Extensible Grouping and Aggregation for Data Reconciliation *

Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake

Department of Computer Science
University of Magdeburg
P.O.Box 4120, 39016 Magdeburg, Germany
{eike|kus|saake}@iti.cs.uni-magdeburg.de

Abstract. New applications from the areas of analytical data processing and data integration require powerful features to condense and reconcile available data. Object-relational and other data management systems available today provide only limited concepts to deal with these requirements. The general concept of grouping and aggregation appears to be a fitting paradigm for a number of the mentioned issues, but in its common form of equality based groups and restricted aggregate functions a number of problems remain unsolved. Various extensions to this concept have been introduced over the last years, especially regarding user-defined functions for aggregation and derivation of grouping properties. We propose generic interfaces for user-defined grouping and aggregation as part of a SQL extension, allowing for more complex functions, for instance integration of data mining algorithms. Furthermore, we discuss high-level language primitives for common applications and illustrate the approach by introducing new concepts for similarity-based duplicate detection and elimination.

1 Introduction

Over the last years a number of new applications with the common characteristic of condensation and integration of large data sets have gained focus in research and practice. This includes the integration of information systems, mainly driven by growing numbers of sources of related information in a global scope like the WWW or in more local scenarios like various departments of a company. More than just making this information available via a uniform interface, inconsistencies and redundancy on the data level have to be removed, and very often only condensed views of the data are required. Similar requirements exist for preparing data for data warehouses, and, later on, analytical processing steps like data mining and online analytical processing. Other examples include information dissemination, i.e. preparing data from various subscribed information channels, and the problem of homogenization of data in communicating mobile systems.

A common problem in the mentioned application scenarios is the elimination of duplicate data objects, very likely having conflicting identifiers and attribute values. Such duplicates may exist redundantly in various systems, due to errors during input or

* This research was partially supported by the BMBF (08SFB031)

for other reasons. For instance, when a user is looking for publications from integrated digital libraries, he may want to have a single representation of one article and a list of possible sources for it as part of the integrated view. Though duplicate elimination is used as an example throughout this paper, the proposed approach is usable for a variety of other issues, including for instance preparation of data for data warehouses, the generation of histograms and new opportunities for analytical data processing.

To integrate these features with current database technology we propose a flexible approach based on generalized concepts for grouping and aggregation. While, in its current form, this paradigm is limited to equality based grouping and restricted aggregate functions, it can be a powerful operation if extended to support more complex intra-group relationships and advanced aggregate functions.

The great number of possible applications and the possibly very complex grouping and aggregation functions raise the question, on what level the extensions should be implemented. While we present a solution based on user-defined functions, we also discuss possible language extensions and implementation alternatives. We do not intend to finally answer the question, what the better approach would be, but instead describe the trade off of criteria that motivated our current implementation. The extensions presented in this paper are implemented in our multidatabase query language FRAQL. This language provides powerful query operations addressing problems of integration and transformation of heterogeneous data [20] and therefore, it is a suitable platform for building up a framework for data preparation and integration.

In section 2 we give an overview of relevant related work from the fields of extended query processing as well as the field of similarity-based duplicate detection and elimination. In section 3 we give a more detailed description of existing problems and the motivation for the used approach. As a basis of the current implementation user-defined aggregation and grouping functions as supported by FRAQL are introduced in section 4. Section 6 describes application scenarios and a discussion about possible implementation alternatives is given in 7. The paper ends with a conclusion and an outlook in section 8.

2 Related Work

The approach described in this paper is intended to be used in data integration scenarios. Related topics are from this field, especially concerning the running example of entity identification, as well as advanced concepts for grouping and aggregation that are relevant in research fields like analytical data processing.

Throughout the paper we illustrate our approach by focusing on the problem of entity identification and duplicate elimination. This problem was discussed extensively in various research areas like database and information system integration [24, 17], data cleaning [1, 7], information dissemination [23], and others. Early approaches were merely based on the equality of attribute values or derived values. Newer research results deal with advanced requirements of real-life systems, where identification very often is only possible based on similarity. Those approaches include special algorithms [18, **?**], the application of methods known from the area of data mining and even ma-

chine learning [16]. Other interesting results came from specific application areas, like for instance digital libraries [8, 14].

