Automatic Data Fusion with HumMer

Alexander Bilke[†] Jens Bleiholder[‡] Christoph Böhm[‡] Karsten Draba[‡]
Felix Naumann[‡] Melanie Weis[‡]

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

[‡] Humboldt-Universität zu Berlin, Germany
Unter den Linden 6, 10099 Berlin, Germany
{bleiho|boehm|kdraba|naumann|mweis}@informatik.hu-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. The Humboldt Merger (HumMer) is a tool that allows ad-hoc, declarative fusion of such data using a simple extension to SQL.

Guided by a query against multiple tables, HumMer proceeds 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 duplicates into a single, consistent, and clean representation.

1 Fusing Heterogeneous, Duplicate, and Conflicting Data

The task of 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 merged to present a clean and consistent result to a user.

Each of these tasks has been addressed individually at least to some extent. (i) Access to remote sources is now state of the art of most systems, using techniques such as JDBC, wrappers, Web Services etc. (ii) Schematic heterogeneity has been a research issue for at least two decades, first in schema integration and then in schema mapping. 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, such as address matching, and several research projects have presented domain-independent algorithms. It is usually performed as an individual task, such as the cleansing step in an ETL procedure. (iv) Data fusion, i.e., the step of actually merging multiple 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.

With the Humboldt Merger (HumMer) we present a tool that combines all these techniques to a one-stop solution for fusing data from heterogeneous sources. 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, HumMer performs *automatic and virtual ETL*. Apart from the known advantages of virtual data integration, 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. Note that these steps are optional: In the usual case, users simply formulate a data fusion query and enjoy the query result.

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

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

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 has come to our attention in the aftermath of the tsunami catastrophe. In the affected area, 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). The motivation of this scenario is similar to that of Trio [7]. 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.

2 HumMer Components

2.1 Query Language

HumMer provides a subset of SQL as a query language, which consists of Select-Project-Join queries, and allows sorting, grouping, and aggregation. In addition, we specifically support the Fuse By statement [2]. This statement is an extension of an SPJ statement specially designed for easy specification of data fusion (see Fig. 1).

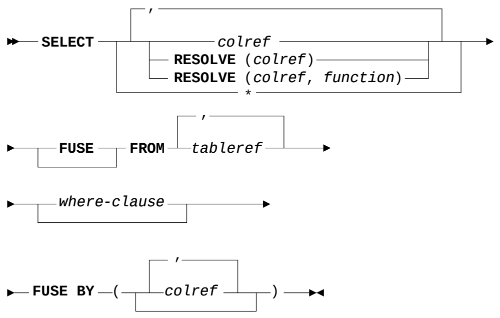

Figure 1: Syntax diagram of the Fuse By statement

The overall idea of Fuse By is to fuse tuples representing a single real-world object by grouping and aggregation. As attribute values in these tuples may contain conflicting data, Fuse By allows the use of conflict resolution functions (marked by RESOLVE). They are implemented as user defined aggregation functions and specified within the SELECT clause of the Fuse By statement. If multiple heterogeneous tables are to be fused, it is not clear until after schema matching which column names to refer to. We offer two solutions: The first is to await schema matching and only then specify column names. The second is to use only column names of one of the tables to be fused.

The FROM or FUSE FROM clause defines the tables to be fused. Join and other predicates may be applied. Using FUSE FROM combines the given tables by outer union instead of cross product. The attributes given in the FUSE BY clause serve as object identifier, and define which sets of tuples represent single real world objects. HAVING and ORDER BY keep their original meaning.

The Fuse By statement offers an intuitive default behavior: The wildcard * is replaced by all attributes present in the sources. If there is no explicit conflict resolution function, SQL's Coalesce is used as a default function. Coalesce is an n-ary function and returns its first non-null parameter value. An example for a Fuse By statement is:

```
SELECT Name, RESOLVE(Age, max)
FUSE FROM EE_Student, CS_Students
FUSE BY (Name)
```

This statement fuses data on EE_- and CS_Students, leaving just one tuple per student. Students are identified by their name, and conflicts in the age of the students are resolved by taking the higher age (assuming students only get older).

2.2 Schema Matching and Data Transformation

Because we consider autonomous databases, they must 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 steps: schema matching and data transformation.

