

DECLARATIVE DATA MERGING WITH CONFLICT RESOLUTION

(Complete paper)

Felix Naumann, Matthias Häussler

IBM Almaden Research Center, San Jose, USA

felix@almaden.ibm.com, mhaeussler@de.ibm.com

Abstract: Database integration is a growing and increasingly important field in both research and industry. Integration requires many steps from initial schema integration and schema mapping, to data scrubbing and cleansing, and finally to data merging. While much research has concentrated on the first steps performed at *schema* level, there are only few publications about actual, practical merging of the *data* in an integrated database or in a query against multiple databases. When merging data, especially data from autonomous sources, there is a large potential for decreasing the quality of the merged data, even below the level of the original sources. The main reasons for decreased quality are data conflicts among the sources. To address this problem, we define resolution functions *merging* conflicting data. We present several alternatives of merging relational data sources with common queries through grouping & aggregating and through partitioning & joining. The resulting queries use resolution functions and can be used to migrate data from multiple sources to a target database, or to define an integrating view on multiple sources. We describe and analyze the advantages of the different approaches, and describe our practical solution in the framework of a schema mapping and data transformation tool.

Key Words: Information Integration, Databases, SQL, Data Consolidation, Data Cleansing

1 INTRODUCTION

Information integration is today one of the most important problems in information technology. This is caused both by the abundance of available information—inside an organization through ever-improving network capabilities and from outside organizations through Web services, portals etc—and by the increasing awareness of added value through information integration. The evidence for the latter reason is seen in data warehousing activities, installation of intra-organizational Web services etc.

Information integration projects and products have in common that they use some form of mappings from one or more data source schemata to a target schema [23]. The mappings either describe how source data is to be transformed to populate the target (e.g., a data warehouse), or the mappings are used for query translation from a query against the target schema to queries against the source schemata (e.g., a federated database system). In both cases, data from multiple sources is merged (or integrated) to conform to a target schema. We address the problem of ensuring important information quality properties for the merged result: We define a merging as *complete* if *all* available data about an object is accommodated in the result. We define a merging as *correct* if each real world object is represented as *only one* tuple.

Completeness of merged data is desirable to ensure that no data or relationship among data is ignored, when merging it from multiple sources. That is, all data stored about an object in all sources should be considered in some form in the target, as long as the target schema can store this data.

In particular, we study the problem of merging multiple tuples, identified as being about the same object. Regarding data about some object, sources can enforce each other, complement each other, or conflict. Sources *enforce* each other if they store the same data for the same attribute about an object. For instance, two sources reporting the same author of a book, identified by its ISBN, enforce that this person is indeed the author of that book. Sources *complement* each other if they store data about different attributes of the same object. For instance, a source storing only the publisher of a book complements a source storing only the number of pages of the same book. Sources *conflict* if they store different values for the same attribute of the same tuple. For instance, a source storing 205 as the number of pages of a book conflicts a source storing 150 as the number of pages of the same book. Resolving conflicts is the most chal-

lenging, but dealing with enforcement and complementation are also not trivial, as we shall see.

Ensuring completeness is, in effect, the problem of *aggregating* attribute values from different tuples into a final value that appears in the merged tuple.

Correctness of merged data is desirable to ensure that the combined data adheres to (possibly implicit) primary key constraints in the target database. That is, even though the data sources store multiple tuples about the same real world object, only one tuple about this object should appear in the target. To this end, one must recognize multiple tuples about the same real world object and assign a common ID. We call two tuples with the same ID *duplicates*, even if they have differing values otherwise. If there is a globally consistent ID, such as the URL or social security number, that is used and provided by all sources, this task is simple: Tuples with the same ID represent the same object and can be merged. In the absence of such an ID, *object identification* techniques can be employed [13]. These techniques find duplicates automatically by evaluating the data that is available. Even though no method performs with perfect accuracy, satisfying results have been reported [7,14,5].

Ensuring correctness is, in effect, the problem of *grouping* tuples about the same real world object into a single tuple in the merged result.

Our approach. In this paper we propose declarative merging of relational data by means of the SQL query language, so that merging can be performed by any existing database management system (DBMS). By using SQL queries for merging, our approach is not only practical but also lends itself to applications and system performing virtual integration. Commercial applications like DiscoveryLink [6], and research systems like Tukwila [11] and TSIMMIS [15] define views between sources and a global schema. Data is never materialized at the global level, so data integration and merging must be performed on the fly - using queries. Further applications of virtual integration include metasearch engines and address finders. Materialized integration systems, such as data warehouses, also profit from our research, because they can use merging queries to populate the integrated database.

Note that there are also applications where merging data is *not* wanted. There, it is important to retain the information coming from separate sources and to annotate conflicting data with its source. It is then left to a user to interpret the results. We address the needs of such applications only to a limited extent, by providing certain annotating resolution functions.

