Advanced Grouping and Aggregation for Data Integration *

Eike Schallehn Kai-Uwe Sattler Gunter Saake
Department of Computer Science, University of Magdeburg
P.O. Box 4120, D-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. As outlined in [1], the general concept of grouping and aggregation appears to be a fitting paradigm for a number of these issues, but in its common form of equality based groups or with current extensions like simple user-defined functions to derive group-by values on a per tuple basis and restricted aggregate functions a number of problems remain unsolved. We describe two extensions to the grouping mechanism, a generic one to support holistic user-defined grouping functions and higher level construct that provides similarity based grouping suitable in a number of applications like duplicate detection and elimination.

1. INTRODUCTION

For 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, inconsistencies and redundancy on the data level have to be removed, and very often only condensed views of the data are required. 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.

Similarity-based duplicate elimination is a common task in data integration, and therefore used as an example application throughout this paper, though the concepts introduced here are not limited to this usage. Duplicate elimination can be considered a two-step process consisting of entity identification and reconciliation. During *entity identification* groups of objects potentially describing the same real-world object are created. 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 as well as standard 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.

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. A simple example for the their usage is:

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

However, complex relationships between tuples, that are for instance not transitive or symmetric do not fit with these concepts, and still require special algorithms for data processing, often including domain-specific knowledge. Our goal is to offer ways to separate generic from domain-specific aspects of common tasks in data integration.

2. ADVANCED GROUPING

We present two proposals, the first being a specialized language extension that offers optimization opportunities, and a generic one based on user-defined functions, both implemented as part of an extended query language for data integration. 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.

2.1 Grouping by Similarity

As an example of similarity based grouping 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
```

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

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 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 for two tuples t_1 and t_2 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.

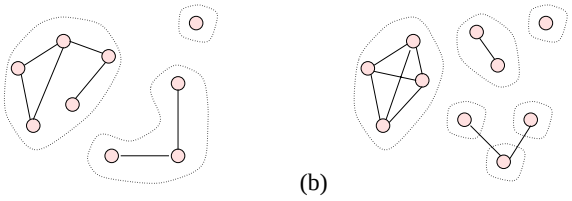

Figure 1: Grouping by (a) transitive and (b) strict similarity

Two simple strategies are introduced here and illustrated in figure 1, 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 for specific applications.

2.2 User-defined grouping

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. The concept of user-defined aggregation and its implementation in FRAQL is described in more detail in [3].

In contrast to the common equality based grouping, 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 or possible group rearrangements. Similarity-based grouping is a special case of *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, 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 their contained tuples.

The following example illustrates the principle of a UDG function.

The grouping function used in this example builds groups of tuples with no gaps in the float values of column A greater than 0.5. Basically this represents a special case of the transitive closure strategy, for which in this case efficient implementaions can be provided easily. Furthermore, 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.

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

$\longrightarrow$

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

Figure 2: Grouping example for maximumDifference

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 is not a direct implementation of similarity grouping described in the motivating example in section 2. However, it forms a generic framework for implementing this kind of grouping as well as other approaches like clustering etc. As an example reconsider the query used to motivate grouping by similarity at the beginning of this section. Using the more general concept of user-defined grouping it can be expressed as.

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

Anyway, it is obvious, that the functionality described before plus potential optimizations have to be realized in the grouping function. This requires more implementation efforts and the resulting implementation in this case is less flexible. An extended discussion of these aspects and application issues are is included in [4].

3. IMPLEMENTATION AND OPTIMIZATION

A dedicated solution for similarity-based grouping opens the possibility to apply special optimizations for avoiding the general cost of $O(n^2)$ for pairwise comparisons of n tuples. Like equality based duplicate elimination this can be optimized to $O(n \log n)$. For example, the grouping operator benefits from special-purpose index structures on grouping attributes, e.g., inverted lists, Tries as described in [5] as well as common index structures with slight modifications. An algorithm considering index structures during evaluation of complex similarity criteria applying the threshold to shortcut computations during the grouping process is not included here due to spacial limits. Another important issue not discussed here is the creation and maintenance, especially in virtual integration scenarios with autonomous source systems. Currently only indexes built on-the-fly are considered. Furthermore, the development of predicates used in the grouping clause is simplified, because only the similarity criterion for two attribute values has to be implemented. However, this approach is limited to certain similarity measures and pre-defined strategies to build groups. Other grouping techniques like clustering of non-textual data would require other extensions.

For the context-aware grouping a nested loops approach is applicable in the general case resulting in an $O(n^2)$ time complexity. However, there are special cases, where an optimization is possible, e.g if indexes can be used similar to the previous descriptions. If a linear order for one of the attributes used for grouping is defined, a sliding window approach could be used [2]. the following section. The proposed approach of context-aware grouping opens a broad 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 currently focus on this solution because similarity grouping can be implemented on top of the context-aware grouping, but not vice versa. Furthermore, common functions could be implemented and packaged as a database cartridge or extender, as available in current database systems.

4. CONCLUSIONS

In this paper we have presented two extensions to a SQL-like query language addressing the problem of data reconcilation. The **`group-by-context`** clause provides a mechanism for applying user-defined functions for grouping purposes. The **`group-by-similarity`** clause is a special case of context aware grouping, providing the possibilty to describe similarity of tuples and and grouping strategies in a more descriptive way. 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 extensibilty 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 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 similarity-based grouping, i.e. an applicable set of system-defined similarity functions and the according dedicated index structures. 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.

5. REFERENCES

[1] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, Open Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1):43–49, 1999.

[2] 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.

[3] K. Sattler and E. Schallehn. A Data Preparation Framework based on a Multidatabase Language. In M. Adiba, C. Collet, and B.P. Desai, editors, *Proc. of Int. Database Engineering and Applications Symposium (IDEAS 2001)*, pages 219–228, Grenoble, France, 2001. IEEE Computer Society.

[4] E. Schallehn, K. Sattler, and G. Saake. Extensible grouping and aggregation for data reconciliation. In *Proc. 4th Int. Workshop on Engineering Federated Information Systems, EFIS'01, Berlin, Germany*, 2001. *To appear*.

[5] H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, 1996.