Declarative Data Fusion – Syntax, Semantics, and Implementation

Jens Bleiholder and Felix Naumann

Humboldt-Universität zu Berlin
Unter den Linden 6, D-10099 Berlin, Germany
{bleiho|naumann}@informatik.hu-berlin.de

Abstract. In today's integrating information systems *data fusion*, i.e., the merging of multiple tuples about the same real-world object into a single tuple, is left to ETL tools and other specialized software. While much attention has been paid to architecture, query languages, and query execution, the final step of actually fusing data from multiple sources into a consistent and homogeneous set is often ignored.
This paper states the formal problem of data fusion in relational databases and discusses which parts of the problem can already be solved with standard SQL. To bridge the final gap, we propose the SQL FUSE BY statement and define its syntax and semantics. A first implementation of the statement in a prototypical database system shows the usefulness and feasibility of the new operator.

1 Data Fusion

Integrated (relational) information systems provide users with only one uniform view to different (relational) data sources. Querying the underlying different data sources, combining the results, and presenting it to the user is done by the integration system.

In this paper we want to present our work on how to do the Data Fusion step in the data integration process. We rely on relational data where conflicts on the schema level already have been solved, but conflicts on the data level remain. Data Fusion is then the process of combining data about the same object from different sources by resolving occurring data conflicts. We assume object identity, that means, it is possible to distinguish between different real-world objects by a globally unique and consistent identifier. In most domains, such an identifier is already present or can easily be created, e.g., by duplicate detection methods.

Figure 1 shows three tables. The first two each represent data of a data source. In the example we talk about real persons (students), identified by their first name. We assume a domain in which these names are unique, consistent and unambiguously identify the students.

The tables overlap intensionally, as well as extensionally. They partially contain the same information about the same objects, but also complement one another: Column CAR is contained in table EE_Students but not in table

EE_Students			
Name	Age	Student	Car
Peter	⊥	no	Ford
Alice	22	yes	⊥
Bob	⊥	yes	VW
Charly	25	yes	Pontiac
Paul	26	yes	Chevy
Paul	⊥	yes	Chevy

CS_Students			
Name	Age	Student	Phone
Alice	⊥	yes	555 1234
Bob	27	⊥	555 4321
Charly	24	yes	⊥
Alice	21	no	555 9876
Mary	24	yes	⊥
Mary	24	yes	⊥

Data fusion result of EE_- and CS_Students				
Name	Age	Student	Car	Phone
Peter	⊥	no	Ford	⊥
Alice	22	yes	⊥	555 9876
Bob	27	yes	VW	555 4321
Charly	25	yes	Pontiac	⊥
Paul	26	yes	Chevy	⊥
Mary	24	yes	⊥	⊥

Fig. 1. Two data sources with conflicting data and the result of data fusion; ⊥ determines NULL values

CS_Students, column Phone is solely contained in table CS_Students. When integrating data from both tables, inter-group conflicts (e.g., *Bob* and *Charly*) and intra-group conflicts (e.g., *Paul*) on the data level can occur. We distinguish between two kinds of conflicts: a) 'uncertainty' about the value, caused by missing information, aka. NULL values in the table, and b) 'contradictions'. An example for the former would be the age of *Bob*, one for the latter the age of *Charly*.

When fusing data from the two source tables into one single table, one has to decide on how to handle these conflicts. This problem has been first mentioned by Dayal [2]. Since then, a couple of approaches and techniques have emerged, many of them trying to "avoid" the conflicts by resolving only the uncertainty of missing values. Anyhow, there is no system so far and no relational technique that is able to produce a result, such as the one given in Fig. 1 at the bottom. Therefore we propose an extension of Sql, the Fuse By statement, which not only resolves uncertainties, but also fuses tables by resolving occurring data conflicts.

Contributions. The main contributions of this paper are an extension of the Sql syntax to support data fusion operations. We provide formal semantics of data fusion in the relational model and demonstrate its feasibility in a prototypical implementation.

Structure of this paper. First, we review related work on data fusion, paying attention both to data integration systems and to individual relational operators enabling data fusion (Sec. 2). Combining the advantages of several approaches, we next define syntax and semantics of our Fuse By statement (Sec. 3). We have implemented the Fuse By statement in a prototypical RDBMS and provide

some initial insights in query processing for data fusion (Sec. 4). Finally, we conclude and point out future directions (Sec. 5).