Structure of this Paper. In Section 2 we consider in more detail the problems of ensuring correctness and completeness. In particular we introduce *resolution functions* to decide on the result of merging conflicting data values. In Section 3 we present a set of useful resolution functions, including well-known aggregation functions and more sophisticated functions, such as an information quality based choice. Section 4 offers several alternatives of performing mergings using common SQL queries, each with advantages and disadvantages. These queries retrieve data from multiple sources, merge duplicates while resolving data conflicts using resolution functions, and return a “clean” data set compiled from all participating sources. Properties of the different translation approaches are examined in Section 5. Related work is reviewed in Section 6 and Section 7 concludes the paper.

2 Merging Data

Data sources can overlap in two dimensions: extensionally and intensionally. The extensional overlap between two sources is the set of real world objects that are represented in both sources. For instance, multiple sources can store information about the same book. The intensional overlap between two sources is the set of attributes both sources provide. For any given book, different sources can store the same information, such as title and author, about it.

To accommodate both types of overlap and to integrate data in a meaningful and useful way, we must recognize identical objects represented in different sources (object identification), and we must be able to resolve any data conflicts among values (conflict resolution). The following sections discuss both problems, but this paper concentrates on the latter—performing conflict resolution for integrated databases.

2.1 Object Identification

Merging data from different sources requires that different representations of the same real world object be identified as such [13]. This process is called object identification. Object identification is difficult, because the available knowledge about the objects under consideration may be incomplete, inconsistent, and sparse. A particular problem occurs if no natural identifiers (IDs) exist. For instance, the URL of a Web page is a natural ID for the page. A meta-search engine can use the URL of reported hits to find and integrate duplicates. On the other hand, a used car typically has no natural ID or sources about used cars do not store an ID. An integrated information system for used cars has no easy way of identifying a specific car being advertised in different data sources.

Object identification in the absence of IDs, which is essentially the same problem as duplicate detection, record linkage, or object fusion [19,21], is typically approached by statistical methods, for instance, using rough set theory [29]. In this paper we assume the task of object identification already performed. I.e., either each tuple in a source has a unique ID-attribute, and tuples gathered from different sources are about the same object if and only if their ID is identical. Or, as an alternative, there can exist a Boolean function taking two tuples and returning `true` if they represent the same real world object and `false` otherwise. In both cases, we are readily able to determine whether two tuples should be merged into one.

2.2 Conflict Resolution

Once different tuples have been identified as representing the same real world object, the data from them can be merged. In general, a result that is integrated from tuples of different sources, contains tuples where

1. the value for some attribute is not provided by any of the sources. Sources may not provide the value, because they do not store the particular attribute, or because they have stored a `null` value for the particular tuple. Because none of the sources provide a value, the tuple in the result has no value either (`null` value).
2. the value for some attribute is provided by exactly one source. In this case, there is also no actual data conflict. When constructing the result, the single attribute value can be used for the result tuple. If a missing value has the meaning “not applicable” instead of “unknown”, the absence of data can be taken into account as well. For the remainder, we assume the “unknown” semantics for null values.
3. the value for some attribute is provided by more than one source. This case demands special attention, because several sources compete in filling the result tuple with an attribute value. If all sources provide the same value, that value can be used in the result. If the values differ, there is a data conflict and a *resolution function* must determine what value shall appear in the result table.

Next, we define resolution functions as a way to resolve conflicts. In essence, a resolution function takes two or more values from a certain domain and returns a single value of the same domain. Additional input to the resolution function can be values from other domains. For instance, when resolving several different `prices`, the value of a `date` attribute might be used to choose the most recent price. To merge data consistently, users assign a resolution function for each attribute of the merged result. This function is then applied to every conflict among values of that attribute.

Definition 1 (Generic Resolution Function) Let D be an attribute domain and $D^+ := D \cup \perp$, where \perp represents the `null` value. Let $E^+ = \times_{i \leq o}(E_i^+)$ be the Cartesian product of further attribute domains E_i^+ . A resolution function f is an (associative) function $f: (D^+ \times E^+) \times \dots \times (D^+ \times E^+) \rightarrow D^+$.

Domain D^+ in Definition 1 represents the domain of the attribute where the conflict occurs. Domains E_i^+ represent domains of further attributes, whose values can be taken into account to resolve the conflict.

While the preceding definition is very general, Definition 2 shows a typical instance of a resolution function, reflecting the three cases of conflicting data and assuming the “unknown” semantics of null values.

Also it assumes only two input values, the extension to more input values being trivial. In essence, the default resolution function takes two values as input and returns a single value. If both values are null values, the null value is returned. If only one of the values is a null value, the other value is returned. If both input values are “proper” values a function like MAX or AVG is applied to them to obtain the result.

Definition 2 (Default Resolution function) Let D be an attribute domain and $D^+ := D \cup \perp$, where \perp represents the null value. A resolution function f is an (associative) function $f: (D^+ \times D^+) \rightarrow D^+$ with

$$f(x, y) := \begin{cases} \perp & \text{if } x = \perp \text{ and } y = \perp \\ x & \text{if } x \neq \perp \text{ and } y = \perp \\ y & \text{if } x = \perp \text{ and } y \neq \perp \\ g(x, y) & \text{else} \end{cases}$$

where $g: D \times D \rightarrow D$. Function g is called an internal (associative) resolution function.

