Extensible and Similarity-based Grouping 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

Data integration as required in a variety of applications like data warehousing, information system integration etc. makes great demands regarding features to deal with overlapping and inconsistent 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 various of the current issues in data integration, but in its common form of equality-based grouping a number of problems remain unsolved. Various extensions to this concept have been introduced over the last years regarding user-defined functions for aggregation and grouping. Especially, existing extensions to the grouping operation like simple derivations of group-by values do not meet the requirements of data integration applications. 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. For both approaches implementation and optimization issues are considered.

1 Introduction

Over the last years a number of new applications with the common characteristics 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 these 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 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 general concept of context-aware grouping introduced here does support holistic grouping 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. We present two proposals, one being a specialized language extension that offers optimization opportunities, and a generic one based on user-defined functions. We do not intend to finally answer the question, what the better approach would be, but instead describe the trade off of criteria that may motivate certain implementations. Though we introduce one special language construct that is suitable for duplicate elimination and similar tasks, we believe that the development of other such extensions should be driven by a thorough investigation of data integration requirements.

The extensions presented in this paper are implemented as part of the 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. Anyway, the concepts introduced here can be implemented on top of existing DBMS by using system-dependent extensibility interfaces.

In section 2 we give an overview of 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 extensible aggregation and grouping operations are introduced in section 4 and the basic semantics are described in section 5. The concepts of the more specialized approach of similarity-based grouping are explained in detail in section 6. A discussion about implementation and optimization issues is given in section 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 [25, 16], data cleaning [1, 6], information dissemination [24], 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 [17, 11], the application of methods known from the area of data mining and even machine learning [15]. Other interesting results came from specific application areas, like for instance digital libraries [7, 13].

An overview of problems related to entity identification is given in [14]. In [16] 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

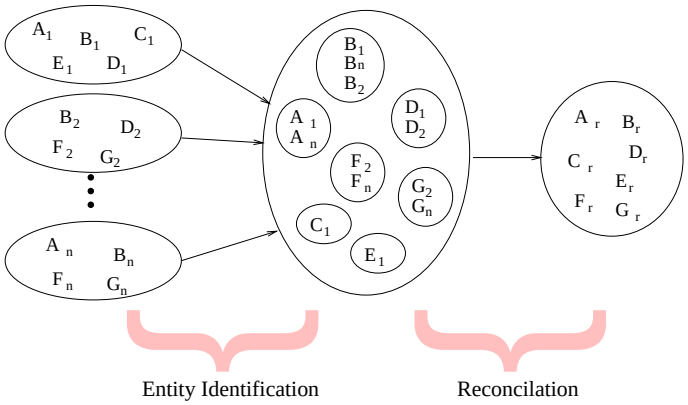


Figure 1: Similarity based duplicate elimination

based on a distance measure and builds transitive clusters in [18]. In [6] 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 [11] Hernández et. al. propose the sliding window approach for similarity-based duplicate identification where a neighborhood 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, 5]. Especially, for data integration issues and the aforementioned problems probabilistic approaches were verified and yielded useful results [22, 12]. 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 [10]. 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 [23] 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 [23] 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 [9], 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.

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 that are typical in heterogeneous environments. 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. in order 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 processes 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 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.

As an example of our approach 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 a keyword list 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. 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. In this case we simply consider the transitive closure of the similarity relation. Other strategies and more details on the basic concepts are given in section 6.

To our best knowledge, there is no existing approach that offers declarative ways to handle the mentioned conflicts, and is capable of dealing with the described kind of data inconsistencies. Regarding the growing level of accessibility to data from overlapping domains, we consider this a key issue in future information systems. The concepts presented here are extensions to relational query languages and query processing, but the basics are applicable to other ways of accessing and presenting data, also including semi-structured data.

4 Extensible 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. This general concept of extensible grouping and aggregation can be used as a basis to provide functionality for that is useful in certain application domains. 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 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. An implementation of a UDA is registered in FRAQL as follows:

```
create aggregation pickBySource
    (varchar, varchar) returns varchar
   external name 'PickBySource' language Java
```

The concept of user-defined aggregation and its implementation in FRAQL is described in more detail in [19].

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. An example for that was given in section 3.