2 Complete and concise data integration

Data integration has two broad goals: Increasing completeness and increasing conciseness of the data that is available to users and applications. An increase in completeness is achieved by adding more information sources to the system. An increase in conciseness is achieved by fusing duplicate entries and merging common attributes into one. After defining both notions, we analyze conventional and extended relational operators with respect to their ability to achieve complete and concise answers. Thus, the second part of this section covers the related work on data fusion.

2.1 Completeness

Completeness of a data set, such as a query result, measures the amount of data in that set both in terms of the number of tuples (extension) and the number of attributes (intension). *Extensional completeness* is the number of tuples in a data set in relation to the overall number of available tuples in the integrated system. Increase is achieved by adding more tuples using union-type operators. *Intensional completeness* is the number of attributes in a data set in relation to the overall number of attributes available in the integrated system. Increase is achieved by integrating sources that supply additional, yet unseen attributes to the relation using join-type operators. This distinction is along the lines of related work [7, 9, 14].

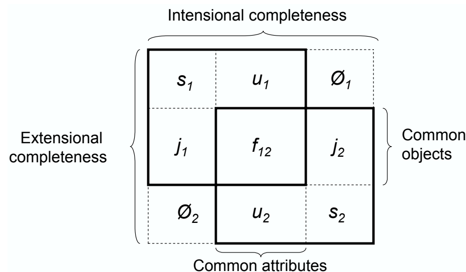

Fig. 2. Extensional and intensional completeness

To illustrate, Fig. 2 labels the different parts of a data set that is integrated from two sources. The generalization to more than two sources is trivial. The data of data source S_1 comprises areas s_1, u_1, j_1, and f_{12}. The data of source S_2 comprises the areas s_2, u_2, j_2, and f_{12}. Areas $\emptyset_1$ and $\emptyset_2$ contain only NULL values. We use the figure to describe what kind of data is produced by different operators.

2.2 Conciseness

Without knowledge of common attributes and common objects, the best that an integrating system can do is produce a result as seen in Fig. 3(a). While this result has a high completeness it is not concise. Knowledge about common attributes,

i.e., knowledge about which attribute in one source semantically corresponds to which attribute in the other source, allows results of the shape as seen in Fig. 3(b). Incidentally, this shape is the result of an outer union operation on the two source relations (assuming semantically corresponding attributes are given the same name). We call such results intensionally concise: No real-world property is represented by more than one attribute.

Knowledge about common objects, e.g., using a globally consistent ID, such as the ISBN for books, or using duplicate detection methods, allows results as seen in Fig. 3(c). Here, the only known common attribute is the ID. This result can be formed using the full outer join operation on the IDs of the two source relations. We call such results extensionally concise: No real-world object is represented by more than one tuple.

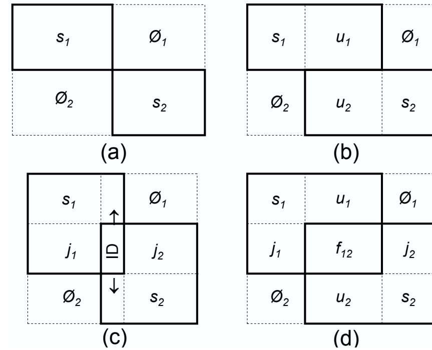

Fig. 3. Four degrees of integration

Finally, Fig. 3(d) shows the result after identifying common attributes *and* common objects. The main feature of this result is that it contains only one row per represented real-world object *and* each row has only one value per represented attribute. This result is the ultimate goal of data fusion. No common relational operator can express this result in the presence of conflicting data. The motivation of this paper is to find a way to declaratively express this result using the SQL language and some extension.

2.3 Relational operators

In the following paragraphs we analyze standard and advanced relational operators that somehow perform data integration. In particular we discuss their ability to achieve complete and concise results. Tab. 1 summarizes the discussion.

The **union join** produces results of the shape of Fig. 3(a), the most inconcise result conceivable and merely of theoretical interest. The result of the **union** ($\cup$) operator is more concise, in that it combines tuples from two union-compatible relations and removes exact duplicates. The **outer union** ($\uplus$) operator alleviates the problem of union-compatibility by adding missing attributes to both relations and padding them with NULL values [4]. Outer union increases both extensional and intensional completeness, as represented in Fig. 3(b). Conciseness is as for the union operator. The **minimum union** operator ($\oplus$) is defined by Galindo-Legaria as an outer union followed by a removal of subsumed tuples [4]. Thus, minimum union takes one step towards increased extensional conciseness: Uncertainties caused by subsumed tuples are resolved but tuples representing the same real-world objects with contradictory data remain.