An overview of problems related to entity identification is given in [15]. In [17] Lim et. al. describe an equality based approach, include an overview of other approaches and list requirements for the entity identification process. Monge and Elkan describe an efficient algorithm that identifies similar tuples based on a distance measure and builds transitive clusters in [19]. In [7] Galhardas et. al. propose a framework for data cleaning as a SQL extension and macro-operators to support among other data cleaning issues duplicate elimination by similarity-based clustering. The similarity relationship is expressed by language constructs, and furthermore, clustering strategies to deal with transitivity conflicts are proposed. In [12] Hernández et. al. propose the sliding window approach for similarity-based duplicate identification where a neighbourhood conserving key can be derived and describe efficient implementations.

Closely related to similarity based entity identification is the integration of probabilistic concepts in data management [4, 6]. Especially, for data integration issues and the aforementioned problems probabilistic approaches were verified and yielded useful results [21, 13]. The WHIRL system and language [3] by Cohen uses text-based similarity and logic-based data access as known from Datalog to integrate data from heterogeneous sources.

The importance of extended concepts for grouping and aggregation in information integration is emphasized by Hellerstein et. al. in [11]. In particular, user-defined aggregation (UDA) were proposed in SQL3 and are now supported by several commercial database systems, e.g. Oracle8i, IBM DB2, Informix. In [22] the SQL-AG system for specifying UDA is presented, that translates to C code. A more recent version of this approach called AXL is described in [22] and its usage in data mining is discussed.

Several extensions to the classic **`group by`**-operator of SQL were proposed. Probably the most important extension is the data cube operator presented in [10], which is now support in some commercial systems. In [2] an additional **`such that`**-clause for the **`group by`**-operator is proposed introducing variables that range over a group and can be qualified by the **`such that`**-clause. Red Brick's RISQL (now Informix) allows functions in the **`group by`**-clause and supports several predefined aggregation functions, e.g. `Rank`, `N_tile`, as well as cumulative aggregates. Some other OLAP vendors provide similar concepts.

3 Motivation

In this section we informally introduce our approach for extended grouping and aggregation and describe problems with the common standard and proposals for extensions made over the last years. To illustrate the motivation behind our concepts we focus on the problem of duplicate elimination, that is common in many of the applications mentioned above. Anyway, the approach is not limited to this specific problem. Other applications are introduced in section 6.

While equality based duplicate elimination is a standard feature of object-relational DBMS, for the scenarios mentioned in section 1 more sophisticated solutions are required to deal with possible inconsistencies and different representation conventions

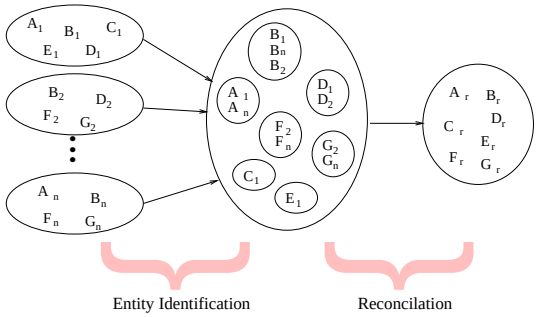


Fig. 1. Similarity based duplicate elimination

that are typical in heterogeneous environments. Furthermore, we assume that other conflicts, for instance regarding data models and structures, are resolved beforehand. The proposed approach was implemented as part of the FRAQL system that provides features to deal with these conflicts and is described in more detail in [20].

Similarity-based duplicate elimination can be considered a two-step process as illustrated in figure 1 consisting of entity identification and reconciliation. During *entity identification* groups of objects potentially describing the same real-world object are created. We have to keep in mind that whatever similarity criterion and strategy we choose, this step can only derive hypotheses about the relationship and will remain error-prone. The number of over-identified (unrelated entities in one group) and under-identified (related entities in separate groups) tuples derived from sample data and evaluated by a user with domain knowledge can be used as a quality measure to tune this step during the design phase.

The *entity reconciliation* step uses the groups found during entity identification as an input to derive one integrated representation for the real-world object represented by this group. This can be done by merging data, e.g. sum up the sales numbers of products from various business areas, or by using additional knowledge about the integrated data, like for instance data quality. So, user-defined aggregation functions appear as an appropriate concept for reconciliation tasks.

