Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Data & Knowledge Engineering 48 (2004) 361–387

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Efficient similarity-based operations for data integration

Eike Schallehn [*], Kai-Uwe Sattler, Gunter Saake

Department of Computer Science, University of Magdeburg, P.O. Box 4120, Magdeburg D-39106, Germany

Received 6 November 2002; received in revised form 9 April 2003; accepted 12 August 2003

Abstract

Dealing with discrepancies in data is still a big challenge in data integration systems. The problem occurs both during eliminating duplicates from semantic overlapping sources as well as during combining complementary data from different sources. Though using SQL operations like grouping and join seems to be a viable way, they fail if the attribute values of the potential duplicates or related tuples are not equal but only similar by certain criteria. As a solution to this problem, we present in this paper similarity-based variants of grouping and join operators. The extended grouping operator produces groups of similar tuples, the extended join combines tuples satisfying a given similarity condition. We describe the semantics of this operator, discuss efficient implementations for the edit distance similarity and present evaluation results. Finally, we give examples of application from the context of a data reconciliation project for looted art.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Data integration; Data cleaning; Similarity-based operations; Duplicate detection; Similarity join

1. Introduction

In the past few years, there has been a great amount of work on data integration. This includes the integration of information from diverse sources in the Internet, the integration of enterprise data in support of decision-making using data warehouses, and preparing data from various sources for data mining. Some of the major problems in this context—besides overcoming structural conflicts—are related to overcoming conflicts and inconsistencies on the data level. This includes the elimination of duplicate data objects caused by semantic overlapping of some sources, as well as establishing a relationship between complementary data from these sources.

[*] Corresponding author.
E-mail address: eike@iti.cs.uni-magdeburg.de (E. Schallehn).

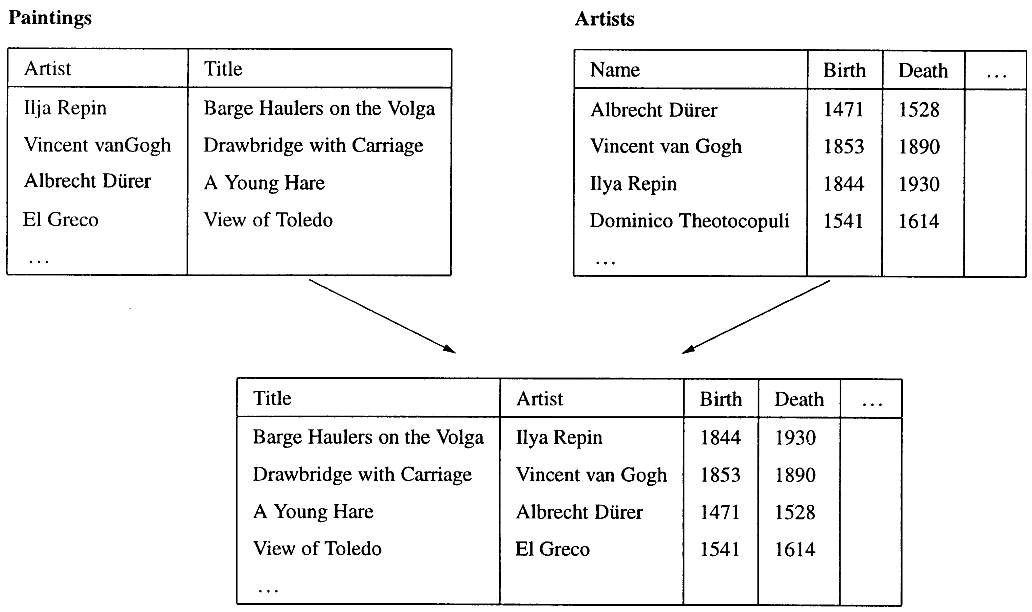
0169-023X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2003.08.004

The implementation of according operations has a significant difference to usual data management operations: only in some rare cases we can rely on equality of attributes. Instead we have to deal with discrepancies in data objects representing the same or related real-world objects which may exist due to input errors or simply due to the autonomy of the sources. Furthermore, the amount of data to be processed in integration scenarios can be equal to or even greater than from a single source, so, efficiency of the implementation becomes a critical issue.

Duplicate elimination is a subtask of data cleaning that comprises further tasks for improving data quality like transformation, outlier detection etc. Assuming SQL-based integration systems, the natural choice for duplicate elimination is the **`group by`** operator using the key attributes of the tuples in combination with aggregate functions for reconciling divergent non-key attribute values.

However, this approach is limited to equality of the key attributes—if no unique key exists or the keys contain differences, tuples representing the same real-world object will be assigned to different groups and cannot be identified as equivalent tuples. The same is true for linking complementary data, which in a SQL system would be done based on equality by the **`join`** operator.

As an example for a similarity join consider an information system on art objects providing information for instance on paintings and their creators. One source may provide a plain collection of these items, but we intend to present additional biographical information on the artists given by a standard catalog integrated from another source. The example shown in Fig. 1 demonstrates three problems common in this application domain. First, due to language issues a number of different spellings or transcriptions of names may exist, like in the case of 'Albrecht Dürer' or 'Ilya Repin'. A common problem in many application domains are inconsistencies due



Paintings

Artist	Title
Ilja Repin	Barge Haulers on the Volga
Vincent vanGogh	Drawbridge with Carriage
Albrecht Dürer	A Young Hare
El Greco	View of Toledo
...	

Artists

Name	Birth	Death	...
Albrecht Dürer	1471	1528	
Vincent van Gogh	1853	1890	
Ilya Repin	1844	1930	
Dominico Theotocopuli	1541	1614	
...			

Title	Artist	Birth	Death	...
Barge Haulers on the Volga	Ilya Repin	1844	1930	
Drawbridge with Carriage	Vincent van Gogh	1853	1890	
A Young Hare	Albrecht Dürer	1471	1528	
View of Toledo	El Greco	1541	1614	
...				

Fig. 1. Example data and result for a similarity join.

to typing errors, like in this case the incorrect writing of ‘Vincent van Gogh’. Whereas both these problems could be handled by string similarity, the problem of pseudonyms as demonstrated by ‘Dominico Theotocopuli’, better known to the world as ‘El Greco’, can be solved by applying thesauri during the join on the artist name.