Associativity of functions f and g is optional. If all resolution functions in a mapping are associative, allows iterative merging of data, discussed in Section 4.3.

Internal resolution functions g are various, depending on the type of attribute, the usage of the value, and many other aspects [12,28]. A simple resolution function for numerical data might return the maximum value; a simple resolution function for textual data might concatenate the values and annotate them with the source that provided the value. Section 3 presents and discusses a set of potentially useful resolution functions, and Section 4 suggests how to put them to use within SQL queries.

2.3 Batch Merging vs. Virtual Merging

We distinguish two modes of merging data. (1) Batch merging retrieves all data from the sources, merges it, and stores it locally in a database. Typically, these tasks are performed in a batch process. An example of batch merging is a data warehouse that uses Extract, Transform, and Load (ETL) tools to retrieve data from multiple sources, merge it, and materialize the merged result (see Figure 2). (2) Virtual merging provides an always up-to-date merging view on the source data. Merging views can be used by integrated information systems, which provide users with a common interface to multiple sources. The system distributes ad hoc user queries to the different databases and integrates the results on the fly. Examples include metasearch engines, and research systems like TSIMMIS [15], and commercial applications like DiscoveryLink [6]. This paper emphasizes virtual merging, i.e., defining merging queries (merging views) on the sources. Our results are also applicable to materialized merging, i.e., the merging queries can be used to populate a database.

ETL tools, such as Vality's Integrity [10] or ETI Extract [4], address materialized merging by offering large (and expensive) applications providing extensive tools for all steps of data merging: retrieving data from arbitrary sources, transforming data to the desired format, merging the data, and finally storing it in a database. In this paper we address data merging with a straightforward, lightweight, and practical solution: extending queries with existing database technology, gaining several advantages:

- Platform-independence and portability as virtually all DBMS can process SQL queries.
- Optimization by the underlying DBMS.

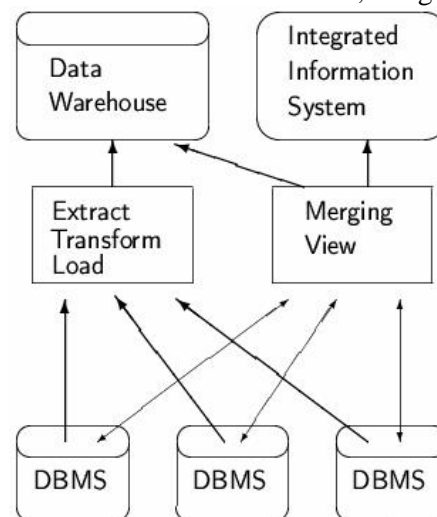


Figure 1 Batch merging in a data warehouse vs. virtual merging in an integrated information system

- Straightforward integration into existing applications, e.g. as views in the DBMS.
- Source independence, assuming relational access to the sources, for instance through wrappers.

3 Resolution Functions

The tables of this section enumerate potentially useful resolution functions. Some are well-known aggregation functions; others are specialized for the purpose of resolving data conflicts. We distinguish functions that can be applied to any data type (Table 1), functions only for numerical data types (Table 2), such as `integer`, and functions only for nonnumeric data types, such as `varchar` (Table 3). All functions can be used as scalar functions or as aggregate functions. For many functions we give an exemplary, self-explanatory attribute, for which the resolution function might be used.

COUNT	Number of non-null values, i.e., the number of conflicting values. The actual data values are lost. Exemplary attribute: none, the resulting value is stored in a special-purpose attribute.
MIN	Minimum input value. For numerical data this is the smallest value, for nonnumeric data this is the lexicographically smallest value. Exemplary attribute: <code>price</code> .
MAX	Maximum input value. For numerical data this is the largest value, for nonnumeric data this is the lexicographically largest value. Exemplary attribute: <code>number_of_children</code> .
RANDOM	Random non-null input value. This function can be used if the user does not care, from which source the data is retrieved.
CHOOSE (source)	Value provided by <code>source</code> . For instance, this resolution function can be used when other sources are known to return incorrect or untrusted values for the particular attribute. Also, this and the following function are useful for resolving foreign key attributes, because the functions retain the original value of a source and do not create a possibly invalid new value.
FAVOR ($E(O)$)	First non-null value, chosen along the complete ordering of all sources $E(O)$.
MAXIQ	Value of highest informational quality. Underlying this resolution function is an information quality model, such as the one suggested by Naumann et al. in [18]. The quality score can refer to a source in general, or be attribute-specific. For instance, one source might be in general of higher quality than another, so the values of this source are favored. Or a certain attribute in a source may have a higher quality than the same attribute in other sources.
GROUP	Set of all conflicting values. This resolution function is applicable only for multi-valued attributes. In effect, this function defers conflict resolution to the user. Confronted with a group of values, it is up to the user to decide on the true value.