Join operators assume at least one common attribute, the join attribute. The **natural join** ($\bowtie$) and **key join** ($\bowtie_{id=id}$) are not well-suited to fuse tables,

because the result contains only objects present in both source tables (low extensional completeness). This disadvantage is removed by the use of the **outer join** operations ($\overset{\circ}{\bowtie}$), which also retains all tuples of either one or both relations (left, right and full outer join). Figure 3(c) shows this result. If the join attribute is a globally consistent ID, the full outer join achieves full extensional conciseness: each real-world object is identified by that ID and appears only once in the result. However, common attributes cannot be combined as long as there are conflicts among the attribute values. Thus, intensional conciseness is low.

Yang and Özsu describe the **match join** operator used in the AURORA system [18]. It can be rewritten as an outer join of all attribute value combinations. The corresponding value of the key attribute is used to perform the join resulting in one large table. Tuples are chosen from this table according to different parameters. Extensional conciseness depends on these parameters and can reach the same level as the full outer join. The operator is able to resolve uncertainties but not conflicts. Based on the match join, Greco et al. define the **merge** ($\boxtimes$) and **prioritized merge** ($\triangleleft$) operators [6]. They are rewritten as the union of two outer joins and thus increase intensional completeness. The use of the SQL function COALESCE with the join increases intensional conciseness by resolving uncertainties. Contradictions remain and the use of union increases extensional completeness but does not increase extensional conciseness.

The notions of increasing extensional and intensional conciseness are naturally reflected by the concepts of **grouping** and **aggregation**. Even though they are standard features of most DBMS, they cannot be readily used for data fusion: There is seldom a globally consistent ID, so grouping must be based on some form of duplicate detecting similarity function instead of equality. More importantly though, most DBMS restrict aggregation functions to the few numeric functions specified in the SQL standard, i.e., COUNT, MIN, MAX, SUM, AVG, and sometimes STDDEV and VARIANCE which are not sufficient to resolve most arising conflicts.

Several projects have sought to overcome this restriction. For instance, Wang and Zaniolo introduce the AXL system to define aggregate functions in DBMS [17]. While their rewriting is already a step forward, aggregate functions allow only one input parameter, namely the column name. However, there are many cases where conflicts should be resolved by taking other data into account as well. The **FraQL** language and system, developed by Sattler et al., allows user-defined aggregates with more than one parameter [13]. They define four 2-parameter aggregation functions, each of which aggregates one column depending on the values of another column. These functions may be used to implement different conflict resolution strategies, for instance choosing values from a specific source (conflict avoidance), choosing the most recent value, or choosing all possible values (and let the user decide).

We summarize in Tab. 1 how the different data fusion operations behave concerning completeness and conciseness. A "+" marks satisfactory behavior, whereas a "−" indicates weaknesses.

Operation	Completeness int.	Completeness ext.	Conciseness int.	Conciseness ext.	Notes
Union-Join	+	+	−	−	
Union	+	+	+	−	assuming union-compatibility
Outer Union	+	+	+	−	
Minimum union	+	+	+	+/−	
Natural join	+	−	+	+	
Key join	+	−	−	+	
Outer natural-join	+	+	+	+	no intra-source duplicates
Outer key-join	+	+	−	+	no intra-source duplicates
Match join	+	+/−	−	+/−	depending on parametrization
Prioritized Merge	+	+	+	−	
User-defined grouping and aggregation	n/a	+	n/a	+	only on single table, thus no effect on intension
Data Fusion	+	+	+	+	all duplicates

Table 1. Summary of operations, compared to the ideal result of data fusion (+ marks satisfactory behaviour, − indicates weaknesses)

2.4 Data Fusion Systems

There are several integrating information systems that achieve data fusion to certain degrees: **TSIMMIS** integrates semi-structured data from multiple sources [5]. Using a rule-based language, developers of mediators can define how data is fused [10]. Special constructs specify favored data sources in case of conflicts. Values for that attribute are taken only from the favored source. Thus, without looking at other data sources, the system may not even become aware of a data conflict and so *avoids* conflicts. The **Hermes** system also integrates data in the mediator by pre-defined rules [16]. The authors explicitly name five different strategies to resolve conflicts during integration: choosing the newest data, two different strategies to choose a value depending on its source, choosing the value of numerical data, e.g., always the minimum, and choosing the value of the more *reliable* source.