Both steps are highly application-dependent, i.e. to support similarity based duplicate elimination a system has to provide concepts to describe the characteristics of both steps. We will later on discuss on which levels this requirement can be addressed. Current database technology and languages do not offer sufficient solutions to support such operations. However, the general concept of grouping and aggregation can be used as a powerful framework to handle these and other integration issues. In the following we will describe shortcomings of the existing operations and required extensions.

Currently the **`group by`**-operator as standardized is equality based and works one tuple at a time, i.e. the identifier of the group the tuple belongs to is derived considering only values of one tuple and no implicit or explicit relationships between tuples. This is also true for current extensions allowing user-defined functions as a **`group by`** clause to derive values that are not in the domain of any of the relations attributes. New applications introduced in section 1 require a more flexible, more general approach. As

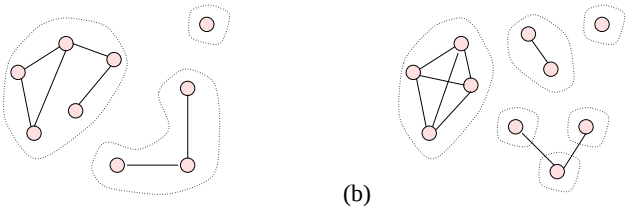


Fig. 2. Grouping by (a) transitive and (b) strict similarity

an example consider the following query that performs similarity-based duplicate elimination for bibliographic records from three sources by describing a pairwise similarity criterion and a strategy to build groups.

```
select pickBySource(title,source),
       fullName(author)
from DBLP union SPRINGER union NCSTRL
group by transitive similarity
on sameText(title) and
   sameName(author) or
   isbn
threshold 0.95
```

The similarity criterion is specified in the **on**-clause by a probabilistic logic expression using system-defined (`sameText`) and user-defined (`sameName`) functions taking advantage of domain knowledge. System-defined methods can for instance be used for common data types without taking advantage of knowledge about the application-dependent semantics of the given attribute. As an example we consider string attributes, where either vector representations and according distance measures for longer text fields like the title or the edit distance to deal with typos etc. in shorter string representations can be applied. The user-defined `sameName`-function in this case can exploit domain knowledge, like the fact that first names are often abbreviated or names can be written "Lastname, Firstname". These functions can be implemented as two-parameter functions for comparing values from two tuples and return float values between 0 and 1 derived from the distance measure. The usage of an attribute, like `isbn` in the example, compares two values for equality and returns either 0 or 1. The expression can be evaluated applying the MIN/MAX combination rule, i.e. for two tuples t_1 and t_2 a similarity value can be derived as follows:

$$
\begin{aligned}
sim_a(t_1, t_2) &:= t_1.a = t_2.a \\
sim_{f(a)}(t_1, t_2) &:= f(t_1.a, t_2.a) \\
sim_{A \wedge B}(t_1, t_2) &:= MIN(sim_A(t_1, t_2), sim_B(t_1, t_2)) \\
sim_{A \vee B}(t_1, t_2) &:= MAX(sim_A(t_1, t_2), sim_B(t_1, t_2)) \\
sim_{\neg A}(t_1, t_2) &:= 1 - sim_A(t_1, t_2)
\end{aligned}
$$

Two tuples are pairwise similar if the evaluated logic expression is above the specified threshold. The similarity relationship is intransitive, hence, a further strategy to establish an equivalence relation is required to build groups.

Two simple strategies are introduced here and illustrated in figure 2, their usefulness depending on a given application scenario. The **`transitive`** closure strategy used in the example above builds groups by simply considering the transitive closure of a tuple as its group. This is a very loose strategy that may result in big groups with potentially very different tuples. A more conservative strategy would be the **`strict similarity`**, that demands pairwise similarity between all tuples within a group and splits the group in case of a conflict.

There is an unlimited number of possible strategies that might become useful in specific applications, especially when other similarity relationships are considered or approaches like clustering or classification are used. Anyway, they all share the common characteristic that the result of the grouping process may depend on the whole input relation, i.e. represents a holistic function.

Though the query example shown above describes a declarative way to express the duplicate elimination, the current implementation is based on the more general concept of user-defined grouping functions and aggregation as introduced in section 4. A discussion of advantages and possible problems regarding language constructs versus user-defined grouping functions is given in section 6.