Table 1 Resolution functions for any attribute type

SUM	Sum of all input values. Exemplary attribute: <code>income</code> .
MEDIAN	Median value, i.e., the value that has as many lower input values as higher input values
AVG	Arithmetic mean of all input values. Exemplary attribute: <code>rating</code> .
VAR	Variance of the input values. Both VAR and STDDEV (see next item) are not typical resolution functions because the semantics of input and output are different. Results of these functions are usually stored in an additional attribute.
STDDEV	Standard deviation of the input values.

Table 2 Resolution functions for numerical attributes

SHORTEST	Minimum length input value, ignoring trailing spaces. Exemplary attribute: <code>summary</code> .
LONGEST	Maximum length input value, ignoring trailing spaces. Exemplary attribute: <code>book_title</code> .
CONCAT	Concatenation of all input values. Because the result size of this (and the next) resolution function grows with the number of input values, special care must be taken, that the result remains within the limits of the data type. Exemplary attribute: <code>description</code> .
ANNCONCAT	Annotated concatenation of the input values. That is, before each part of the result value, its source is specified. Exemplary attribute: <code>interpretation</code> . As for GROUP, this resolution function defers conflict resolution to the user.

Table 3 Resolution functions for nonnumeric attributes

Commutativity is a prerequisite for resolution functions, because sources and tables in SQL queries are unordered. All presented resolution functions are commutative. *Associativity* on the other hand is not required but useful. If all resolution functions used in a merging query are associative, merging can be performed iteratively. Thus, if the merging queries are used to populate tables (materialized merging), it is possible to merge further data into the already merged result. For example, a data warehouse application can merge daily updates into the data warehouse. If the merging queries are used as integrated views (virtual merging), one can define further merging views on top of those views without compromising the correctness of the result. It is possible to retain this ability for some non-associative functions, for instance by storing `SUM` and `COUNT` to recalculate `AVG`.

4 Merge queries

This section provides techniques for perform merging with resolution functions in SQL statements. SQL is the common query language for all major relational database systems [3,20,25]. In the following sections we describe three alternatives, each with advantages and disadvantages for the merging query. The first approach is based on the SQL `GROUP BY` operator and wraps the union of individual queries to the sources with conflict resolving grouping. The second approach dismantles the result of a user query into partitions with differing conflict potential, resolves the conflicts, and reassembles the partitions. The third approach is a special case of the second approach, assuming associative resolution functions. For simplicity, we assume each source to consist of only one table. If this is not the case, each table of a source can be viewed as a separate source.

4.1 GROUP Queries

The SQL `GROUP BY` statement groups a set of tuples according to the values of one or more grouping attributes. Tuples with the same values for all grouping attributes are merged to a single tuple in the result. The values of all other attributes are combined through aggregate functions. For instance, the following query returns a table with books, their longest title, and their lowest price listed in the `books` table.

```
SELECT      isbn, LONGEST(title), MIN(price)
FROM        books
GROUP BY    isbn
```

Books stored in multiple sources have potentially many duplicates and the opportunity to merge data.

Imagine two tables `books1` and `books2` having the same schema. The following query generates the `UNION` of data from both sources and groups the result by the `isbn`. The result contains one row for each unique `isbn`, with the longest `title` and the lowest `price`. We chose the `LONGEST` resolution functions, assuming that a longer title contains more information about the book, such as its subtitle.

```
SELECT books.isbn, LONGEST(books.title),
       MIN(books.price)
FROM (
      SELECT * FROM books1
      UNION
      SELECT * FROM books2)
AS books
GROUP BY books.isbn
```

This translation can be generalized to any number of sources. If a source does not have the same schema as the other sources or as the result schema, the inner `SELECT` statement can be replaced by an appropriate mapping query that generates data for the desired schema. Missing attributes are padded with null values.

Advantages. Wrapping source queries with `GROUP BY` statements extends the query length by only a constant. Given a typical query size limit of 64 Kbytes, this small increase is especially advantageous for queries against many sources. Also, the only expensive operator added to the queries is the `GROUP BY` operator itself, which essentially involves a sorting operation. Thus, we expect good performance when executing these queries. In Section 5 we add further evidence in form of experimental studies.

When merging data from multiple sources with a single query, duplicates in the result occur for two rea-

sons:

1. The duplicates are already stored in a single source, e.g., if the source has not specified a unique identifier for tuples. We call such duplicates *intra-source* duplicates. Often such data in a source is undesired and thus constitutes *dirty data*.
2. The duplicates are stored among multiple sources. Such *inter-source* duplicates do not constitute dirty data, as each individual source might be clean. Rather, this is a typical and usually desired case of integrated information systems. Gathering data about an object from multiple sources, and merging it meaningfully enhances the overall knowledge about that object.

The *only* way to merge intra-source duplicates is through a `GROUP BY` operation as in this approach. Thus, this approach merges both inter-source duplicates and intra-source duplicates.