Similarity-based grouping is supported in FRAQL based on *context-aware* grouping. Here the problem is, that the group membership of a tuple can not be determined 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 (TupleID tid, Object[] values);
  void finish ();
  void groupOpen ();
  GroupID groupNext ();
  void tupleOpen (GroupID gid);
  TupleID tupleNext (GroupID 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 (`groupNext`), which returns a group identifier (`gid`) as well as for navigating over the tuples (`tupleNext` returning a tuple identifier `tid`) of a group given by the group identifier.

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 creating a transitive closure with a simple similarity criterion, which can easily be implemented and optimized within a UDG. Furthermore, it is obvious that the UDG needs not

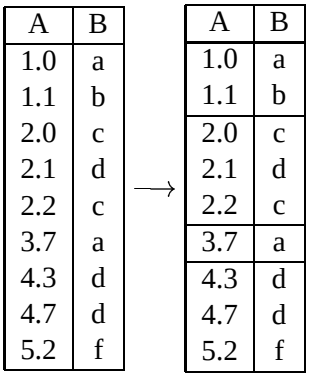


Figure 2: Grouping example for **maximumDifference**

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 introduced in section 6. Furthermore, it serves as a more generic view for explaining the semantics of the introduced concepts in section 5.

5 Semantics of extensible Grouping and Aggregation

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 3 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 like described in the example of section 4. This results in two groups g_1 and g_2. For each of these groups the reconciliation function $\psi_{\text{avg}(A),\min(B)}$ derives a single tuple (`avg`(A), `min`(B)).

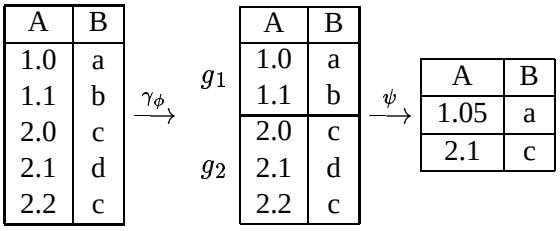


Figure 3: Application of γ and ψ

First of 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:

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

6 Concepts of similarity-based Grouping

The implementation of the commonly used, equality-based **`group by`** operation utilizes existing order on group by attributes or hash tables to derive the group membership for a tuple, i.e. find the relevant group and either add the tuple to the group, or create a new group if no relevant group was found. Once the processing of the tuple is done, its group membership is a fixed property of the tuple.

The general concept of context-aware grouping as introduced in this paper may require more complex computations according to looser inter-tuple relationships, such as similarity measures, and inter-group relationships, like the introduced simple strategies for building groups. Still, a general framework for according algorithms can be outlined as shown in Fig. 4.

An iterative processing of an input relation is considered throughout this paper, but this framework itself does not ensure that the iteration order does not have an impact on the result of the grouping

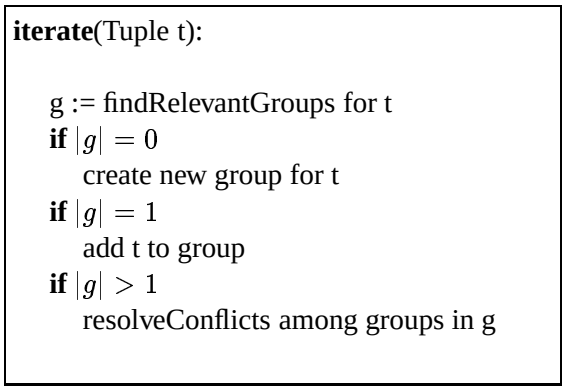


Figure 4: Framework for similarity grouping

operation. The fulfillment of this requirement for relational semantics is either burden to the implementer of the UDG, or can be discarded if the specific application allows this relaxation.

The major difference to equality-based grouping is for one part hidden in the possibly very complex determination of relevant groups. A naive implementation may for instance require pairwise comparisons with previously analyzed tuples, e.g. to derive a distance measure. On the other hand, the resolution of conflicts may involve complex modifications of the group structure like merging, splitting etc.

