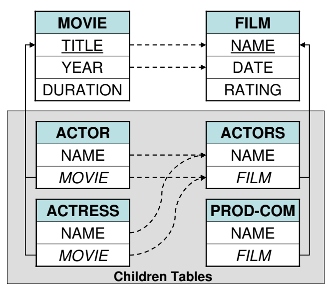

Figure 3: Matching children tables to improve duplicate detection

Once the descriptions is determined automatically, HumMer provides a comfortable means to modify the selection of interesting attributes proposed by our heuristics.

2.2.2 Duplicate Detection

After finalizing the selection of descriptions of an object, tuples are compared pairwisely using a thresholded similarity approach. More specifically, using a similarity measure $sim(t_1, t_2)$ that computes the similarity between two tuples t_1 and t_2, we classify a tuple pair as sure duplicate, possible duplicate, or non-duplicate, using two thresholds θ_{sure} and θ_{poss} in the following duplicate classifier.

$$\Gamma(t_1, t_2) = \begin{cases} t_1 \text{ and } t_2 \text{ sure duplicates} & \text{if } sim(t_1, t_2) > \theta_{sure} \\ t_1 \text{ and } t_2 \text{ possible duplicates} & \text{if } \theta_{poss} \leq sim(t_1, t_2) \leq \theta_{sure} \\ t_1 \text{ and } t_2 \text{ non-duplicates} & \text{otherwise} \end{cases}$$

The currently used similarity measure $sim()$ is proposed in [25] and takes into account (i) matched vs. unmatched attributes, (ii) data similarity between matched attributes using edit distance and numerical distance functions, (iii) the identifying power of a data item, measured by a soft version of IDF, and (iv) matched but contradictory vs. non-specified (missing) data; contradictory data reduces similarity whereas missing data has no influence on similarity. The number of pairwise comparisons are reduced by applying a filter and comparing only the remaining pairs. The filter used in combination with our similarity measure is the filter proposed in [25] that is defined as an upper bound to the similarity measure. Hence, if $sim(t_1, t_2) \leq filter(t_1, t_2) \leq \theta_{poss}$, then we can classify pair (t_1, t_2) as non-duplicate without computing the actual similarity measure, which is more complex to compute than the filter.

Pairs classified as possible duplicates are presented to the user in descending order of similarity. The user can then manually classify the pair as sure duplicate or non-duplicate. Using the descending order of similarity, users can often conclude that after classifying several pairs as non-duplicates, the remaining pairs, which are less similar, are also non duplicates.

When all pairs of sure duplicates are finally available, the transitive closure over duplicate pairs is formed to obtain clusters of objects that all represent a single real-world entity. The output of duplicate detection is the same as the input relation, but enriched by an *objectID* column for identification. Thus, inconsistencies at tuple-level are resolved: The identity of each object and its multiple representations is know. Conflicts among duplicates are resolved during conflict resolution.

2.3 Conflict Resolution

The last step in a data integration process, after *schema matching* and *duplicate detection* has been done, is to combine the different representations of a single real world object into one single representation. This step is referred to as *data fusion* and aims at resolving the still existing conflicts (uncertainties and contradictions) in the attribute values. First we show a query language to specify for each attribute a functions to resolve conflicts. Thereafter we present initial ideas to optimize queries involving data fusion.

2.3.1 Specifying data fusion

We consider *data fusion* as a step in the integration process that is guided by an (expert) user. The user specifies how the different representations and their values are used in determining the final representation, whereas a specific information system, like our HumMer system, carries out the fusion itself. In fusing data from different sources, a user can follow one of several different strategies that are repeatedly mentioned in literature [11, 16, 18, 20, 22] and categorized in [5]. Example strategies are:

- Consider all possibilities: Conflicts are *ignored* and all possible combinations of values (occasionally creating ones that have not been present in the sources before) are passed on to the user, who finally decides about which "possible world" to choose.

- Trust your friends: Specific conflicts are *avoided* by taking a preference decision beforehand and using only values from a specific source, leaving aside the (possibly conflicting) values from other sources.

- Cry with the wolves: Choosing the value that is most often used, results in *resolving* a conflict by taking one of the existing values and following the idea that correct values prevail over incorrect ones.

- Meet in the middle: Another possible way of *resolving* the conflict is in creating a new value that is a compromise of all the conflicting values, e.g., an average over several numeric values.