Disadvantages. The main disadvantage is the limited practical extensibility of this approach. An SQL `GROUP BY` statement demands an aggregate function on all target attributes not in the `GROUP BY` clause. The ANSI/ISO SQL99 standard [24] defines five aggregate functions: `AVG`, `MAX`, `MIN`, `SUM`, and `COUNT`. Typical database systems supply two additional built-in aggregate functions: `STDDEV` and `VARIANCE`. Obviously, these seven aggregate functions are not sufficient to resolve all conflicts as suggested in the previous sections. In particular, most resolution functions suggested in Section 3 cannot be expressed. Some database systems, such as Informix Dynamic Server [9], allow users to define new aggregate functions, which can then be used in queries. Other systems, such as IBM's DB2 UDB [3], Oracle DB [20], or Microsoft's SQL Server [25] do not allow such extensions, limiting the practicality of the `GROUP` approach. Wang and Zaniolo introduce the AXL system built on top of a DBMS, allowing users to define aggregate functions in such systems [26]. The authors report only slight performance penalties.

4.2 JOIN Queries

The principle idea of the JOIN-approach is to dismantle a query result that merges multiple sources into different parts, resolve conflicts within each part, and union them to a final result. Consider the two sources `books1` and `books2` of the previous example. We partition the union of the two sources into three parts: the intersection of the two (the set of tuples that have identical IDs in both sources), the part of `books1` that does not intersect with `books2`, and the part of `books2` that does not intersect with `books1`. We apply resolution functions to each part separately. Finally, the union of the three parts constitutes the final, merged result.

The query to extract the intersection is a join query between the two sources. Query 1 generates the join result of the two tables with `isbn` as join attribute and applies resolution functions to all other attributes. Notice that the resolution functions are not aggregate functions as in the previous `GROUP`-approach, but scalar functions taking a fixed set of parameters and producing a scalar output. Hence, we can use any built-in scalar function, such as `+` or `||` (concatenation), as resolution functions. Also, all major database systems support user-defined scalar functions, greatly extending the range of possible conflict resolution functions.

Query 1

```
SELECT books1.isbn,
       LONGEST(books1.title, books2.title),
       MIN(books1.price, books2.price)
FROM   books1, books2
WHERE  books1.isbn = books2.isbn
```

The query to extract the part of `books1` that does not intersect with `books2` is a selection of all tuples of `books1`—the minuend—less those tuples that are in the intersection—the subtrahend. The subtraction is performed through the `NOT IN` operator on the query producing the intersection. After applying a resolution function, a tuple in the minuend may have different data than one in the subtrahend with the same ID. In a sense, this effect is the whole point of subtraction: We remove tuples with identical IDs but differing data.

The query to extract the part of `books2` that does not intersect with `books1` is a selection of all tuples of `books2`—the minuend—less those tuples that are in the intersection—the subtrahend. The subtraction is performed through the `NOT IN` operator on the query producing the intersection. After applying a resolution function, a tuple in the minuend may have different data than one in the subtrahend with the same ID. In a sense, this effect is the whole point of subtraction: We remove tuples with identical IDs but differing data.

There is some freedom in constructing the subtrahend of the preceding query. Query 2a subtracts exactly those tuples from `books1` that are common with `books2`. Query 2b, on the other hand, subtracts not just the overlapping part of the two sources, but the entire source `books2`. The results are the same, but exe-

cution time varies. In the following two sections we explain the different variations and analyze their execution time in Section 5.

The queries for `books2` are analog. The results of all queries are combined with the UNION operator. As more sources are merged, the queries become more complicated because more combinations of sources must be considered, and the part to be subtracted from them grows.

As we have shown in the previous paragraphs, for two data sources, we must consider the (overlapping) join part between the two, and the each source individually. In general, for n sources we must consider all 2^{n-1} nonempty combinations of sources, as each combination might return some distinct tuples, to which we apply certain

resolution functions. So, for three sources we must consider the objects represented in all three sources, the objects represented in all combinations of two sources (and not in the combination of all three), and the objects represented in each of the individual sources (and not in any combination).

Subtracting overlap

Recall, that to dismantle a query, we generate each combination of participating sources, representing the minuend tuple set. From each combination we must subtract those parts that are not disjoint from all other combinations—the subtrahend tuple set. Here, we describe three schemes of subtraction, in turn increasing the complexity of the query but decreasing the number of tuples that are actually subtracted. In principle, combinations of all schemes are also possible, but for simplicity we consider only the “pure” variants.

- **A. Subtract combinations of size one.** Scheme A subtracts from a combination all sources that do not appear in therein. E.g., having three sources S_1 , S_2 , and S_3 , from the combination (S_1, S_2) we subtract combination (S_3) , that is, source S_3 . With this scheme we subtract many tuples needlessly, i.e., they do not appear in the minuend. However, the cost of calculating each subtrahend is low.
- **B. Subtract combinations of same size.** Scheme B subtracts from a combination of size k all other combinations of size k . E.g., from the combination of S_1 and S_2 we subtract combinations (S_1, S_3) and (S_2, S_3) .