Fig. 2 demonstrates another problem during data integration, namely the identification and reconciliation of tuples representing the same real-world entity. Assume the input relation represents the combined information on paintings from a number of sources, which may overlap semantically and provide incomplete or imprecise information. In addition to this and the problems mentioned above, the example illustrates that a complex similarity description involving a number of attributes is required. Furthermore, we have to deal with the fact that more than two tuples may represent the same object, and among these representations may exist varying degrees of similarity. Finally, we have to provide means to establish a single representation of identified objects, which for instance can be done based on additional information on data quality of the integrated sources.

In this paper we address these problems and present similarity-based operators for joining and grouping based on previous work [25]. We extend our earlier work by giving clear semantics of the operators, describing the implementation and evaluating optimization techniques. Both operators are based on extended concepts for similarity-based predicates. Major concerns are the new requirements resulting from the characteristics of similarity relationships, most of all atransitivity, and support for the efficient processing of similarity predicates.

The operators have not necessarily to be provided as a language extension, though we did this in our own query engine and use this syntax for illustration purposes. Instead it also can be implemented by utilizing extension mechanisms which are offered by today’s DBMS. The implementation and the evaluation results described in this paper are based on table functions available in Oracle8i.

Title	Artist	Year
Resurrection	El Greco	1579
Resurrection	Dieric Bouts	1460
...		
The Holy Trinity	El Greco	1577
The Holy Trinity	El Greco	16th cen.
...		
Self-Portrait at 28	Albrecht Dürer	1500
Self-Portrait at 28	Albrecht Dürer	
Self Portrait at 28	Albrecht Dürer	1500
...		
Fifteen Sunflowers	Vincent van Gogh	1888
Fifteen Sunflowers	Vincent van Gogh	1889
...		

Title	Artist	Year
Resurrection	El Greco	1579
Resurrection	Dieric Bouts	1460
...		
The Holy Trinity	El Greco	1577
...		
Self-Portrait at 28	Albrecht Dürer	1500
...		
Fifteen Sunflowers	Vincent van Gogh	1888
Fifteen Sunflowers	Vincent van Gogh	1889
...		

Fig. 2. Example data and result for similarity-based duplicate elimination.

The remainder of this paper is organized as follows. After a discussion of related work in Section 2, we describe the characteristics and requirements of similarity predicates useful in data integration in Section 3. The proposed similarity operators are defined with respect to their semantics in Section 4. In Section 5 we describe strategies for an efficient implementation of these operators focusing on edit distances similarity measures. Results of our evaluation are given in Section 6. Finally, in Section 7 we present several aspects of the application of the similarity operations. Section 8 concludes the paper and points out ongoing work.

2. Related work

The concepts described in this paper are intended to be used in data integration and cleaning scenarios. Related topics are from this field and similarity-based data operations, as well as from the field of analytical data processing.

The concept of similarity in databases has been studied thoroughly related to multimedia and information retrieval. While the current focus is mainly on similarity in terms of distances in multi-dimensional spaces, we also consider a broader perspective on similarity, that is for instance more common in the area of case-based reasoning. Overviews are given for instance by Jantke in [15] and Richter in [23]. In [24] Santini and Jain describe characteristics of various similarity measures and the according consequences for similarity-based operations. In Section 3 we put our work into the context of these common definitions of similarity.

Closely related to similarity-based operations is the integration of probabilistic concepts in data management. In [3] Dey and Sarkar propose an extended relational model and algebra supporting probabilistic aspects. Fuhr describes a probabilistic Datalog in [6]. Especially, for data integration issues and the aforementioned problems probabilistic approaches were verified and yielded useful results, as described by Tseng et al. in [28]. In our current research we work with probabilistic concepts on the predicate and operational level, but are well aware that for a limited number of applications extended data models are useful.

The WHIRL system and language described in [2] by Cohen is based on Fuhr's work and uses text-based similarity and logic-based data access as known from Datalog to integrate data from heterogeneous sources. Cohen describes an efficient algorithm to compute the top scoring matches of a ranked result set. The implementation of the similarity predicate uses inverted indices common in the field of information retrieval. A general framework for similarity joins for predicates on data types that can be mapped to multi-dimensional spaces is presented by Shim et al. in [27]. The approach is based on an extended version of the kdB tree.

While efficient implementations of similarity predicates can be provided based on established index structures described above, most of the real-life applications considered in this paper require predicates for string attributes. Though there is a number of similarity measures for strings, namely the edit distance and it's derivates, for which a good overview is given by Navarro in [22], the efficient implementation for large data sets is a current research topic. In [11] Gravano et al. present an approach for similarity-based joins on string attributes using an efficient pre-selection of q-grams for optimization. In short, the approach is based on down-sizing the data sets on which a similarity predicate is evaluated by first doing an equality-based join on substrings of fixed length q. The authors extend and modify this approach to support string tokens based on techniques similar to those used by Cohen in [2,10]. Nevertheless, both approaches require fully materialized data sets and index structures, hence they are not applicable in virtual integration scenarios and introduce a huge space overhead.

Though our approach is not limited to string based predicates, we implemented an edit distance string similarity predicate using a trie as an index structure based on results by Shang and Merret described in [26] for evaluation purposes, which is described in Section 5.

A major focus of our work is the problem of duplicate detection. This problem was discussed extensively in various research areas like database and information system integration [18,31], data cleaning [1,7], information dissemination [30], 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 [13,20], the application of methods known from the area of data mining and even machine learning [17]. Other interesting results came from specific application areas, like for instance digital libraries [8,14]. While these approaches are mostly very specific regarding certain applications, our goal is to provide a more general framework for the process of duplicate detection, that may well include these approaches on the predicate level.

An overview of problems related to entity identification is given in [16]. In [18] 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 [21]. 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. Luján-Mora and Palomar propose a centroid method for clustering in [19]. Furthermore, they describe common discrepancies in string representations and derive a useful set of pre-processing steps and extended distance measures combining edit distance on a token-level and similarity of token sets. In [13] Hernández and Stolfo propose the sliding window approach for similarity-based duplicate identification where a neighborhood conserving key can be derived and describe efficient implementations.