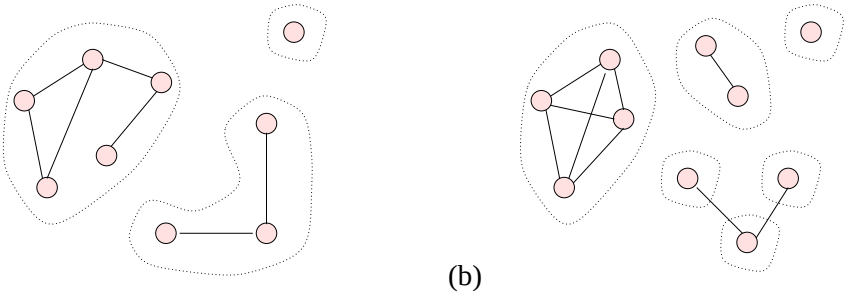
To illustrate this framework, let us consider the concept of similarity based grouping as introduced in section 3 This represents a specialization of the the more general context-aware group by operation, allowing a declarative way to group tuples by similarity and allowing for certain optimization approaches described in section 7.

```
select <aggregation_list>
from ...
 ...
group by { transitive | strict } similarity
on <similarity_expression>
threshold <probability>
```

The usage and advantages/disadvantages of user-defined and system-defined similarity measures was already introduced in section 3. For two given tuples t_1 and t_2 the similarity expression can be evaluated as follows:

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

The two tuples t_1 and t_2 are considered similar, if the returned value for the overall expression is above the given threshold. Based on this the groups can be build according to given strategy.



(a) (b)

Figure 5: Grouping by (a) transitive and (b) strict similarity

Two simple strategies are introduced here and illustrated in figure 5, 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. With respect to the framework presented above, a non-optimized implementation only considering the transitive closure strategy is outlined in Fig. 6.

```
findRelevantGroups(Tuple t):

    for all groups g
        for all tuples t_g in group g
            if evaluate(expression,t,t_g) > threshold
                add g to resultset
                break
        endfor
    endfor
    return resultset


handleConflicts(g):

    merge groups g
```

Figure 6: Naive algorithm for similarity grouping

For each tuple we find the groups where at least one other tuple is similar regarding the given thre-

shold and similarity criterion. If more than one group is found, we simply merge the groups and build the transitive closure this way. For the strict strategy we would have to check all tuples within a group. Conflicts arise not only if more than one group is found, but also if there are conflicting matches within one group. For the strict strategy conflict are handled by completely splitting the groups to one tuple per group. Obviously these implementations imply an $O(n^2)$ complexity for all realistic application scenarios. In section 7 a discussion of possible optimizations is given.

7 Implementation and Optimization

To better understand implementation and optimization issues we have to consider in what kind of scenarios the proposed concepts are to be used. The need to integrate data exists in applications as diverse as data warehousing, information dissemination, and meta search engines for web databases. All of these applications have very different requirements regarding the data volume and the user expectations for certain performance criteria, that should be supported by a system used as an integration platform.

The time complexity of naive implementations of the introduced concepts as described in section 6 is obviously prohibitive in certain scenarios mentioned above. To provide an efficient implementation we use certain index structures to speed up the integration. Because this is done in a heterogeneous environment, we have to deal with the following aspects:

Index support for similarity: For the index support we rely on current research work done in the field of index structures supporting similarity queries, as well as commonly used index structures.

- **Hash structures:** apart from supporting equality matches hash structures can be useful, when the hash function can be applied to a value derived from the actual value and the threshold.
- **B-trees:** for scalar values similarity detection can be considered a range query, where the range is derived from a threshold and some predicate specific distribution.
- **Tries:** in [21] Shang and Merret present an approach to use Tries for efficiently searching by string similarity with an edit distance limit that can be derived from the given threshold in our approach. For data integration purposes it seems reasonable to support certain derivates of Trie algorithms, supporting for instance token indexing and abbreviations.
- **Inverted lists:** for longer text attributes techniques used in the area of information retrieval can be applied.

Global indexing versus autonomy of integrated systems: A major problem is the fact that the integrated systems have to remain autonomous and the data accessed via an integration layer is physically stored in the source systems. If we consider to use index structures to speed up certain integration tasks, several questions arise, such as

- Where to store and use indexes?
  Main memory only, or caching on secondary storage?
- How to create and maintain such structures?
  Create on the fly and discard, or following some update scheme?
- Is it possible to use index structures from source systems? Via open index interfaces or derived queries?

We currently focus on data integration scenarios with views of integrated sources, i.e. the index structures have to be created on the fly and are used from main memory. The usage of existing index structures from source systems or, the persistent storage and maintenance within the integration system is subject of future research.