This scheme lies in the middle in terms of needlessly subtracted tuples and complexity of determining the subtrahends. A further optimization is to subtract only combinations that have at least one source in common with the sources representing the minuend. Other combinations are guaranteed to have no tuple in common with the minuend. So for example, having four sources S_1, \dots, S_4 we subtract from the combination (S_1, S_2) the combinations (S_1, S_3) and (S_2, S_3) but not the combination (S_3, S_4) . We employed this technique in the experiments.

- **C. Subtract larger combinations.** Scheme C subtracts from a given combination all combinations whose size is one greater. E.g., from all tuples that appear in S_1 and S_2 we subtract those that appear in the combination (S_1, S_2, S_3) . The result is the set of tuples exclusively in S_1 and S_2 . Again, we refine this scheme by subtracting only combinations that have at least one source in common with the sources representing the minuend.

In this scheme we subtract only such tuples that actually appear in the combination from which they are subtracted. Determining these tuples, however, is a $(k+1)$ -way join for combinations of

Query 2a

```
SELECT isbn, title, price
FROM   books1
WHERE  books1.isbn NOT IN
      (
        SELECT   books1.isbn,
        FROM     books1, books2
        WHERE    books1.isbn = books2.isbn
      )
```

Query 2b

```
SELECT isbn, title, price
FROM   books1
WHERE  books1.isbn NOT IN
      (
        SELECT   books2.isbn,
        FROM     books2
      )
```

size k , and there are $\binom{n}{k}$ such combinations for n sources.

Advantages. The use of scalar functions instead of aggregate functions extends the number and type of possible resolution functions greatly. Typical DBMS supply a large number of built-in scalar functions and additionally allow users to define new external scalar functions, for instance coded in C or Java.

Recall that we assume to be able to identify tuples about same objects across sources, for example through a globally consistent ID. In absence of such an ID, there is an opportunity for object identification in the query itself. An identification function such as a similarity measure could be used for merging. Whereas the GROUP approach implicitly uses equality to group tuples, this JOIN approach states the predicate that recognizes duplicates explicitly. In the WHERE clause we specify predicates like “books1.isbn = books2.isbn”. The equality predicate can be replaced with some similarity predicate, such as “similarity(books1.title,books2.title) > 0.9”, to identify whether or not two tuples should be merged.

Disadvantages. In this approach intra-source duplicates (duplicates within one source) are not recognized and merged—only the GROUP BY operator can do this. If the source data is clean—without duplicates—this is not a problem; otherwise a GROUP query could be applied either to just the source, or to the final result. However, this solution yields the same disadvantages as stated in Section 4.1.

A much more important disadvantage is the length and complexity of the queries. The number of different partitions rises exponentially with the number of sources. Accordingly, the number of elements to subtract from each partition grows as well. This affects usability and practicality of the approach. Usability is affected, because typical DBMS limit the length of queries to 64 Kbytes or so. This limit is reached already for approximately ten sources. Practicality is affected, because query execution time is already prohibitively high for fewer than six sources (see Section 5 for details). Because these complex queries are essentially unions of smaller queries, their execution to materialize merged data can be staged, executing each part separately and inserting the result into a database. However, this approach is not possible for virtual merging, i.e., when the queries are used to define a view.

4.3 NESTED JOIN Queries

If only associative resolution functions are used in the query, it is not necessary to treat all combinations of sources separately. Instead, we use a nested merging approach: First we merge only two sources, then merge this result with the next source and so on. Associativity guarantees correctly resolved conflicts.

Query construction uses *common table expressions* to construct intermediate results. Each but the first common table expression contains three parts, which are combined with the UNION operator. Part 1 subtracts from the previous intermediate results those tuples that overlap with the current source. Part 2 identifies and re-merges this resolved overlap. Part 3 represents the non-

```

WITH
A1(isbn, price) AS
( SELECT  books1.isbn, books1.price
  FROM    books1),
A2(isbn, price) AS
( SELECT  isbn, price
  FROM    A1
  WHERE   isbn NOT IN (
    SELECT A1.isbn
    FROM   A1, books2
    WHERE (A1.isbn = books2.isbn) )
  UNION
  SELECT  A1.isbn, MAX(A1.price,books2.price)
  FROM    A1, books2
  WHERE   (A1.isbn = books2.isbn)
  UNION
  SELECT  isbn, price
  FROM    books2
  WHERE   isbn NOT IN (
    SELECT  A1.isbn
    FROM    A1, books2
    WHERE   (A1.isbn = books2.isbn) ) ),
A3(isbn, price) AS
( . . . )
SELECT *
FROM AN;

```

overlapping part of the current source. The final common table expression also represents the final merged result.