4 User-defined Grouping and Aggregation

Obviously, using grouping for duplicate identification and aggregation for reconciliation depends heavily on the problem domain. Additionally, only in rare cases simple built-in aggregation functions are sufficient for reconciliation purposes. Therefore, it is necessary to support application-specific grouping and aggregation functions, i.e., user-defined functions. Whereas user-defined aggregation (UDA) is considered in the current SQL standard documents and already supported by commercial database systems like Informix, Oracle8i or IBM DB2, to our best knowledge user-defined grouping (UDG) was not addressed until now. SQL allows only simple grouping by attributes and only some proposed OLAP extensions to SQL enable at least the usage of predefined functions as grouping parameter.

Our query language FRAQL supports both concepts: UDA and UDG. A UDA is implemented in FRAQL as an external class written in C++ or Java. The interface of this class consists of the following methods:

```
public interface UDA {
  void init ();
  boolean iterate (Object[] args);
  Object result ();
}
```

At the beginning of processing a relation, the method `init` is called. For each tuple the `iterate` method is invoked. The final result is obtained via the method `result`.

Because a UDA class is instantiated once for the whole relation the "state" of the aggregate can be stored. Therefore, UDA functions can be used for reconciliation, i.e., deriving a representative value from a group of values representing the same real-world concept.

Regarding user-defined grouping we have to distinguish two cases. If the assignment of a tuple to a group is based on equality of attribute values, only one tuple at a time has to be considered, because the group membership can be computed only from the attribute values. In contrast, if we want to assign a tuple to a group based on attribute similarity, it has to be compared to all current members of a group (or at least to one representative) and to all groups. Depending on this comparison we can decide on the group membership.

This difference in grouping is addressed by two modes: context-free and context-aware grouping. *Context-free* grouping is the usual approach as known from SQL. FRAQL extends this by enabling arbitrary expressions as grouping parameter. So, user-defined grouping can be implemented as an expression including the invocation of external functions, which may be defined as follows:

```
create function regionCode
    (float, float) returns integer
   external name 'RegionCode'
   language Java
```

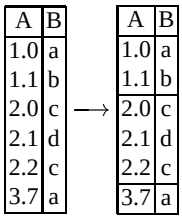
An example of using a UDF for grouping is the following query that groups tuples representing weather measurements by regions. The region is computed from the geographical position with the help of the UDF `regionCode`:

```
select avg(temperature), rc
from Weather
group by regionCode (longitude,
         latitude) as rc
```

Similarity-based grouping is supported in FRAQL as *context-aware* grouping. Here the problem is, that the group membership of a tuple can be determined not until all tuples of the relation are processed. Furthermore, as discussed in section 3 groups are not constant during processing a relation, because groups could be split or merged due to similarity relationships of new tuples. So, UDG functions are implemented as classes with the following interface:

```
public interface UDG {
  void init (Object[] args);
  boolean iterate (long tid,
                   Object[] values);
  void finish ();
  void groupOpen ();
  long groupNext ();
  void tupleOpen (long gid);
  long tupleNext (long gid);
}
```

The meaning of these methods is as follows, whereas the processing is performed in two steps. Starting in the first step with a new input relation the `init` method is called for initialization purposes. Then, each tuple is processed by invoking the `iterate` method and finally the `finish` method is called. Before `finish` the group partitioning can change, but after `finish` was called, the number of groups as well as the assignment of tuples are fixed. In the second step, projection and aggregation are applied to the individual groups. For this purpose, a UDG provides iterator-like methods for navigating over the groups and contained tuples. The following example illustrates the principle of a UDG function. The grouping function used in this example builds groups of tuples



A	B
1.0	a
1.1	b
2.0	c
2.1	d
2.2	c
3.7	a

A	B
1.0	a
1.1	b
2.0	c
2.1	d
2.2	c
3.7	a

Fig. 3. Grouping example for **maximumDifference**

with no gaps in the float values of column A greater than 0.5. The according implementation is a special case of computing a transitive closure similar to the algorithm given later on in section 6. Because of the simple 'similarity measure' there are more efficient implementations for this special case. Anyway, it is obvious that the UDG needs not to store the whole tuple, but only a tuple id, which can be used for retrieving the actual tuple during the second step.

The special treatment of context-aware grouping is expressed by the additional keyword **`context`** in the **`group by`** clause:

```
select avg(A),min(B)
from FloatMap
group by context
   maximumDifference(A,diff = 0.5)
```

Context-aware grouping provided by FRAQL is not a direct implementation of similarity grouping described in the motivating example in section 3. However, it forms a generic framework for implementing this kind of grouping as well as other approaches like clustering, which we will describe later.