A dedicated solution for similarity-based grouping opens the possibility to apply indexing as an optimization strategy for avoiding the general cost of $O(n^2)$ for pairwise comparisons of n tuples. Just like equality based duplicate elimination this can be optimized to $O(n \log n)$. Furthermore, the development of user-defined similarity for usage 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.

```
init(Expression expr):
    transform expr to disjunctive normal form
    find index-supported predicates within conjunctions


findRelevantGroups(Tuple t):

    result r_disj := ∅
    for all conjunctions c in expr
(1)     result r_conj := all
        for all index supported predicates involving attribute a
            g := indexScan(t.a, threshold)
            r_conj := r_conj ∩ g
        endfor
(2)     ni := non-index predicates in c
        for all groups g in r_conj
            for all tuples t_g in g
                if evaluate(ni,t,t_g) < threshold
                    remove g from r
                    break
            endfor
        endfor
        r_disj := r_disj ∪ r_conj
    endfor
    return r_disj
```

Figure 7: Index-based algorithm for finding relevant groups

The sketch of the algorithm in Fig. 7 for index-optimized similarity-based grouping is again based on the general framework given in section 6. The optimization avoids the evaluation of the inner nested loop by applying index-based lookup of relevant groups whereever possible.

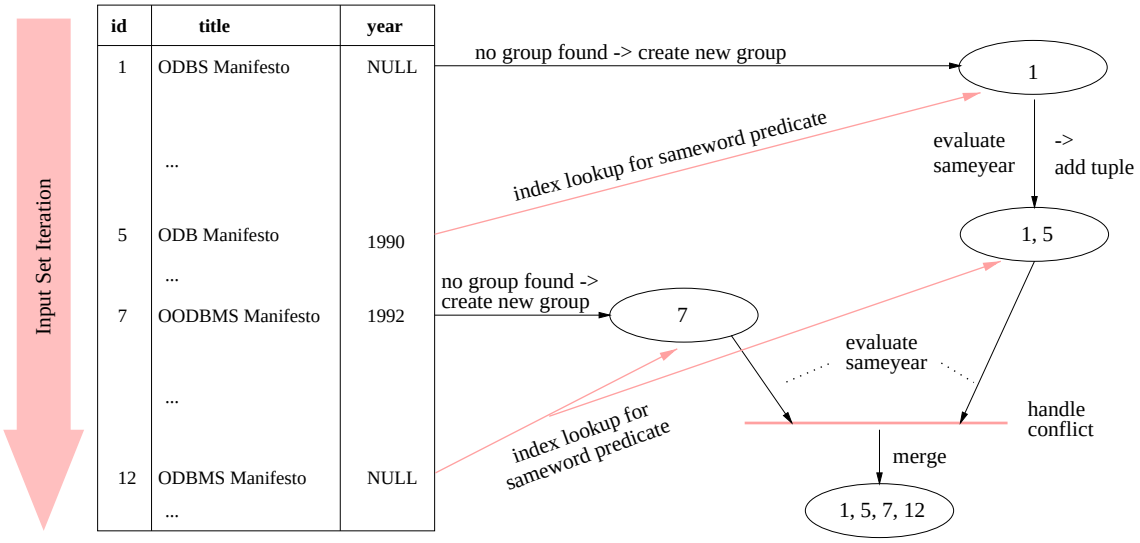


Figure 8: Building groups applying index support

The similarity expression $expr$ is initially transformed to disjunctive normal form and the contained predicates are checked for index support. The iteration and conflict management is identical to the framework presented in section 6, but requires in addition to build the indexes on the fly. For the evaluation of the conjunctions index supported predicates are processed first (1), resulting in sets of possibly relevant groups. This can be done in $O(\log n)$. For predicates without index support the intersection of the previous evaluation is used as an input for simple loop (2). Assuming that the index evaluation returns groups of an approximately constant size independent of n the cost for this loop is constant. This results in an overall complexity of $O(n \log n)$. The threshold t is used to truncate the evaluation of the expression as well as for limiting the search for index supported predicates. Other possible shortcuts and utilization of parallelisms that do not change the overall complexity of the algorithm are not described here for reasons of simplicity. As a simple example consider the following query:

```
select pickBySource(title, 'DBLP')
from NCSTRL union DBLP union CSBIB union CITESEER
group by transitive similarity
on sameword(title) and sameyear(year)
threshold 0.9
```