Discussion. The advantages of the NESTED JOIN approach are the same as for the JOIN approach of the previous section: The usage of UDFs as resolution functions and the ability to include a similarity-based merging instead of an ID-based merging. There are two additional advantages: (1) Because merging only occurs between two tables, UDFs must be implemented for only two parameters. (2) Queries grow only linearly with the number of sources.

The main disadvantage of this approach is the restriction to associative resolution functions. Nonassociative resolution functions produce incorrectly merged values in the final result, although there are workarounds, e.g., as discussed earlier for AVG. The disadvantage of using many join operations is not as drastic as for the pure JOIN approach, but we expect execution time to be considerably higher than for the GROUP approach.

5 Evaluation and Comparison

This section summarizes the discussion of the different approaches of Section 4. Table 4 lists the main features of the approaches. An important distinction of the GROUP queries, the three schemes for subtracting tuples from JOIN queries, and the NESTED JOIN queries is their expected execution time. In general, the higher the flexibility of an approach in using different resolution functions, the higher the response times of its queries are expected to be; the advantages gained by using the JOIN approaches are at the price of long execution times. We performed several tests confirming this trade-off.

	GROUP	JOIN A	NESTED JOIN
Resolution functions	Built-in aggr. func.	UDFs	Associative UDFs
Intra-source duplicates	Yes	No	No
Inter-source duplicates	Yes	Yes	Yes
Similarity based merging	No	Yes	Yes
Query length (n sources)	$O(n)$	$O(2^n)$	$O(n)$

Table 4 Comparison of different translations

To examine the efficiency of the different strategies and not be distracted by complex schemas, we choose a simplistic setup for the experiments: We created six relations (sources), each with an identical schema consisting of only one integer-type attribute, and each of the same size. The single attribute was used as the ID attribute. We did not use resolution functions in the queries of the experiments, as we wanted to examine the complexity of the queries and not that of various resolution functions. The execution cost of resolution functions heavily depends on their implementation and whether the DBMS can optimize their position in an execution plan. Note that each conflicting value is resolved only once in both approaches. However, JOIN queries make many calls to the scalar functions, while GROUP queries call each function only once. Thus, we would expect an advantage for the GROUP queries using aggregate functions.

For different experiments we varied relation cardinality from 100 tuples to 100,000 tuples (randomly picked integers), we varied the expected cardinality of the join result between any two tables (overlap) from 0.1 percent to 100 percent, and we varied the number of relations to be merged.

As expected, we can conclude from the experiments that the GROUP and NESTED JOIN approaches outperform the JOIN approaches dramatically. Also as expected, we observed for the JOIN queries an exponential increase of execution time with growing number of sources. Figure 2(a) shows execution times on a log scale for 1000 tuples in each source.¹ We must conclude that the JOIN approach is of little use for more than three sources, unless all resolution functions are associative, in which case the NESTED JOIN

¹ The queries were performed on DB2 v7.2 using a Pentium III processor with 750MHz. Because the experiments compare the performance of different queries, the absolute execution times are irrelevant.

can be used. GROUP queries and NESTED JOIN queries on the other hand scale well for large numbers of sources.

Interestingly, we observe scheme C to be the most performant variant of the JOIN approach, allowing the conclusion that it is more efficient to expensively calculate a precise subtrahend than to subtract an unnecessarily large subtrahend. Figure 2(b) highlights this observation. Low overlap causes small intermediate results. This is best taken advantage of scheme C, where joins between sources are used heavily. We are aware that these experiments must be read with extreme caution, as they heavily depend on many different parameters, such as buffer size, table sizes, and on optimization method, join algorithms etc. Figure 2 is meant to give only a flavor of the differences. Future work will examine further variations of the queries to help optimizers find good plans. In particular, there is a large potential of exploiting the numerous common subexpressions in the JOIN queries.