5 Semantics and Implementation

In this section we sketch the semantics of our proposed operations. We extend the standard relational algebra with a generalized grouping operator $\gamma_{\phi,\psi}$ where ϕ is a grouping function and ψ is a reconciliation function. Figure 4 illustrates the application of this

operation. An input relation consisting of two columns A and B has to be grouped by similar values of A. This results in two groups g_1 and g_2. For each of these groups the reconciliation function $\psi_{\text{avg}(A),\text{min}(B)}$ derives a single tuple ($\texttt{avg}(A)$, $\texttt{min}(B)$). First of

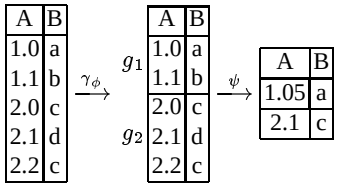


Fig. 4. Application of γ and ψ

all, for the grouping operator a "same-group" function ϕ is required. Assuming $r(R)$ as a relation of schema R, we can define the signature of ϕ as follows:

$$\phi : r(R) \times r(R) \rightarrow \textbf{bool}$$

where it holds $\phi(t_1, t_2) = \phi(t_2, t_1)$. This function is applied on two tuples at a time and returns **true** if the tuples are similar, i.e. belonging to the same group. If we consider holistic functions the test for membership in the same group has to be performed in the context of the whole relation. Therefore, an extended version $\overline{\phi}$ is necessary, where the function depends on the relation as a whole:

$$\overline{\phi_{r(R)}} : r(R) \times r(R) \rightarrow \textbf{bool}$$

As an example for this kind of functions consider the computation of the transitive closure as part of a binary similarity operator.

Based on a group function we can next define the grouping operator γ_ϕ as follows:

$$\gamma_\phi : r(R) \rightarrow 2^{r(R)}$$

where

$$\gamma_\phi(t_1) = \gamma_\phi(t_2) \text{ iff } \phi(t_1, t_2) = \textbf{true}$$

A reconciliation function ψ has the signature

$$\psi : 2^{r(R)} \rightarrow r(R)$$

and consists of a list of aggregate functions either built-in like `avg`, `min`, `max` etc. or user-defined as introduced in section 4. The resulting tuples have the same tuple type as the input relation. So, this function denotes a reconciliation grouping. For the general case, ψ has the following signature:

$$\psi : 2^{r(R)} \rightarrow r(R')$$

Using both same-group function ϕ and reconciliation function ψ we can finally define the grouping operation:

$$\gamma_{\phi,\psi}(r) = \psi(\gamma_\phi(r))$$

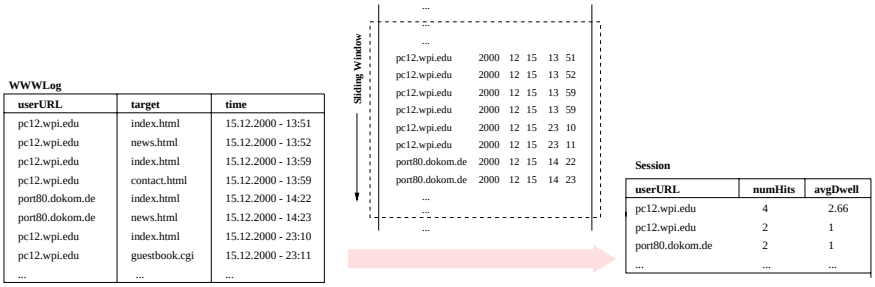


Fig. 5. Using the sliding window approach to analyse WWW logs

6 Applications

The concepts introduced in the last sections allow for arbitrary grouping and aggregation functions. In this section we discuss some applications and related implementation issues. Let us first consider the example of similarity-based duplicate elimination as introduced in section 3, where groups are build using the transitive closure strategy. A general implementation framework for a UDG applying the transitive closure strategy with quadratic time complexity can be sketched as follows:

```
UDG: transitiveSimilarity

init():
    initialize group_table as hash table

iterate(tuple_id, values):
    for all groups in group_tables
        for all tuples in group
            if tuples are similar
                if already found another group
                    merge groups
                    add tuple
                else add tuple
        endfor
    endfor
    if no group found
        create new group in group_table
        add tuple to new group

finish():
    finished = true
```

For each tuple the `iterate`-method of the UDG is called and checks for similar tuples in all existing groups using a nested loop and builds groups according to the strategy. This way a function representing the similarity criterion given in section 3 can be used in a query as follows:

```
select pickBySource(title,source),
       fullName(author)
from DBLP union SRINGER union NCSTRL
group by context
   transitiveSimilarity(isbn,title,
                        author,0.95)
```

The time complexity for the nested loop can be impractical in scenarios where large input sets are the common case, like for instance when removing duplicates during the data cleaning step for data warehousing.