Schema matching is the (semi-automatic) process of detecting attribute correspondences between two heterogeneous schemas. Various approaches that exploit different kinds of information, i.e., schema information, instances, or additional metadata, have been proposed. As we assume the databases to contain duplicates according to our scenarios, we apply the DUMAS schema matching algorithm [1]. First, the algorithm efficiently detects a few duplicates in two unaligned databases and then derives attribute correspondences based on similar attribute values of 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 one 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 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* [3], 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. In our demo, the correspondences are presented, allowing to manually add missing or delete false correspondences. Since data fusion can take place for more than 2 relations, HumMer is able to display correspondences simultaneously over many relations.

The following *transformation* phase is straightforward: 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.

2.3 Duplicate Detection

In [6], 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 to detect duplicates in a table using not only its attribute values, but also "interesting" attributes from relations that have some relationship to the current table. By interesting, we mean 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. Additionally, our tool provides a comfortable means to modify the selection of interesting attributes proposed by our heuristics.

Once relevant data for an object has been selected, tuples are compared pairwisely using a similarity measure that 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 (upper bound to the similarity measure) and comparing only the remaining pairs. Objects with a similarity above a given threshold are considered duplicates. 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. Conflicts among duplicates are resolved during conflict resolution.

2.4 Conflict Resolution

Conflict resolution is implemented as user defined aggregation. However, the concept of conflict resolution is more general than the concept of aggregation, because it uses the entire query context to resolve conflicts. The query context consists not only of the conflicting values themselves, but also of the corresponding tuples, all the remaining column values, and other metadata, such as column name or table name. This extension enables authors of FUSE BY statements to employ many different and powerful resolution functions.

In addition to the standard aggregation functions already available in SQL (min, max, sum,...), the following list gives further examples for functions that may be used for conflict resolution. These functions cover most of the strategies to resolve data conflicts repeatedly mentioned in the literature. Of course HumMer is extensible and new functions can be added.

CHOOSE(SOURCE): Returns the value supplied by the specific source.

COALESCE: Takes the first NON-NULL value appearing.

FIRST / LAST: Takes the first/last value of all values, even if it is a NULL value.

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.

GROUP: Returns a set of all conflicting values and leaves resolution to the user.

(ANNOTATED) CONCAT: Returns the concatenated values, including annotations, such as the data source.

SHORTEST / LONGEST: Chooses the value of minimum/maximum length according to a length measure.

MOST RECENT: Recency is evaluated with the help of another attribute or other metadata.

3 HumMer Architecture and Demo

The Humboldt-Merger is implemented as a standalone Java application. The underlying engine of the entire process is the XXL framework, an extensible library for building database management systems [4]. 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. 2.

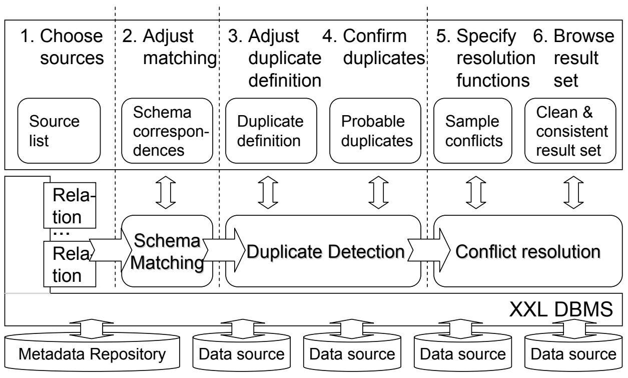

Figure 2: The HumMer framework fusing heterogeneous data in a single process

HumMer works in two modes: First, querying via a basic SQL interface, which parses entire 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. Optionally, 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 can consult 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.

For the demonstration we provide sample data from all scenarios mentioned in the introduction. Additionally, we plan to show examples taken from the recent THALIA benchmark for information integration [5].

Acknowledgments. This research was supported in part by the German Research Society (DFG grants no. NA 432 and GRK 316). We thank Louiqa Raschid for bringing to our attention the informational situation in the tsunami areas.

References

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

[2] J. Bleiholder and F. Naumann. Declarative data fusion - syntax, semantics, and implementation. In *Advances in Databases and Information Systems*, Tallin, Estonia, 2005. to appear.

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

[4] 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 *Proc. of the International Conference on Very Large Databases (VLDB)*, pages 39–48, 2001.

[5] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA: Test harness for the assessment of legacy information integration approaches. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 485–486, Tokyo, Japan, 2005.

[6] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, Baltimore, MD, 2005.

[7] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*, pages 262–276, Asilomar, CA, 2005.