The importance of extended concepts for grouping and aggregation in information integration is emphasized by Hellerstein et al. in [12]. 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 [29] 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 [29] and its usage in data mining is discussed. Our approach builds on this work for the purpose of data reconciliation as described in Section 7.

3. Similarity measures

Similarity based operators like the similarity join and the similarity-based grouping discussed here are based on similarity measures for attribute values and their logical combination. Other operators requiring concepts of similarity include for instance nearest neighbour queries and attribute similarity selections. These concepts currently find their way into commercial data management solutions, or are the topic of ongoing research. This section discusses useful similarity measures, their characteristics and requirements for common applications.

3.1. Basic similarity predicates

We use the following basic terms of similarity measures: let x and y be objects in a given universe of discourse U, a similarity measure is a function $SIM(x,y) \rightarrow [0,1]$. Alternatively a distance measure $d(x,y) \rightarrow \mathbb{R}$ can be used. The latter can be transformed to a similarity measure, for instance using the simple transformation $SIM(x,y) = 1 - \frac{d(x,y)}{\max}$, where max is the maximum difference between objects in U, if applicable. This transformation implies a normalization, though other normalizations of distances within a given range are conceivable. A binary similarity predicate $SIM(x,y) \subseteq U^2$, meaning "y is similar to x", can for instance be derived from a similarity or distance measure using thresholds $t \in [0,1]$ or $k \in \mathbb{R}$ like $SIM(x,y) \Longleftrightarrow SIM(x,y) \geqslant t$ or $SIM(x,y) \Longleftrightarrow d(x,y) \leqslant k$. SIM is in most cases considered as a reflexive, symmetric and atransitive relation.

While a number of approaches to describe similarity stemming from areas like information retrieval, multimedia data management or case-based reasoning exist, one of the major problems of expressing similarity within sets of structured data is, that the concept of similarity is in most cases highly dependent on the given application domain. Therefore, we describe basic similarity measures for common data types and ways of using these as primitives and for combination to derive measures suitable for real life applications.

A widely used measure is the distance d of data points x, y in a metric space S, for instance the *Euclidean Distance* in an n-dimensional space. In a metric space the distance function fulfills the following conditions:

$$\forall x,y \in S, \quad d(x,y) = 0 \Longleftrightarrow x = y \tag{1}$$

$$\forall x,y \in S, \quad d(x,y) = d(y,x) \tag{2}$$

$$\forall x,y,z \in S, \quad d(x,y) \leqslant d(x,z) + d(z,y) \tag{3}$$

Especially the symmetry and the triangular inequality of such a distance measure given in (2) and (3) provide the fundamental for efficient applications, e.g. in information retrieval and data mining. To use such measures, the data objects to be compared solely consist of coordinates in a metric space, or otherwise have to be transformed to represent points in this space, e.g. extracting feature vectors from multimedia data or deriving term-based vector representations of textual data. Supported by multi-dimensional indexing, predicates on these distance measures can be used efficiently, though efficiency is limited by the number of dimensions.

Another well-studied distance measure is the *Levenshtein* or edit distance $edist(p,w)$ on string representations. Certain costs are assigned to operations like insertion, deletion or substitution of characters to transform an original pattern string p to a comparison string w, and the minimal distance is computed. For instance, assuming constant costs of 1 for the three mentioned basic operations, the edit distance of "edna" and "eden" is 2, because the smallest sets of applicable operations are $\{substitute(\#3, \text{“}e\text{”}), substitute(\#4, \text{“}n\text{”})\}$ and $\{insert(\#3, \text{“}e\text{”}), delete(\#5)\}$ both having two operations. Common derivates also allow a transposition operation or apply heuristic-based costs for the operations, e.g. substituting or deleting vowels is usually less expensive than operations on consonants. This distance measure fulfills the three conditions given above for distances in metric space, this way granting efficient implementations. Though the edit distance is a powerful measure to detect inconsistencies in data, for instance for applications in the field of data integration and data cleaning, it is not widely used in current data management solutions. In Sections 5 and 6 we present an efficient implementation of a similarity predicate based on edit distance used with index-based optimization through tries as proposed in [26]. Other distance measures for strings include

- the *Hamming distance*, allowing only substitutions,
- the *episode distance*, allowing only insertions, and
- the *longest common subsequence distance* allowing insertions and deletions.

A good overview of approximate string matching is given in [22]. Similar concepts of edit distances exist for other types of data representations, e.g. special sequences like genome data, spatio-temporal data, trees and graphs in general.

Textual and numerical data, the latter including the special case of one-dimensional data and the difference as a distance measure plus widely used index structures like B-trees, is covered by the approaches introduced so far. A similarity measure for categorical data such as work departments, countries, art form, etc. can be defined, if the categories can be mapped to

- a simple partial order (e.g. alphabetical order),
- a metric space as described above (e.g. location in space), or
- a graph representing categories and their relationships (e.g. a representation of terms in an ontology).

Distance measures for nodes in graphs are not discussed here, but it is worth mentioning that within graphs meaningful distance measures can be defined, that do not fulfill the criteria of symmetry and the triangular inequality. The same problem exists for some similarity measures on sets. $SIM(A,B) = \frac{|A \cap B|}{|A|}$ can be a relevant similarity measure, if we consider A and B as sets of features, for instance of a software product. In this case we can say B is similar to A, if it includes many of the features of A, while the reverse might not necessarily be true. Another common similarity measures for sets given as $SIM(A,B) = \frac{|A \cap B|}{|A \cup B|}$ is symmetric, but it does not yield the results we are interested in, if B includes many additional features. Asymmetric similarity is sometimes considered less important, but can easily become relevant for user-defined similarity measures discussed later on.

3.2. Complex and application-specific similarity

So far we have discussed similarity measures applicable to atomic or homogeneously structured data types independently of a special application scenario. In real-life scenarios the expression of similarity has to deal with additional aspects to improve efficiency and the results of similarity based operations.

Complex similarity conditions: Similarity-based operators have to process tuples or more complex objects. The description of similarity between two of those objects may consist of a combination of more than one similarity predicate for an attribute and may use different similarity measures on them, e.g., for information on paintings in a database we can use the edit distance on artist names and the distance of vector representations for descriptions of the pictures contents.