Fusionplex performs data fusion by allowing advanced conflict resolution techniques [8]. Metadata, such as timestamp, cost, accuracy, availability, and clearance, is used to choose the most recent, most accurate, or cheapest data among all available data from different sources. Using this kind of source metadata reduces conflict resolution to favoring a source given some data quality criteria and therefore to conflict *avoidance* as in TSIMMIS. Using the additional metadata is possible only after extending all relational operators. As in grouping and aggregation, the value chosen for an attribute is independent of other attribute values.

Data cleansing systems are less focused on fusing data but on cleansing an existing single table. They provide simple data scrubbing methods, duplicate detection algorithms, and let users specify how duplicates are to be merged. However, typical data cleansing procedures as **Potter's Wheel** [11] or **Ajax** [3]

are implemented as separate systems and do not provide declarative data fusion operators.

3 The FUSE BY Statement

The FUSE BY statement represents a simple way of expressing queries that fuse multiple tuples describing the same object into one tuple while resolving uncertainties and contradictions. It is based on the standard SQL syntax and resembles in syntax and semantics the GROUP BY statement.

3.1 Syntax

The syntax diagram of the FUSE BY statement is shown in Fig. 4. Tuples going into the fusion process are from the tables given in the `FROM` clause. Join conditions may apply and are possible, as are subselects. `FUSE FROM` indicates combining the given tables by outer union instead of cross product, saving complex subselects in most cases as can be seen further on. Please note that when using `FUSE FROM` tuples are ordered in the order of the tables specified. (In `FUSE FROM t1, t2` all tuples from `t1` are considered before the tuples from `t2`.)

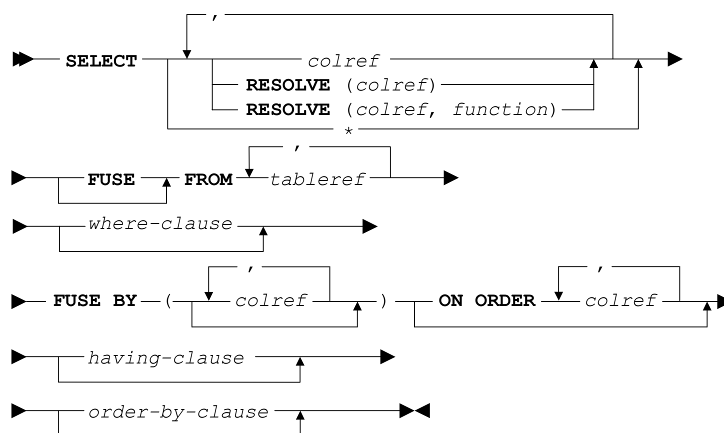

Fig. 4. Syntax diagram of the FUSE BY statement

Similar to the `GROUP BY` clause, the `FUSE BY` clause defines which objects are considered as the same real world objects, and are therefore fused into one single tuple. The attributes given here serve as identifier. `ON ORDER` influences the order in which tuples are considered when resolving conflicts. All attributes that do not appear in the `FUSE BY` clause may contain data conflicts. The keyword `RESOLVE` in the `SELECT` clause marks these columns and also serves to specify a conflict resolution function (*function*) to resolve conflicts in this column. The wildcard

'*' or not specifying a conflict resolution function results in a default conflict resolution behavior.

Keep in mind that both the HAVING-clause and ORDER BY clause can be used additionally and keep its original meaning. A small example for a FUSE BY statement is:

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

This fuses the data on EE- and CS-Students, leaving just one tuple per student. Students are identified by their name and conflicting age values are resolved by taking the higher age (assuming people only get older...).

3.2 Semantics

The overall idea behind FUSE BY is the idea of fusion by grouping and aggregation. Ideally, FUSE FROM and the outer union operator is used. Possible data conflicts are resolved in each group separately.

FUSE BY statements possess an intuitive and beneficiary default behavior: If there is no information of how to group objects, only exact duplicates and subsumed tuples are removed. If there is no information on how to resolve conflicts, known NON-NULL values in the tuples are preferred to NULL values.