Data fusion in the HumMer system is implemented as a relational operator. It takes as input a number of tables containing multiple representations of a real world object and gives as output one table with exactly one representation for each real world object. This is done by grouping and aggregation, hereby using a global key to group the representations. The key needs to be provided by duplicate detection techniques applied before on the data. In each group conflicts may arise in each column that is not used for grouping. These conflicts are resolved per column by applying a conflict resolution function to the data. Functions that can be used do not only include the standard SQL aggregation functions (min, max, sum, ...) but other more elaborate functions as well, for instance functions that not only use the conflicting values in determining a final value, but also other data from the same attribute, data from other attributes or metadata as given by the query context (e.g., statistics, meta data of sources, etc.). The HumMer system is extensible allowing user defined conflict resolution functions. In the following a brief list of some functions that could be used:

- MAX / MIN: Returns the maximum/minimum value of the conflicting data values.
- GROUP: Returns a set of all conflicting values and leaves resolution to the user.
- SHORTEST / LONGEST: Chooses the value of minimum/maximum length.
- VOTE: Returns the value that appears most often among the present values. Ties could be broken by a variety of strategies, e.g., choosing randomly.
- FIRST / LAST: Takes the first/last value of all values, even if it is a NULL value.
- COALESCE: Takes the first NON-NULL value appearing.
- CHOOSE(SOURCE): Returns the value supplied by the specific source.
- MOST RECENT: Recency is evaluated with the help of another attribute or other metadata.

The fusion operation is expressed with the help of the FUSE BY statement as described in [4]. Defaults, such as using **coalesce** as the default conflict resolution function or using the order of the tables given as preference judgement, as well as implicitly removing subsumed tuples, make it easy to specify conflict resolution in an SQL-like syntax. By issuing such a FUSE BY statement the user is able to accomplish many of the different possible fusion strategies. An example for a FUSE BY statement is:

```
SELECT    Name, RESOLVE(Age, max), RESOLVE(Address, choose(EE_Students))
FUSE FROM EE_Students, CS_Students
FUSE BY   (Name)
```

This statement fuses data two student tables, leaving just one tuple per student. Students are identified by their name, conflicts in the age of the students are resolved by taking the max (assuming that people only get older), and conflicts in the address are avoided by choosing the address from source EE_Students (implementing a TRUST YOUR FRIENDS strategy).

2.3.2 Optimizing data fusion

In an information integration scenario a FUSE BY statement could be seen as a mediator, composing different sources and shaping a consistent view on these sources. Querying such a view results in a query tree combining other relational operators and the fusion operator (e.g., Figure 4). The sources used in the view may themselves be complex queries involving other relational operators as well.

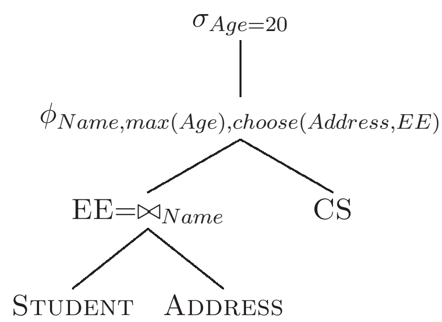

Figure 4: Query on two tables EE and CS involving Fusion and selection, assuming dirty tables CS and Student, and clean table Address (one address per name). The ϕ operator denotes fusion

For ad-hoc queries efficiency is important. To support algebraic query optimization we analyze different properties of the conflict resolution functions, e.g., commutativity, order dependance, decomposability, etc. These play an important role when deciding whether a fusion operator can be pushed down below a join, or a selection can be pushed down below a fusion, etc. Rules for decomposable, order- and duplicate insensitive functions, such as max and min, can be taken from the literature on optimization of grouping and aggregation ([12, 23]) and used in pushing down fusion beyond joins. Likewise, rules for selection pushdown below Group By for these kinds of functions also apply to fusion [7].

An example for such a transformation is in Figure 5, where early selection and fusion decreases the cardinality of intermediate results. We are currently investigating rules for the more complex functions (Vote, Choose, etc.), eventually making it necessary to use an extended relational algebra that is order-aware. The rules will be included into the query optimizer of the HumMer system. Choosing among different equivalent plans in a cost based fashion (physical query optimization) is currently only supported in the HumMer system by using a tuple-based cost model.