Without describing the details of the similarity criterion and its implementation for this case, it is obvious that there is potential for optimization. Using index structures supporting similarity and application domain knowledge the common complexity for duplicate elimination of $O(n \log n)$ can be achieved in certain scenarios.

In the following simple example a web server access log is queried to derive information about "user sessions", i.e. the period of time a user spends to continuously browse pages on this server. If the time difference between two hits exceeds a certain limit, the session has ended.

```
select count(*), avgDwell(time)
from WWWLog
group by context
  sameSession (userURL,
               time,maxDiff=30)
```

The `sameSession` grouping function can use the neighbourhood conserving approach described in [12] and illustrated in figure 5 by using the guest URL as well as extracted year, month, day etc. as a key. This way an order is established and the sliding window approach can be applied to check if the difference of time between hits exceeds the limit of 30 minutes. For the sliding window approach only tuples within a window of a size greater than the number of the maximum expected tuples within a group are compared to each other. This is possible with constant time complexity. The moving of the window has linear complexity, so, including the sorting the overall grouping process has a complexity of $O(n \log n)$. This approach, though quite efficient, can only be used if a meaningful key for a neighbourhood conserving order can be derived.

Using user-defined grouping and aggregation functions is a simple way to integrate clustering algorithms and make use of them in database environments. The following example uses DBSCAN [5] to build density based clusters of locations.

```
select biggestCity(name,occupation),
       avg(precipitation)
from meteoDat natural join locations
group by context
   DBSCAN(longitude,latitude,
          minNeigh = 2, eps = 100)
```

The query returns a representative city for each cluster and the average precipitation in the area.

The extended grouping functionality described here also offers new options to condense data for online analytical processing. The overall concept fits well with new extensions made to the SQL standard to support the generation of multidimensional views. Using clustering or, more generally speaking, similarity-based grouping at this level helps to make implicit dependencies and relationships obvious more easily.

7 Alternatives and Implications

Considering the motivating query from section 3 containing a **`group by similarity`** clause and the proposed **`group by context`** extension the question arises, which is the better approach ?

A dedicated solution for similarity-based grouping opens the possibility to apply special optimization strategies. For example, the grouping operator could benefit from special-purpose index structures on grouping attributes, e.g., inverted lists. Furthermore, the development of predicates used in the grouping clause is simplified, because only the similarity criterion for two attributes values has to be implemented. However, this approach is limited to the domain of similarity. Other grouping techniques like clustering of non-textual data would require other extensions.

A more generalized solution like the proposed approach of context-aware grouping opens a broader range of applications. The disadvantage is the higher complexity of developing grouping functions. All the tasks of group membership checking as well as merging and splitting groups are burden to the implementer. Anyway, we argue for this solution because of several reasons:

- Text-similarity grouping can be implemented on top of the context-aware grouping, but not vice versa.
- Though grouping as well as grouping functions are often application-dependent, common functions could be implemented and packaged as a database cartridge or extender, as already available in modern database systems.
- Using SQL/PSM or at least a kind of embedded SQL simplifies the development of aggregation and grouping functions. In addition, this makes the implementation of these functions transparent to the query processor and allows the inclusion in the query optimization process.

Particularly the latter issue is subject of our future work.

8 Conclusions

In this paper we have presented two extensions to a SQL-like query language addressing advanced requirements regarding data reconcilation in integration scenarios. The `group-by-context` clause provides a mechanism for applying user-defined functions for grouping purposes. The merging or reconciliation of the tuples of the identified groups is performed via aggregation functions. Both features together form a powerful framework for data reconciliation as part of extended SQL queries which can be applied in various application scenarios. Additionally, it improves the extendible of database

systems and could be utilized in database extenders or cartridges. The presented extensions are implemented as part of our federated query engine for the FRAQL language. In the context of a multidatabase language concepts for advanced data reconciliation are particularly useful.