Application-specific similarity measures: The semantics of values to be compared in given applications is known, which allows the usage of more precise similarity measures based on domain knowledge. Though we could use the edit distance to compare names of persons, we achieve better results if the similarity measure would consider that "Andy Warhol", "A. Warhol" and "Warhol, Andy" most likely refer to the same person.

By using similarity predicates as described above we can simply build *complex similarity conditions* by applying the logical operators $\wedge$, $\vee$ and $\neg$. As an alternative, a fuzzy logic can be applied to similarity measures directly, as proposed for instance in [2]. To reach the level of expressiveness we gain by specifying thresholds as part of every similarity predicate in the former approach, the concept of weighting the desired impact of every similarity measure would have to be added to the latter. An efficient evaluation of a complex similarity condition consisting of similarity predicates is described in Section 5.

Application-specific similarity measures and predicates can be defined in terms of user-defined functions as supported in most database systems. As an example consider a function $distName(x,y)$ that takes into account the various conventions for writing names as described above. The algorithm can remove special characters, tokenize the string, find first letter matches and finally apply $edist(token1, token2)$ on candidate tokens, that possibly represent the last name, to take care of typos or inconsistent spelling of names. There are two practical problems with user-defined similarity measures:

(1) *Efficiency:* it is not conceivable, that index support or similar optimizations, that are given for basic similarity measures, can easily be implemented by the designer of the function.
(2) *Properties:* if the implemented similarity measures may not grant symmetry, this will have a severe impact on the implementation of the similarity based operator.

The problem of efficiency is discussed more detailed in Section 5. The general strategy would be to conjunctively combine the user-defined similarity predicates with index-supported equality or similarity predicates for pre-selection purposes. Asymmetric similarity measures are not considered here, so symmetry remains a requirement that has to be granted by the user-defined measure.

Using similarity measures or predicates for database operations also introduces new requirements resulting from the property of atransitivity of the induced similarity relation. Existing operations in the relational algebra base largely on equivalence relations established through the equality of attribute values. To integrate with these concepts an equivalence relation can be derived from a similarity predicate SIM. Because establishing this equivalence relation is not our major focus here, throughout this paper we use the simple strategy of constructing an equivalence relation SIM_{EQ} by building the *transitive closure* $SIM_{EQ} := SIM^{+}$, i.e. a partition of the universe of discourse U is a maximal set of objects that are similar either directly or indirectly. A more rigid but still simple approach to establish SIM_{EQ} that is suitable in a range of applications requires pairwise similarity of all objects in a partition of U. We refer to the latter as the *strict* strategy. Both

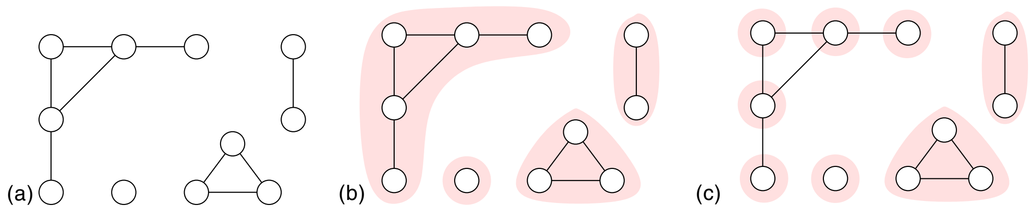


Fig. 3. Equivalence relation on a given similarity relation (a) by (b) transitive and (c) strict similarity.

strategies are outlined in Fig. 3. As an example, consider the strings "ODBMS", "OODBMS, and "DBMS". With an allowed edit distance threshold of 1 they would all be found similar, if we apply the transitive closure strategy, but would not using the strict strategy, because "DBMS" and "OODBMS" have a distance of 2 without the connection via "ODBMS". Especially related to entity identification centroid or density-based clustering techniques proved to be useful strategies for dealing with atransitivity and provide a high level of accuracy, as for instance described in [19,21].

4. Semantics of the similarity operators

In this section we describe the semantics of our similarity-based operators as extensions of the standard relational algebra. We assume the following basic notations: let R be a relation with the schema $S = \{A_1, \ldots, A_m\}$, $t^R \in R$ is a tuple from the relation R and $t^R(A_i)$ denotes the value of attribute A_i of the tuple t^R.

The core concept for similarity-based operations is a *similarity condition*. It expresses whether two tuples are similar in terms of their attribute values or not. Because we define our operators as an extension of the standard relational algebra, we do not deal with probabilities in conditions—by using a similarity threshold we can always rely on boolean values for such conditions. Hence, a similarity condition $\langle sim\_cond \rangle$ is a conjunction of predicates:

$$\langle sim\_cond \rangle = \bigwedge_{i=1}^{m} \langle sim\_pred \rangle (A_i)$$

where $\langle sim\_pred \rangle$ denotes an atomic predicate which could be either *eq* or a "similarity predicate" like $edist_k, diff_k, \ldots$ with the following meanings:

- $eq(A_i)$ means equality of two tuple values of attribute A_i: $t_j(A_i) = t_l(A_i)$;
- $edist_k(A_i)$ is the edit distance between two values with a given threshold k: $\mathrm{edist}(t_j(A_i), t_l(A_i)) \leqslant k$;
- $diff_k(A_i)$ means a numeric approximation: $|(t_j(A_i) - t_l(A_i))| \leqslant k$.

In the same way, any other similarity measure as discussed in Section 3 could be used here.

Similarity join. Based on the similarity condition introduced above the semantics of the similarity join between two relations R_1 and R_2 can be described in a straightforward way. For a given similarity condition $\langle sim\_cond \rangle$ we denote the set of all attributes referenced in this expression as

$$\widetilde{S} = \{A_i | A_i \text{ is referenced in } \langle sim\_cond \rangle\}$$

Then, it holds

$$\begin{aligned} R_1 \bowtie_{\langle sim\_cond \rangle} R_2 = \{t | & \exists t_1 \in R_1 : t_1(S_1 \setminus \widetilde{S}) = t(S_1 \setminus \widetilde{S}) \wedge \\ & \exists t_2 \in R_2 : t_1(S_2 \setminus \widetilde{S}) = t(S_2 \setminus \widetilde{S}) \wedge \\ & \langle sim\_cond \rangle(t_1, t_2) = \mathbf{true}\} \end{aligned}$$

This simply means, a pair of tuples from the relations R_1 and R_2 appears in the result of the join operation if the similarity condition is fulfilled for this two tuples.