Fusion process. The fusion process consists of two phases. First, all the tuples from all the sources involved are combined to form just one single table (Step 1). This increases completeness. In a second phase conciseness is being increased by grouping together tuples representing the same real world object and resolving conflicts (Step 2 through 4).
Step 1: Increasing completeness. To execute a FUSE BY statement, the tuples going into the fusion process are determined first by evaluating the FROM clause as given in the statement and eventually applying an existing WHERE condition to it. If FUSE FROM is used instead of FROM, the given tables are combined by an outer union instead of cross product. Because there is no separate outer union operator in SQL this operation needs to be rewritten (see Sec. 3.2).
Step 2: Identifying tuples to be fused. Second, all the tuples that describe one and the same real world object are grouped together. This is done by doing a grouping on the column(s) given in the FUSE BY clause. We hereby assume that we are able to rely on a globally unique and consistent identifier that we can use to do the grouping. This identifier may be produced by detecting duplicates and assigning equal keys to the same real world objects or using multiple columns as key. For this reason duplicate detection needs to be done in advance to the fusion process. Using the WHERE clause, tuples may be filtered out before the grouping.
Step 3: Increasing conciseness. Then, exact duplicates and subsumed tuples are removed per group. A tuple t_1 subsumes another tuple t_2 if they are defined

on the same attributes, t_2 has more $\perp$ values than t_1 and t_1 coincides with t_2 in all NON-NULL attributes [4]. The removal of subsumed tuples is neither a standard operation of the relational algebra, nor does there exist a specific SQL statement. Rao et al. nevertheless show how subsumed tuples can be removed from a single table [12]. However, removing subsumed tuples per group as needed in our case does not yield the same result as removing subsumed tuples from the entire table. Therefore the technique applied by [12] is not feasible in our case. All the remaining tuples of one group are then fused together to just one tuple, at the same time resolving inconsistencies and data conflicts. This is done by applying conflict resolution functions to the columns as indicated in the `RESOLVE` parts of the `SELECT` clause. More details on conflict resolution follow in Sec. 3.4.
Step 4: Shaping the result. Finally, only the desired columns as indicated in the `SELECT` clause are projected to form the final result. Additional `HAVING` and `ORDER BY` clauses are applied afterwards on this result.

Figure 5 shows the query that is used to produce the table in Fig. 1 from the introduction. Please note that the order of the tables and the order by Age influences the values chosen, e.g. the phone number of *Alice*.

```
SELECT Name, RESOLVE(Age, max), RESOLVE(Car),
       RESOLVE(Student, vote), RESOLVE(Phone)
FUSE FROM EE_Students, CS_Students
FUSE BY (Name) ON ORDER Age
```

Fig. 5. Example query that produces the Data Fusion result from Fig. 1

Rewriting fusion queries. Parts of FUSE BY can be rewritten by standard SQL and therefore directly executed by any standard DBMS. This rewriting does not include the conflict resolution functions (c.f. Sec. 3.4) and the grouping, as we show in the following paragraphs. Please reconsider the example query from Fig. 5. The rewriting of the query is shown in Fig. 6, the non-standard parts are marked by italic font.

The outer union operation as needed by `FUSE FROM`, together with the necessary order of the tuples by source table, can be rewritten as shown in lines 4 to 9. For each input table there is a `SELECT` statement with all the attributes from all tables. Attributes not present in a table are padded with NULL values. The data from the two tables is combined by `UNION ALL`. Exact intra-source duplicates as well as exact inter-source duplicates are removed by a `DISTINCT` in the enclosing `SELECT`. The extension to more than two tables is straightforward, but increases complexity of the rewritten statement.

As Union is not order preserving, the order of the tuples by table (using an additional column `src`) as required by FUSE BY is guaranteed by the `ORDER BY` in line 9, as well as the order implied by the `ON ORDER` clause of FUSE BY.

```
 1: SELECT Name, max(Age), cr_coalesce(Car), cr_vote(Student),
 2:                        cr_coalesce(Phone)
 3: FROM (SELECT DISTINCT Name, Age, Car, Student, Phone
 4:       FROM (SELECT Name, Age, Car, Student, NULL as Phone, 1 as src
 5:             FROM EE_Students
 6:              UNION ALL
 7:             SELECT Name, Age, NULL as Car, Student, Phone, 2 as src
 8:             FROM CS_Students
 9:              ORDER BY src, Age
10:            )
11:      )
12: group by Name
```

Fig. 6. Example query producing the result from Fig. 1, rewritten by means of SQL and using non standard aggregation functions

To do the grouping and prepare for conflict resolution the result is grouped by the attributes given in the `FUSE BY` clause of the statement (line 12). The attributes with the needed conflict resolution are placed in the `SELECT` clause (line 1 and 2).