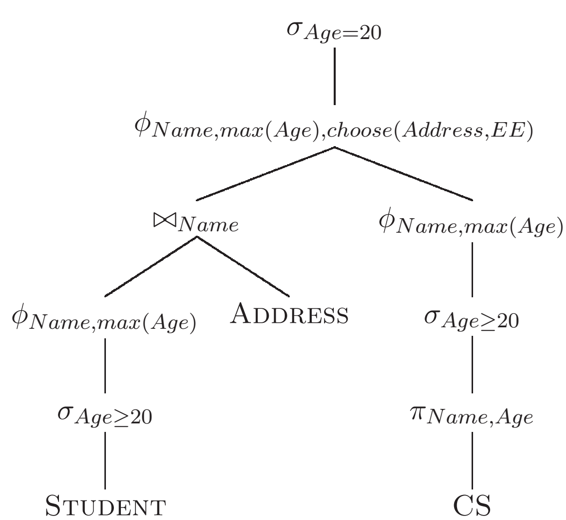

Figure 5: Query from Figure 4, optimized, decreasing the size of intermediate results by pushing selection and fusion down the tree. The ϕ operator denotes fusion

3 Composing the Individual Steps

The Humboldt Merger is implemented as a stand-alone Java application. The underlying engine of the entire process is the XXL framework, an extensible library for building database management systems [8]. This engine together with some specialized extensions handles tables and performs the necessary table fetches, joins, unions, and groupings. On top of the process lies a graphical user interface that drives the user experience. HumMer combines the techniques described in the previous section to achieve all phases of data fusion in a single system. A metadata repository stores all registered sources of data under an alias. Sources can include tables in a database, flat files, XML files, web services, etc. Since we assume relational data within the system, the metadata repository additionally stores instructions to transform data into its relational form. This section briefly describes the architecture and dataflow within the system, as shown in Fig. **??**.

HumMer works in two modes: First, querying via a basic SQL interface, which parses Fuse By queries and returns the result. Second, querying via a wizard guiding users in a step by step fashion: Given a set of aliases as chosen by the user in a query, HumMer first generates the relational form of each and passes them to the schema matching component. There, columns with same semantics are identified and renamed accordingly, favoring the first source mentioned in the query. The result is visualized by aligning corresponding attributes on the screen. Users can correct or adjust the matching result. Data transformation adds an extra sourceID column to each table to store the alias of the data source and performs a full outer union on the set of tables.

The resulting table is input to duplicate detection. If source tables are part of a larger schema, this component consults the metadata repository to fetch additional tables and generate child data to support duplicate detection. First, the schema of the merged table, along with other tables that still might reside in the databases is visualized as a tree. Heuristics determine which attributes should be used for duplicate detection. Users can optionally adjust the results of the heuristics by hand within the schema. The duplicate detection component adds yet another column to the input table – an objectID column designating tuples that represent the same real-world object. The results of duplicate detection are visualized in three segments: Sure duplicates, sure non-duplicates, and unsure cases, all of which users can decide upon individually or in summary.

The final table is then input to the conflict resolution phase, where tuples with same objectID are fused into a single tuple and conflicts among them are resolved according to the query specification. At this point, the relational engine also applies other query predicates. The final result is passed to the user to browse or use for further processing. As an added feature, data values can be color-coded to highlight uncertainties and data conflicts. Also HumMer collects lineage information for each value, so that users can see the original conflicting values and their data source. Figure 6 shows a screen shot of this final view.

4 Outlook

We conclude by reiterating the observation that surprising little work has been done in the field of data fusion to improve the quality of data. A survey of the field of duplicate detection, which would be an obvious place for authors to at least indicate what is to be done once duplicates are detected, yielded no satisfactory approaches. In fact, a common synonymous term for duplicate detection is "duplicate elimination", which precisely describes what many authors propose: Simply remove all but one representative of a duplicate group.

In the field of data integration there are indeed a few concrete approaches to data fusion as cited exemplarily in Section 2.3. Those results have not yet moved to commercial databases and applications