Similarity grouping. For defining the semantics of the grouping operator we rely on the algebra operator for standard grouping as presented in database textbooks [5]:

$$\langle grouping\_attrs \rangle^{\mathscr{F}[\langle aggr\_func\_list \rangle](R)}$$

Here $\langle grouping\_attrs \rangle$ is a list of attributes used for grouping relation R, $\langle aggr\_func\_list \rangle$ denotes a list of aggregate functions (e.g., count, avg, min, max, etc.) conveyed by an attribute of relation R. For simplification, we assume that the name of an aggregated column is derived by concatenating the attribute name and the name of the function. An aggregate function f is a function returning a value $v \in \text{Dom}$ for a multi-set of values $v_1, \ldots, v_m \in \text{Dom}$:

$$f(\{|v_1, \ldots, v_m|\}) = v$$

where Dom denotes an arbitrary domain of either numeric or alphanumeric values and the brackets $\{| \cdots |\}$ are used for multi-sets.

We extend this equality-based grouping operator $\mathscr{F}$ with regard to the grouping criteria by allowing an similarity condition and call this new operator Γ:

$$\langle sim\_cond \rangle^{\Gamma[\langle aggr\_func\_list \rangle](R)}$$

This operator again has a list of aggregate functions $\langle aggr\_func\_list \rangle$ with the same meaning as above. However, the grouping criteria $\langle sim\_cond \rangle$ is now a similarity conjunction as introduced above.

The result of Γ is a relation R' where the schema consists of all the attributes referenced in $\langle sim\_cond \rangle$ accompanied with *eq* and the attributes named after the aggregates as described above.

The relation R' is obtained by the concatenation of the two operators γ and ψ which reflect the two steps of grouping and aggregation.

The first operator $\gamma_{\langle sim\_condt \rangle}(R) = \mathscr{G}$ produces a set of groups $\mathscr{G} = \{G_1, \ldots, G_m\}$ from an input relation R. Each group is a non-empty set of tuples with the same schema as R. Furthermore, all tuples t_i^G of a group G are transitively similar to each other regarding the similarity condition $\langle sim\_cond \rangle$:

$$\forall G \in \mathscr{G} : \forall t_i^G, t_j^G \in G : t_j^G \in tsim_{\langle sim\_cond \rangle}(t_i^G)$$

where $tsim_{\langle sim\_cond \rangle}(t)$ denotes the set of all tuples which are in the transitive closure of the tuple t with regard to *sim_cond*:

$$\begin{aligned} tsim_{\langle sim\_cond \rangle}(t) = \{t' | & sim\_cond(t, t') = \mathbf{true} \vee \\ & \exists t'' \in tsim_{\langle sim\_cond \rangle}(t) : sim\_cond(t', t'') = \mathbf{true}\} \end{aligned}$$

and no tuple is similar to any other tuple of other groups

$$\forall G_i, G_j \in \mathcal{G}, i \neq j : \forall t_k^{G_i} \in G_i \nexists t_l^{G_j} \in G_j :$$
$$sim\_cond(t_k^{G_i}, t_l^{G_j}) = \textbf{true}$$

The second operator $\psi_{A_1,\ldots,A_l,\langle aggr\_func\_list\rangle}(\mathcal{G}) = R'$ reconciles (i.e., merges) the tuples from each group and produces exactly one tuple for each group of $\mathcal{G}$ according to the given aggregate functions. Thus, it holds $\forall G \in \mathcal{G}$ with $G = \{t_1^G, \ldots, t_n^G\}$ there is one and only one tuple $t^{R'} \in R'$ with

$$\forall i = 1, \ldots, l : t^{R'}(A_i) = t_1^G(A_i) = t_2^G(A_i) = \cdots = t_n^G(A_i)$$

where $A_1, \ldots, A_l$ are attributes referred by the *eq* predicates of the approximation condition, (i.e., for these attributes all tuples have the same value) and

$$\forall j = l+1, \ldots, m-l : t^{R'}(A_j) = f_{j-l}(\{|t_1^G(A_j), \ldots, t_n^G(A_j)|\})$$

where $f_1, \ldots, f_m$ are aggregate functions from $\langle aggr\_func\_list\rangle$.

Based on these two operators we can finally define the Γ operator for similarity-based grouping as follows:

$$\langle sim\_cond\rangle^{\Gamma[\langle aggr\_func\_list\rangle](R)} = \psi_{A_1,\ldots,A_l,\langle aggr\_func\_list\rangle}(\gamma_{\langle sim\_cond\rangle}(R))$$

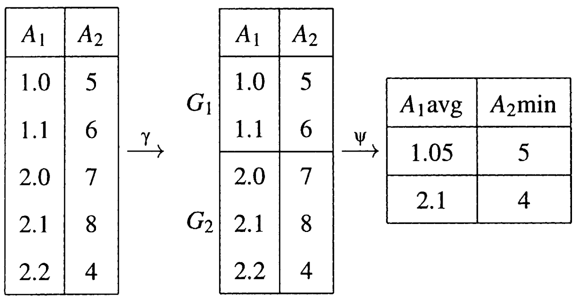
where $A_1, \ldots, A_l$ are again attributes referenced by the *eq* predicates in $\langle sim\_cond\rangle$.

Fig. 4 illustrates the application of these operators for the query:

$$diff_{0.2}(A_1)^{\Gamma[\mathrm{avg}(A_1),\mathrm{min}(A_2)](R)}$$

The input relation consisting of two attributes A_1, A_2 has to be grouped by similar values of A_1, e.g. using the approximation condition "$diff(A_1) \leqslant 0.2$".

In the first step, the γ operator produces two groups G_1 and G_2. Let us now assume an aggregation function list "$\mathrm{avg}(A_1), \mathrm{min}(A_2)$". Then, the ψ operator derives for each of these groups a single tuple as shown in the table at the right-hand side.