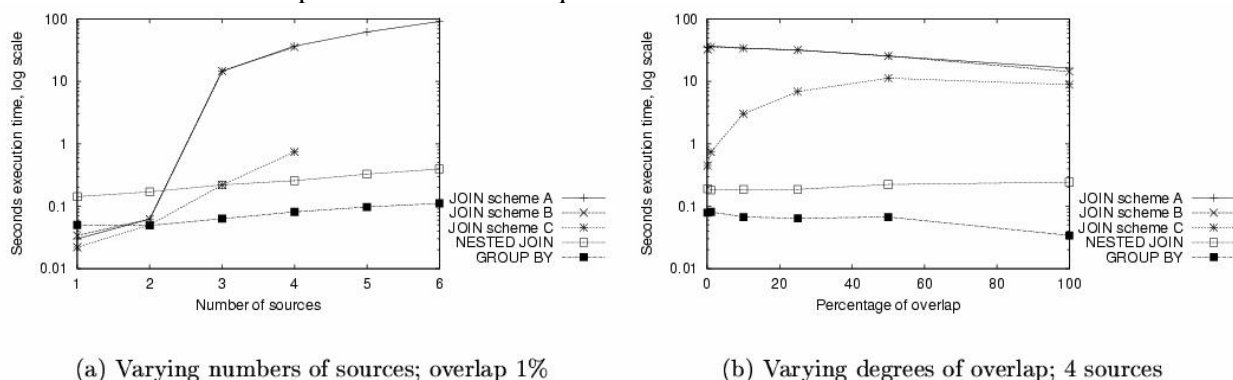


Figure 2 Execution times for all five query types

6 Related Work

The problem of conflicting data values gathered from multiple databases was first described by Dayal [2]. While the work is concerned primarily with query optimization, the author describes the problem of integrating databases as one of generalizing source databases to a target database. The need for resolution functions (there: aggregate functions) to determine the data value of an attribute in the integrated database is recognized, and several such functions are enumerated. The optimization techniques of [2] are well applicable to the type of queries we produce, so a DBMS that employs these techniques is able to execute these queries efficiently.

Among the suggestions of how to deal with conflicting data, there are several that propose to retain conflicting data and annotate the merging results in various ways. For instance, the Polygen model extends the relational model by tagging each data value with its data source [27]. The authors provide a query translation mechanism to accommodate the tags and let users select source-specific data and view data conflicts. However, such approaches must extend the query language, the data model, and query translation schemes, diminishing their usefulness. Furthermore, these approaches leave conflict resolution to the user, merely *displaying* the conflicting values, not solving the problem for very large amounts of data, or if the resolved data is processed automatically. Addressing one of these problems, The work by Lim and Chiang stores not only the conflicting values but additionally the resolved value [16]. The authors further suggest resolving conflicting values only if their “difference” does not exceed a predefined threshold. Again, users must be aware of the extended data model to make use of the added functionality and conflict resolution.

Schallehn et al. address the same problem that we discuss in this paper, namely declarative merging of data from multiple databases [22]. The author's solution is different to ours in that it extends the SQL language to solve the problem of identifying duplicates (user defined grouping), and it extends the SQL language to solve the problem of resolving conflicts among duplicates (user defined aggregation). Once

these extensions are part of a DBMS and its optimizer, they harmonize with the findings here. Galhardas et al. present a framework for declarative data cleaning [5], implemented as the Ajax tool [1]. The framework encompasses all steps of data cleaning including operators for object identification and data merging. The authors suggest a proprietary merge operator grouping tuples over ID attributes and applying functions to resolve conflicts. Details of the implementation, types of resolution functions, etc. are left open. In this paper we take a more practical approach by solving these problems with the tools at hand, i.e., with the capabilities of current database systems.

7 Conclusions

With the abundance of data sources available on the Web, data integration and data merging is becoming more and more important. When merging data, especially data from autonomous and independent sources, data value conflicts are likely to occur. In many situations it is desirable or even necessary to resolve conflicts and present a properly merged result. This paper presents two necessary components that together allow data merging with conflict resolution in current database systems.

(1) We provide a large set of potentially useful conflict resolution functions. For each function we examined several properties, among them associativity, which allows greater flexibility when constructing queries using these functions. (2) To make practical use of the resolution functions and their implied merging technique, we provide several schemes for queries that are executable by any database system. The GROUP approach is efficient, but limits the types of resolution functions that can be used. The JOIN approaches are less efficient, but allow arbitrary resolution functions.

The ideas presented in this paper have been implemented within Clio [17,8], a semi-automatic tool to support schema mapping. Our tool aids users in defining resolution functions in several ways: *Conflict Detection* determines attributes with a conflict potential, depending on the chosen query approach; *Context Sensitivity* determines which resolution functions apply to the currently selected attribute, depending on the attribute type and the chosen query approach; *Example Conflicts* show conflicting data values and their resolved values using the actual data sources.

The research presented here is a starting point for two immediate and one long-term research problems.

(1) Conflict resolution: The set of resolution functions is extensible and many domain-specific resolution functions will need to be specified. An important and complex resolution function is MAXIQ, choosing the data value of highest quality. The result of such a function could be determined by a simple quality model based for example on the *recency* of the data, or based on a more sophisticated model, such as the one presented in [18]. (2) Optimization: For the JOIN queries of Section 4.2 there are several opportunities for optimization, for instance the ones suggested in [2]. Also, these queries contain many subexpressions multiple times, giving an optimizer opportunity to recognize and optimize query execution further. (3) Data integration: The merging functionality presented here, will be included in a larger data integration/data cleansing framework, including tools for schema integration, schema mapping, data transformation, and object identification.

REFERENCES

- [1] Helena Galhardas, Daniela Florescu, Dennis Shasha and Eric Simon. AJAX: An Extensible Data Cleaning Tool. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 590, Dallas, TX, 2000, Demonstration.
- [2] Umeshwar 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.
- [3] IBM DB2 Universal Database version 7. www.ibm.com/software/data/db2/udb/
- [4] Evolutionary Technologies International - ETI Extract. www.eti.com
- [5] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon and Cristian Saita. Declarative data cleaning: language, model, and algorithms. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 371-380, Rome, Italy, 2001.