Using `GROUP BY` in the rewriting requires the use of aggregation functions with all the attributes not present in the `GROUP BY` clause. As our approach allows the conflict resolution functions to be more general than aggregation functions, this part cannot be further rewritten, simply because such conflict resolution functions are not part of SQL.

As `GROUP BY` is not order preserving and we cannot influence the order in the resulting groups, only conflict resolution functions that are not order dependant can be used. As soon as order dependant conflict resolution functions are used, an order preserving version of `GROUP BY` is needed (marked by an italic *group by*). Also, `GROUP BY` does not allow for removing subsumed tuples in the groups.

Default behavior and wildcards. Wildcards, e.g., `*`, are replaced by all attributes present as given by the `FROM` or `FUSE FROM` clause, if necessary accompanied by `RESOLVE`. If no explicit conflict resolution function is given, COALESCE is used as default function. COALESCE is an n-ary function and returns its first NON-NULL parameter value. Using COALESCE as default, the order of the tuples is important and directly influences the chosen value. If no attribute is given in the `FUSE BY` clause, all tuples form one large group, performing removal of exact duplicates and subsumed tuples on all tuples in this large group.

3.3 Examples - Describing fusion queries

Query 1 of Fig. 7(a) groups the tuples of one table S1 by the values in column A. All other columns (replacing wildcard `*`) of table S1 may contain conflicting data that is resolved by the default conflict resolution function COALESCE. This

way, the statement behaves like a `GROUP BY` with a COALESCE aggregation, additionally removing subsumed tuples per group. Fusion by more than one column is possible, replacing A by all desired columns.

```
SELECT *
FROM S1
FUSE BY (A)
```

(a) Removing data conflicts

```
SELECT *
FROM S1
FUSE BY ()
```

(b) Removing exact duplicates and subsumed tuples

```
SELECT *
FUSE FROM S1, S2
FUSE BY ()
```

(c) Fusing two tables by minimum union

Fig. 7. Three simple FUSE BY statements

In Query 2 in Fig. 7(b) there is no column present in the `FUSE BY` clause. All tuples are treated equally as being in one large group. Exact duplicates and subsumed tuples are removed. Conflicts are not resolved and this corresponds to the result of a `DISTINCT` operator and the removal of subsumed tuples (indicated as $S1\downarrow$ by [4]).

Query 3 of Fig. 7(c) combines the two tables S1 and S2 by outer union. It completes missing values in columns by NULL values and removes exact duplicates and subsumed tuples. Together with COALESCE as default conflict resolution function this corresponds to the result of a DISTINCT operator and a minimum union operator [4]. Examples with three or more tables look and behave similarly.

3.4 Conflict Resolution

Different conflict resolution functions and strategies are required by different domains, thus encapsulating expert knowledge to fuse data in a domain. Nevertheless, there are some conflict resolution functions that are applicable in a wide variety of domains.

Conflict resolution functions. The concept of conflict resolution is more general than the concept of aggregation, because the functions can be arbitrarily complex and can take more data into account to compute a value. In the most general case, they can use the information given by the query context. This query context consists not only of the conflicting values themselves, but may also consist of the corresponding tuples, all remaining column values or other metadata (e.g. column or table name). This extension of aggregation functions enables the author of a FUSE BY statement to use many different and powerful ways to resolve conflicts.

Table 2 shows a list of useful conflict resolution functions starting with the standard aggregation functions followed by more complex functions. The column

containing all conflicting values is passed as a first parameter to all functions. Depending on the function, additional parameters may be used, e.g., the source in function Choose. A Fuse By query using some of these functions to fuse three movie database tables is presented later in Sec. 4.

Conflict Resolution Strategies. There are several simple strategies to resolve conflicts that are repeatedly mentioned in the literature ([10, 16, 15]). With Fuse By all these strategies can be applied in an easy and consistent way.

Preferring one source over others. The Fuse By statement explicitly orders the tuples by sources as given in the `FUSE FROM` clause. Therefore, this strategy can be applied by writing the preferred source first and using First as conflict resolution function. Coalesce is used to fall back on values of other sources in case the desired source does not provide a value for the attribute. Choose may also be used.

Choosing the most common value. The intuition behind this strategy is that correct values prevail over incorrect ones, given enough evidence. It is implemented by applying the Vote function on a column.

Choosing the most recent value. This requires time information about the recentness present in the tables as a separate attribute or by other means. This strategy can then be applied by either ordering on this attribute and using First/Coalesce or using a special function additionally using the time information.

Take all, let the user decide. Using Group applies this strategy.