A_1	A_2
1.0	5
1.1	6
2.0	7
2.1	8
2.2	4

$\xrightarrow{\gamma}$

	A_1	A_2
G_1	1.0	5
	1.1	6
G_2	2.0	7
	2.1	8
	2.2	4

$\xrightarrow{\psi}$

A_1avg	A_2min
1.05	5
2.1	4

Fig. 4. Application of the grouping operator.

5. Implementation and optimization

In this section we outline our implementation of the similarity-based operators introduced in the previous sections. For an efficient realization dedicated plan operators are required, which implement the semantics described above. That means for instance for the similarity join, even if one formulates a query as follows

```
select*
from r1, r2
where edist(r1.title, r2,title) < 2
```

the similarity join implementation exploiting special index support has to be chosen by the query optimizer instead of computing the Cartesian product followed by a selection. In case of the similarity grouping a simple user-defined function is not sufficient as grouping function, because during similarity grouping the group membership is not determined by one or more of the tuple values but depends on already created groups. In addition, processing a tuple can be conveyed by merging existing groups.

Thus, we describe in the following the implementation of these two plan operators SimJoin and SimGrouping and assume, that the query optimizer is able to recognize the necessity of applying these operators during generating the query plan. This could be supported by appropriate query language extensions, e.g. for the similarity join like

```
select*
from r1 similarity join r2
     on edist(r1.title, r2,title) threshold 0.9
```

where **threshold** specifies the maximum allowed value for the normalized edit distance. For the similarity grouping this could be formulated as follows:

```
select*
from r1
group by similarity on edist(title) threshold 0.9
```

For evaluation purposes we used an index-supported similarity predicate on string attributes using edit distance and tries, that is also described briefly. The following description refers to conjunctively combined, symmetric similarity predicates and the transitive closure strategy for the grouping operator introduced in the previous sections.

5.1. A trie-based similarity predicate for strings

In our approach a similarity predicate p consists of a distance or similarity function and an according threshold. Hence, the index lookup performed requires the actual value $t(A_p)$ of an involved attribute A_{p_i} and the threshold k_p. Currently, for our implementation we focus on edit distances as the primary similarity measure. For this purpose, we have adopted the approach proposed in [26] of using a trie in combination with a dynamic programming algorithm for computing the edit distance. The main idea is to traverse the trie containing the string values of all already processed tuples in depth-first order, trying to find a match with the search pattern, i.e., the attribute value of the currently processed tuple (Algorithm 1). Due to the usage of the edit distance, we must not stop the traversal directly after a found mismatch. Instead an edit operation (insert, remove or replace a character) is applied and the search is continued. Only after exceeding the given threshold, we can stop the traversal and go back to the next subtrie. Hence, the threshold is used for cutting off subtries containing strings not similar to the pattern. In addition, the effort for computing the dynamic programming tables required for determining the edit distance can be reduced, because all strings in one subtree share a common prefix and therefore the same edit distance. We omit further details of this algorithm and refer instead to the original work.

Algorithm 1. Approximate trie searching

```
Globals
    Threshold $k$
    Pattern string $p$, target string $w$

Procedure approxSearch(TrieNode $n$, int $level$)
    begin
        for all child nodes $c$ of $n$
            $w[level]$ := character $z$ of $c$
            if $c$ is a leaf node $\wedge$ $edist\ (w, p, level) < k$
                output tuple-ids for node $c$
            if $edist\ (w, p, level) > k$
                continue /* cut off */
            approxSearch $(c, level + 1)$
        end for
    end
```

In our implementation of the previously introduced operators tries are created on the fly for each grouping attribute or join predicate which appears together with an edit distance predicate. Such a trie stores not only the actual string values but also the tuple-id of the associated tuple. Therefore, besides inserting new string values no updates on the trie are necessary.

5.2. Similarity join

The implementation of a similarity join outlined in this section is quite straightforward. Like for conventional join operators index support for predicates can be exploited to improve performance by reducing the number of pairwise comparisons. However, the different predicates of a similarity expression require different kinds of index structures:

- For equality predicates $eq(A_i)$ common index structures like hash tables or B-trees can be utilized.

- Numeric approximation predicates like $diff_k(A_i)$ can be easily supported by storing the minimum and maximum value of the attribute for each group.
- For string similarity based on edit distances $edist(A_i)$ tries are a viable index structure, as previously introduced.
- For the other similarity predicates discussed in Section 3 index support is given, for instance through multi-dimensional indexes like R-trees and its derivates on data mapped to a metric space.

Given such index structures a join algorithm can be implemented taking care of the various kinds of indexes. In Algorithm 2 a binary join for two relations R_1 and R_2 is shown, assuming that indexes for relation R_2 either exist or were build on the fly in a previous processing step. The result of this algorithm is a table of matching tuples for usage described later on. Alternatively, result tuples can be produced for pipelined query processing directly at this point. The notations I_{p_i} and k_{p_i} refer to the index on predicate p_i and the specified threshold, respectively. A_{p_i} refers to the involved attribute.

Algorithm 2. Processing a tuple from join relation R_1 during similarity join

```
Globals
    Conjunctive join condition $c = p_1 \wedge \cdots \wedge p_n$
    Set of indexes $I_{p_i}, 1 \leqslant i \leqslant n$ on join relation $R_2$
      for index supported predicates
    Mapping table tid_tid for matching tuples

Procedure processTuple(Tuple $t$)
    begin
        for all index supported equality predicates $p_i$
            set of tuples $s_{conj} := indexScan(I_{p_i}, t(A_{p_i}))$
        end for
        for all index supported similarity predicates $p_i$
            $s_{conj} := s_{conj} \cap indexScan(I_{p_i}, t(A_{p_i}), k_{p_i})$
        end for
        for all tuples $t_l \in s_{conj}$
            boolean $similar :=$ true
            for all non-index supported predicates $p_i$
                $similar := similar \wedge$
                    $evaluate(p_i, k_{p_i}, t(A_{p_i}), t_l(A_{p_i}))$
                if not $similar$ break
            end for
            if similar insert $(t,t_l)$ in tid_tid
        end for
    end
```

As a side note, more complex similarity conditions could easily be supported by adding disjunctions. The similarity condition c can be transformed to disjunctive normal form. For all conjunctions of $c = \bigvee_{i=1}^{m} conj_i$ the s_{conj_i} are computed and the set of relevant groups would be $s_{disj} = \bigcup_{i=1}^{m} s_{conj_i}$.