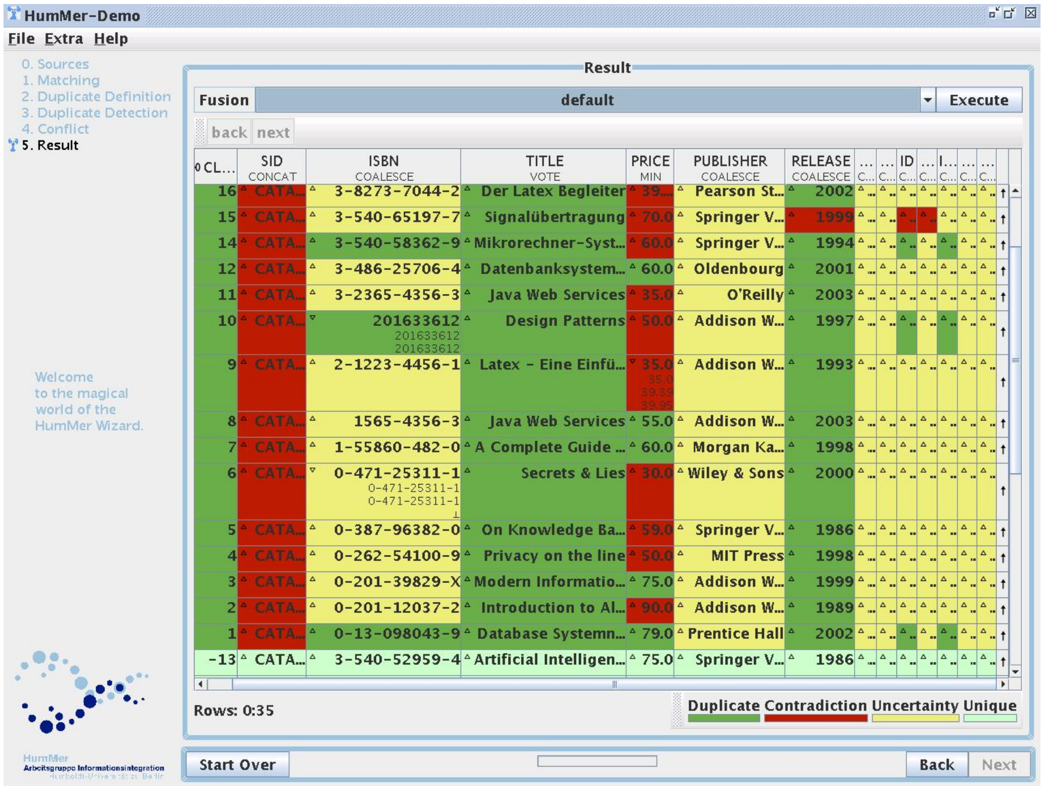

Figure 6: Screenshot of HumMer

yet, despite great efforts of vendors to extend databases to wrap heterogeneous data sources. Data fusion in real-world applications is mostly performed manually or is hard-coded into proprietary applications or ETL scripts. We believe that inclusion of data fusion capabilities into the DBMS kernel is promising.

Acknowledgments. Christoph Böhm and Karsten Draba were very helpful in implementing HumMer. This research was supported by the German Research Society (DFG grants no. NA 432 and GRK 316).

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Hong Kong, China, 2002.

[2] A. Bilke, J. Bleiholder, C. Bhm, K. Draba, F. Naumann, and M. Weis. Automatic data fusion with HumMer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2005. Demonstration.

[3] A. Bilke and F. Naumann. Schema matching using duplicates. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 69–80, Tokyo, Japan, 2005.

[4] J. Bleiholder and F. Naumann. Declarative data fusion - syntax, semantics, and implementation. In *Advances in Databases and Information Systems (ADBIS)*, Tallin, Estonia, 2005.

[5] J. Bleiholder and F. Naumann. Conflict handling strategies in an integrated information system. In *Proceedings of the International Workshop on Information Integration on the Web (IIWeb)*, Edinburgh, UK, 2006.

[6] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb)*, pages 73–78, 2003.

[7] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 342–353, Florence, Italy, 1983. Morgan Kaufmann.

[8] J. V. den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - a library approach to supporting efficient implementations of advanced database queries. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 39–48, 2001.

[9] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 85–96, 2005.

[10] I. Fellegi and A. Sunter. A theory of record linkage. *Journal of the American Statistical Association*, 64(328), 1969.

[11] A. Fuxman, E. Fazli, and R. J. Miller. Conquer: efficient management of inconsistent databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 155–166, New York, NY, USA, 2005. ACM Press.

[12] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[13] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[14] W.-S. Li and C. Clifton. SEMINT: a tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering (DKE)*, 33(1):49–84, 2000.

[15] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Rome, Italy, 2001.

[16] A. Motro, P. Anokhin, and A. C. Acar. Utility-based resolution of data inconsistencies. In *Proceedings of the 2004 international workshop on Information quality in informational systems*, pages 35–43. ACM Press, 2004.

[17] F. Naumann, C.-T. Ho, X. Tian, L. Haas, and N. Megiddo. Attribute classification using feature analysis. In *Proceedings of the International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002. Poster.

[18] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 413–424, Bombay, India, 1996.

[19] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[20] E. Schallehn, K.-U. Sattler, and G. Saake. Efficient similarity-based operations for data integration. *Data Knowl. Eng.*, 48(3):361–387, 2004.

[21] P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 297–308, 2005.

[22] V. S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross, and C. Ward. Hermes: A heterogeneous reasoning and mediator system. Technical report, University of Maryland, 1995.

[23] A. Tsois and T. K. Sellis. The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *VLDB*, pages 644–655, 2003.

[24] M. Weis and F. Naumann. Detecting duplicate objects in XML documents. In *Proceedings of the SIGMOD International Workshop on Information Quality for Information Systems (IQIS)*, Paris, France, 2004.

[25] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 431–442, Baltimore, MD, 2005.