In the future, we plan to use SQL for implementing grouping and aggregation functions in order to support a more declarative way for specifying these functions and to establish a basis for optimizing grouping queries together with queries as part of the functions. A second important task is to utilize the optimization potential during the context-aware grouping, i.e. the mentioned sliding window approach, dedicated index structures for text-based similarity or caching and parallelization. For this purpose the properties and requirements of UDA and UDG functions have to be specified and taken into account during query optimization and evaluation.

References

1. D. Calvanese, G. de Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. A principled approach to data integration and reconciliation in data warehousing. In *Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'99), Heidelberg, Germany*, 1999.
2. D. Chatziantoniou and K.A. Ross. Querying multiple features of groups in relational databases. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors, *Proc. of 22th Int. Conf. on Very Large Data Bases (VLDB'96), Mumbai (Bombay), India*, pages 295–306. Morgan Kaufmann, 1996.
3. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 201–212. ACM Press, 1998.
4. D. Dey and S. Sarkar. A probabilistic relational model and algebra. *ACM Transactions on Database Systems*, 21(3):339–369, September 1996.
5. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In E. Simoudis, J. Han, and U.M. Fayyad, editors, *Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231. AAAI Pres, 1996.
6. N. Fuhr. Probabilistic datalog – A logic for powerful retrieval methods. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Retrieval Logic, pages 282–290, 1995.
7. H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: an extensible data cleaning tool. In Weidong Chen, Jeffery Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, volume 29(2), pages 590–590, 2000.
8. C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *DL'98: Proceedings of the 3rd ACM International Conference on Digital Libraries*, pages 89–98, 1998.
9. G. Graefe. Query Evaluation Techniques For Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
10. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In S.Y.W. Su, editor, *Proceedings of the 12th Int. Conf. on Data Engineering (ICDE'96), New Orleans, Louisiana*, pages 152–159. IEEE Computer Society, 1996.

11. J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, Open Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1):43–49, 1999.
12. M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 127–138, San Jose, California, 22–25 May 1995.
13. S. B. Huffman and D. Steier. Heuristic joins to integrate structured heterogeneous data. In *AAAI Spring Symposium on Information Gathering*, 1995.
14. J. A. Hylton. Identifying and merging related bibliographic records. Technical Report MIT/LCS/TR-678, Massachusetts Institute of Technology, February 1996.
15. W. Kent. The breakdown of the information model in multi-database systems. *SIGMOD Record*, 20(4):10–15, December 1991.
16. Wen-Syan Li. Knowledge gathering and matching in heterogeneous databases. In *AAAI Spring Symposium on Information Gathering*, 1995.
17. E.-P. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity identification in database integration. In *International Conference on Data Engineering*, pages 294–301, Los Alamitos, Ca., USA, April 1993. IEEE Computer Society Press.
18. A. E. Monge and C. P. Elkan. The field matching problem: Algorithms and applications. In Evangelos Simoudis, Jia Wei Han, and Usama Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, page 267. AAAI Press, 1996.
19. A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'97)*, 1997.
20. K.-U. Sattler, S. Conrad, and G. Saake. Adding Conflict Resolution Features to a Query Language for Database Federations. *Australian Journal of Information Systems*, 8(1):116–125, 2000.
21. F. Tseng, A. Chen, and W. Yang. A probabilistic approach to query processing in heterogeneous database systems. In *Proceedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 176–183, 1992.
22. H. Wang and C. Zaniolo. Using sql to build new aggregates and extenders for object- relational systems. In A. El Abbadi, M.L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proc. of 26th Int. Conf. on Very Large Data Bases (VLDB'00), Cairo, Egypt*, pages 166–175. Morgan Kaufmann, 2000.
23. T. W. Yan and H. Garcia-Molina. Duplicate removal in information dissemination. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB '95)*, pages 66–77, San Francisco, Ca., USA, September 1995. Morgan Kaufmann Publishers, Inc.
24. G. Zhou, R. Hull, R. King, and J. Franchitti. Using object matching and materialization to integrate heterogeneous databases. In *Proc. of 3rd Intl. Conf. on Cooperative Information Systems (CoopIS-95), Vienna, Austria*, 1995.