5.3. Similarity-based grouping

Like the join operator, the similarity-based grouping operator is based on the efficient evaluation of similarity predicates, but in addition has to deal with problems arising from the atransitivity of similarity relations. The goal of a grouping operator is to assign every tuple to a group. A naive implementation of the similarity-based operator would work as follows:

(1) Iterate over the input set and process each tuple by evaluating the similarity condition with all previously processed tuples. Because these tuples were already assigned to groups, the result of this step is a set of groups.
(2) If the result set is empty, a new group is created, otherwise the conflict is resolved by merging the groups according to the transitive closure strategy.

The strict grouping strategy would in contrast require pairwise similarity between all tuples in a group. In case of any conflict with a found group or between more than one found groups, existing groups would be split and not considered during further processing. This behavior can be utilized to provide pipelined processing of the operator.

Obviously, the previously described naive implementation would lead to $O(n^2)$ time complexity for an input set of size n. Similar to processing a similarity join we assume that there are index-supported predicates for equality and similarity, and in addition, predicates like user-defined similarity predicates, that can not be supported by indexes, the following optimized Algorithm 3 was implemented. Please note that this algorithm implements only the γ operator as described in Section 4, because the ψ operation corresponds to the traditional projection/aggregation operation.

Algorithm 3. Processing a tuple during similarity grouping

```
Globals
  Conjunctive similarity condition $c = p_1 \wedge \cdots \wedge p_n$
  Set of indexes $I_{p_i}, 1 \leqslant i \leqslant n$
    for index supported predicates
  Mapping table $gid\_tid$ assigning tuples to groups

Procedure processTuple(Tuple $t$)
  begin
    set of groups $r_{conj}$ := all groups from $gid\_tid$
    for all index supported equality predicates $p_i$
      set of tuples $s := indexScan(I_{p_i}, t(A_{p_i}))$
      $r_{conj} := r_{conj} \cap gid\_tid(s)$
    end for
    for all index supported similarity predicates $p_i$
```

```
      set of tuples $s := indexScan(I_{p_i}, t(A_{p_i}), k_{p_i})$
      $r_{conj} := r_{conj} \cap gid\_tid(s)$
    **end for**
    **for all** groups $g_j \in r_{conj}$
      boolean $member$ := **false**
      **for all** tuples $t_l \in g_j$
        boolean $similar$ := **true**
        **for all** non-index supported predicates $p_i$
          $similar := similar \wedge$
            $evaluate(p_i, k_{p_i}, t(A_{p_i}), t_l(A_{p_i}))$
          **if** not $similar$ **break**
        **end for**
        $member := member \vee similar$
        **if** $member$ **break**
      **end for**
      **if** not $member r_{conj} := r_{conj} - g_j$
    **end for**
    **if** $r_{conj} = \emptyset$ group $g$ := new group in $gid\_tid$
    **else** group $g$ := merge all $r_{conj}$ in $gid\_tid$
    insert $t$ in $g$
  **end**
```

Similar to join processing, for each tuple t the algorithm tries to find a minimal set r_{conj} of groups that relate to t by applying index-supported equality and similarity predicates first. This can even be improved, if information about the costs and selectivity of the index-based predicate evaluation exist and an according processing order is established. A pairwise comparison is only performed for tuples from this small subset of groups in the second half of the algorithm. As a result of this procedure the mapping table *gid_tid* is adjusted to the newly found group structure.

Implementing the described similarity operators in a SQL DBMS as native plan operators supporting the typical iterator interface [9] requires significant modifications to the database engine and therefore access to the source code. So, in order to add these operators to a commercial system the available programming interfaces and extensibility mechanisms should be used instead. Most modern DBMS support so-called table functions which can return tables of tuples, in some systems also in a pipelined fashion. In this way, our operators can be implemented as table functions consuming the tuples of a query, performing the appropriate similarity operation and returning the result table. For example, a table function `sim_join` implementing Algorithm 2 and expecting two cursor parameters for the input relations and the similarity join condition could be used as follows:

```
select*
from table (sim_join (cursor (select* from data1),
          cursor (select* from data2),
          'edist (data1. title, data2. title) < 2'))
```

This query performs a similarity join with one similarity predicate on the title attributes in two given relations, where the edit distance between the string values in this field is less than 2. However, a problem of using table functions for implementing query operators are the strong typing restrictions: for the table functions a return type has always to be specified that prevents to use the same function for different input relations.

As one possible solution we have implemented table functions using and returning structures containing generic tuple identifiers (e.g., Oracle's `rowid`). So, the SimGrouping function produces a tuple of tuple identifier/group identifier pairs, where the group identifier is an artificial identifier generated by the operator. Based on this, the result type `gid_tid_table` of the table function is defined as follows:

```
create type gid_tid_t as object
  gid int, tid int;
create type gid_tid_table
  is table of gid_tid_t;
```

Using a grouping function `sim_grouping` a query can be written as the following query:

```
select ...
from table (sim_grouping (
         cursor (select rowid, * from raw_data),
         'edist(title) < 2')) as gt,
       raw_data
where raw_data.tid = gt.tid
group by gt.gid
```

This query groups tuples having an edit distance of less than 2 either directly or indirectly. A discussion of according aggregate functions is given in Section 7.

Our approach allows to implement the function in a generic way, i.e., without any assumption on the input relation. In order to apply aggregation or reconciliation to the actual attribute values of the tuples, they are retrieved using a join with the original relation, whereas the grouping is performed based on the artificial group identifiers produced by the grouping operator.

In the same way, the SimJoin operator can be implemented as a table functions returning pairs of tuple identifiers that fulfill the similarity condition and are used to join with the original data.