The predicate **sameword** is a system-defined similarity measure that represents the edit distance and is supported by a trie index and the according lookup algorithms for similarity matches. The **sameyear**-function is user-defined and without index support. It simply accepts missing values and differences of values up to two years. In figure 8 the query processing is sketched for building one group from a data set involving a conflict resolution. Assuming the threshold value of 0.9 translates to an edit distance of 1 for the title attribute, when tuple **7** is processed, no match can be found, because the edit distance to previous tuples is > 1. For tuple **12** two relevant groups are found and the conflict has to be resolved by merging the groups according to strategy of transitive similarity.

Considering the more general concept of context-aware grouping, implementation and optimization are burden to the implementer. Moreover, there are certain tasks common in various application scenarios, like finding, merging and splitting groups, which can be supported by an implementation framework

offering basic functionality as well as other primitives. The implementation of the traditional, context-free grouping is based on sorting or hashing as described in detail in [8]. Even if an expression is used for grouping, the intermediate result after applying this expression can be sorted in order to simplify the partitioning of the relation. 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 a linear order for one of the attributes used for grouping is defined, a sliding window approach could be used [11].

Finally, it should be mentioned that the presented grouping operation can be implemented not only as a language extension as shown in section 4, but also on top of commercial database systems. For example, the new Oracle9i system supports so called *pipelined table functions* which accept cursors as parameters and return tables. Using this feature our **`group by similarity`**-operator could be implemented as a table function processing a cursor of the input relation, a similarity expression as well as threshold and returning a relation of grouped tuples:

```
select *
from table (group_by_context (
                cursor (select * from library),
                'sameword(title) and sameyear(year)', 0.9))
```

The implementation of this function follows the algorithm from Fig. 7. However, because of the required type declaration of the function parameters, this implementation approach lacks support for a generic, type-independent usage.

The proposed approach of context-aware grouping opens a broad range of applications. The disadvantage is the higher complexity of developing grouping functions. Anyway, we currently focus on this solution because of the following reasons:

- 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 and Outlook

Data reconciliation is an important task in data integration. One of the key issues for data reconcilation is the so-called same-object problem, which is aimed at identifying and eliminating duplicate tuples, i.e., tuples representing the same real-world object. A common problem in this context is, that matching of tuples cannot always be decided only based on equality of attribute values. Rather it is necessary to consider *similarity* criteria. Furthermore, after identifying groups potentially describing the same real-world object, a representative object for each group has to be chosen.

In this paper we have presented two extensions to SQL-like query languages, that can be used to address these problems. The **`group-by-context`** clause provides an open mechanism for applying

user-defined functions for grouping purposes. The **`group-by-similarity`** is a specialized language construct on top of context aware grouping to build groups from pairwise similarity of tuples. Because the context aware grouping approach is quite simplistic, it leaves the burden of implementing complex grouping functions to the implementer. Specialized constructs like the similarity grouping are less flexible, but allow for efficient implementations of the often complex grouping functionality. For both approaches, the merging of the tuples of the identified groups is performed via user-defined aggregation functions. Advanced grouping and aggregation 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 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 similarity-based grouping, i.e. an applicable set of system-defined similarity functions and the according dedicated index structures, as well as 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] 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.

[6] 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.

[7] 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.

[8] G. Graefe. Query Evaluation Techniques For Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[9] 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.

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

[11] 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.

[12] S. B. Huffman and D. Steier. Heuristic joins to integrate structured heterogeneous data. In *AAAI Spring Symposium on Information Gathering*, 1995.

[13] J. A. Hylton. Identifying and merging related bibliographic records. Technical Report MIT/LCS/TR-678, Massachusetts Institute of Technology, February 1996.

[14] W. Kent. The breakdown of the information model in multi-database systems. *SIGMOD Record*, 20(4):10–15, December 1991.

[15] Wen-Syan Li. Knowledge gathering and matching in heterogeneous databases. In *AAAI Spring Symposium on Information Gathering*, 1995.

[16] 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.

[17] 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.

[18] 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.

[19] K. Sattler and E. Schallehn. A Data Preparation Framework based on a Multidatabase Language. In *Proc. of Int. Database Engineering and Applications Symposium (IDEAS 2001)*, Grenoble, France, 2001. *To appear*.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.