4 Implementation

We are implementing the Fuse By operator as part of an integrated information system. We base our implementation on the XXL framework — a Java library for building database systems [1]. The library builds on the cursor concept to implement relational database operators. We used the library to implement additional cursors for the outer union operator, the removal of subsumed tuples, and the Fuse By operator, and to implement a selection of conflict resolution functions. They are used in our experiments, which are currently all performed in main memory.

Computing Fusion Queries. The implementation of the Fuse By cursor follows the definition of its semantics as described in Sec. 3.2. The implementation of outer union simply concatenates all the input tuples adding null values if necessary. Our first naive implementation of the removal of subsumed tuples simply compares every tuple to all other tuples in the same group and tests for subsumption.

Experiments. We conducted several experiments with three data sources of the movie domain, kindly provided to us by the respective organizations: the Internet Movie Database[1] (I), a non-public movie collection (C) and a movie

[1] http://www.imdb.com

Function	Description
Count	Counts the number of distinct non-null values, i.e., the number of conflicting values. Only indicates conflicts, the actual data values are lost.
Min / Max	Returns the minimal/maximal input value with its obvious meaning for numerical data. Lexicographical (or other) order is needed for non numerical data.
Sum / Avg / Median	Computes sum, average and median of all present non-null data values. Only applicable to numerical data.
Variance / Stddev	Returns variance and standard deviation of data values. Only applicable to numerical data.
Random	Randomly chooses one data value among all non-null data values.
Choose	Returns the value supplied by a 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 can be broken by a variety of strategies, e.g., choosing randomly.
Group	Returns a set of all conflicting values. Leaves resolution to the user.
Shortest / Longest	Chooses the value of minimum/maximum length according to a length measure.
(Annotated) Concat	Returns the concatenated values. May include annotations, such as source of value.
Highest Quality	Evaluates to the value of highest information quality, requiring an underlying quality model.
Most Recent	Takes the most recent value. Most recentness is evaluated with the help of another attribute or other data about recentness of tuples/values.
Most Active	Returns the most often accessed or used value. Usage statistics of the DBMS can be used in evaluating this function.
Choose Corresponding	Chooses the value that belongs to the value chosen for another column.
Most complete	Returns the value from the source that contains the fewest null values in the attribute in question.
Most distinguishing	Returns the value that is the most distinguishing among all present values in that column.
Highest information value	According to an information measure this function returns the value with the highest information value.
Most general / specific concept	Using a taxonomy or ontology this function returns the most general or specific value.

Table 2. Conflict resolution functions

collection frequently used in the collaborative filtering community, Movielens[2] (M). We extracted nine different attributes out of all the movie data present in these sources and built an artificial ID. The three sources have significant intensional and small extensional overlap.

Figure 8 shows an example query from this movie domain. It illustrates the application of conflict resolution functions from Table 2. In this query, movie data is fused from the three sources (I, M, and C). Equal movies are identified by the attribute ID and conflicts in all other attributes are resolved as follows: The value for the attribute DIRECTOR is chosen from source I, assuming source I to contain the correct answer. Information about the production company (PROD_COMP) is taken from the source that contains the most information on production companies. Taking the value for the production country (PROD_COUNTRY) from the same source assumes that if a source knows a lot about production companies it also knows a lot about production countries, as these are two related aspects of making a movie. The same applies to RELEASE and DISTRIBUTOR. Worth mentioning is also the conflict resolution for the attribute GENRE. Given a taxonomy of different genre descriptions and given conflicting values, MOSTSPECIFIC returns the most specific of them in the taxonomy.

Conflict resolution for the remaining attributes is straight-forward.

```
SELECT ID,
       RESOLVE (TITLE, Longest),
       RESOLVE (YEAR, Vote),
       RESOLVE (DIRECTOR, Choose(I)),
       RESOLVE (PROD_COMP, MostComplete),
       RESOLVE (PROD_COUNTYR, ChooseCorresponding(PROD_COMP)),
       RESOLVE (GENRE, MostSpecific),
       RESOLVE (RELEASE, Earliest),
       RESOLVE (COLOR, Vote),
       RESOLVE (DISTRIBUTOR, ChooseCorresponding(RELEASE))
FUSE FROM I,M,C FUSE BY (ID)
```

Fig. 8. Complex FUSE BY example query, fusing data from three different movie data sources (I, M and C). Data conflicts are resolved, showing the use of some of the functions from Tab. 2.

Findings/Insights. The FUSE BY operator scales well. In the movie domain it is able to handle simple queries over at least 330,000 tuples using XXL and our implementation. Dominating the runtime is the sort operation. As the extensional overlap in our test tables is not very high (1-3% of the total number of tuples from the sources), the groups consist only of a few (approximately 1-10) tuples (also accounting for fuzzy duplicates in single sources). Therefore the nearly quadratic runtime of the removal of subsumed tuples hardly affects the total runtime.

[2] http://www.movielens.org

5 Conclusions

Simple, declarative, and almost automatic data integration is a pressing problem of today's large-scale information systems. This paper deals with the *data fusion* step in the data integration process. In this step, several representations of same real world objects, that may be scattered among several data sources, are fused to a single representation. During this process completeness and conciseness of the integration result are increased, while possible uncertainties and contradictions in the data are resolved.

As no relational technique so far produces such a complete yet concise result, we next propose the FUSE BY extension of SQL, which allows to declaratively specify how to fuse relational tables and thereby resolve data conflicts. Formal syntax and semantics of this new SQL clause are given. A main feature of the operator is the use of conflict resolution functions in the `SELECT` clause. We give examples and describe how they relate to aggregation functions known from conventional DBMS. Also, FUSE BY has convenient default behavior, such as the elimination of subsumed tuples, allowing sophisticated data fusion already with very simple statements.

The new operator is successfully implemented as part of our research integration system. We are currently enhancing our data fusion DBMS in terms of (i) scalability and optimization techniques, (ii) addition of conflict resolution functions, and (iii) integration with a domain-independent duplicate detection technique. Together with the optimizer already present in the XXL framework, we will be able to support the full life cycle of a query: writing the query, optimizing the query and finally executing it. As more and more efficient functionality is present, interesting optimization issues abound, particularly concerning the execution of conflict resolution functions.

In summary, writing SQL queries using the FUSE BY statement is as simple as writing conventional grouping and aggregation queries, but has the added value of a complete and concise result without contradictory data.

Acknowledgment. This research was supported by the German Research Society (DFG grant no. NA 432).

References

1. J. v. 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 VLDB*, pages 39–48, 2001.
2. U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proc. of VLDB*, pages 342–353, 1983.
3. H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An extensible data cleaning tool. In *Proc. of SIGMOD*, page 590, 2000.
4. C. Galindo-Legaria. Outerjoins as disjunctions. In *Proc. of SIGMOD*, pages 348–358, 1994.
5. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.

6. S. Greco, L. Pontieri, and E. Zumpano. Integrating and managing conflicting data. In *Revised Papers from the 4th Int. Andrei Ershov Memorial Conf. on Perspectives of System Informatics*, pages 349–362, 2001.
7. A. Motro. Completeness information and its application to query processing. In *Proc. of VLDB*, pages 170–178, Kyoto, Aug. 1986.
8. A. Motro and P. Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Information Fusion*, 2004. In Press.
9. F. Naumann, J.-C. Freytag, and U. Leser. Completeness of integrated information sources. *Information Systems*, 29(7):583–615, 2004.
10. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. of VLDB*, pages 413–424, 1996.
11. V. Raman and J. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *Proc. of VLDB*, pages 381–390, 2001.
12. J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *Proc. of SIGMOD*, pages 671–682. ACM Press, 2004.
13. K. Sattler, S. Conrad, and G. Saake. Adding Conflict Resolution Features to a Query Language for Database Federations. In *Proc. 3rd Int. Workshop on Engineering Federated Information Systems, EFIS*, pages 41–52, 2000.
14. M. Scannapieco and C. Batini. Completeness in the relational model: a comprehensive framework. In *Proceedings of the International Conference on Information Quality (IQ)*, pages 333–345, Cambridge, MA, 2004.
15. E. Schallehn, K.-U. Sattler, and G. Saake. Efficient similarity-based operations for data integration. *Data Knowl. Eng.*, 48(3):361–387, 2004.
16. V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. L. Lu, A. Rajput, T. J. Rogers, R. Ross, and C. Ward. Hermes: A heterogeneous reasoning and mediator system. Technical report, University of Maryland, 1995.
17. H. Wang and C. Zaniolo. Using SQL to build new aggregates and extenders for object- relational systems. In *Proc of VLDB*, pages 166–175, 2000.
18. L. L. Yan and M. Özsu. Conflict tolerant queries in AURORA. In *Proc. of CoopIS*, page 279, 1999.