6. Evaluation

The similarity-based grouping and join operators described in Section 4 were implemented as part of our own query engine and, alternatively, using the extensibility interfaces of the commercial database management system Oracle as outlined in Section 5. For evaluation purposes the latter implementation was used. The test environment was a PC system with a Pentium III (500 MHz) CPU running Linux and Oracle 8i. The extended operators and predicates were implemented using C++. All test results refer to our implementation of the string similarity predicate based on the edit distance and supported by a trie index. A non-index implementation of the predicate is provided for comparison. Indexes are currently created on the fly and maintained in main memory only during operator processing time, which appears to be a reasonable approach considering the targeted data integration scenarios. The related performance impact is discussed below.

For the grouping operator test runs separate data sets containing random strings were created according to the grade of similarity to be detected, i.e. for one original tuple between 0 and 3 copies were created that fulfilled the similarity condition of the test query. The test query consisted of an edit distance predicate on only one attribute. Using the edit distance with all operations having a fixed cost of 1 and a edit distance threshold k on an attribute, each duplicate tuple had between 0 and k deletions, insertions or transpositions. As the number of copies and the numbers of applied operations on the string attributes were equally distributed, for n original tuples the total size of the data set to be processed was approximately $3 * n$ with an average distance of $\frac{k}{2}$ among the tuples to be detected as similar. Furthermore, to check the accuracy of the approach, additional information about the creation of the input set were stored with duplicate tuples. Part of an input relation is shown in Fig. 5.

Grouping based on an exact matching ($k = 0$) has the expected complexity of $O(n)$, which results from the necessary iteration over the input set and the trie lookup in each step, which for an exact match requires average word-length comparisons, i.e. can be considered $O(1)$. This conforms to equality based grouping with hash table support. For a growing threshold the number of comparisons, i.e. the number of trie nodes to be visited, grows. This effect can be seen in Fig. 6, where the complexity for $k = 1$ appears to be somewhat worse than linear, but still reasonably efficient.

Actually, the complexity grows quickly for greater thresholds, as larger regions of the trie have to be covered. The dynamic programming approach of the similarity search ensures that even for the worst case each node is visited only once, which results in equal complexity as pairwise similarity comparison, not considering the cost for index maintenance etc. The currently used

Id	Data	CopyOf	Edist
1	abhfhfhhflk		
2	huiqwerzhads		
3	hdhhhhrrrr		
…	…	…	…
567	abhffhhflk	1	1
568	ahbfhfhhfk	1	2
569	huiqwerzhads	2	0
570	hdhhhrrrr	3	1
571	hdhhhhrr	3	2
…	…	…	…

Fig. 5. Example input relation.

main memory implementation of the trie causes a constant overhead per insertion. Hence, the $O(n^2)$ represents the upper bound of the complexity for a rising threshold k, just like $O(n)$ is the lower bound. For growing thresholds the curve moves between these extremes with growing curvature. This is a very basic observation that applies to similarity based operations like similarity-based joins and selections as well, the latter having a complexity between $O(1)$ and $O(n)$. The corresponding test results are shown in Fig. 7.

The previous test results were presented merely to make a general statement about the efficiency of the similarity-based grouping operator. An interesting question in real life scenarios would be, how the operator performs on varying ratios of duplicates in the tested data set. In Fig. 8 the dependency between the percentage of duplicates and the required processing time is given for the threshold $k = 2$. While the relative time complexity remains, the absolute processing time decreases for higher percentages of detectable duplicates. Obviously, and just as expected, using a

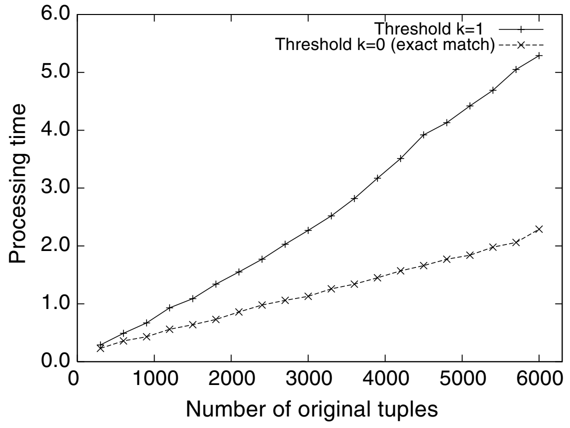


Fig. 6. Grouping with exact match and threshold $k = 1$.

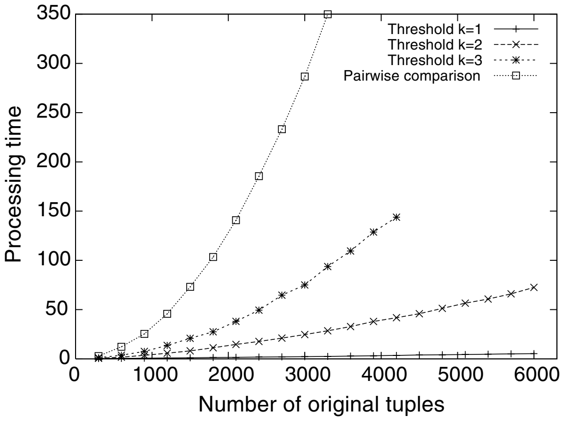


Fig. 7. Grouping with varying thresholds $k \geqslant 1$ and the naive approach of pairwise comparisons.

similarity measure is more efficient, if there actually is similarity to detect. Otherwise, searching the trie along diverging paths represents an overhead that will not yield any results.

We received similar results for the described implementation of a similarity join. The test scenario consisted of two relations R_1 and R_2, with a random number of linked tuples, i.e. for each tuple in R_1 there were between 0 and 3 linked records in R_2 and the join attribute values were within a maximum edit distance. The results are shown in Fig. 9. As the implementation of the join operation is similar to the grouping operation the complexity is between $O(n)$ and $O(n^2)$ depending on the edit distance threshold.

To get an impression of the complexity of using a trie for lookup with threshold k, we analyze a specialized scenario. We assume a completely filled trie with depth d for an alphabet Σ of size $b = |\Sigma|$. Obviously, the trie distinguishes b^d words, and a lookup for $k = 0$ touches d inner nodes of the trie (where $d = \log_b(n)$). A scan for $k = 0$ has therefore complexity $O(m \log n)$ for a relation size m and n stored values in the trie.

For $k = 1$, we concentrate on the situation where one letter is substituted by another letter, i.e. for simplification we consider the *Hamming Distance*. The substitution can occur at any position

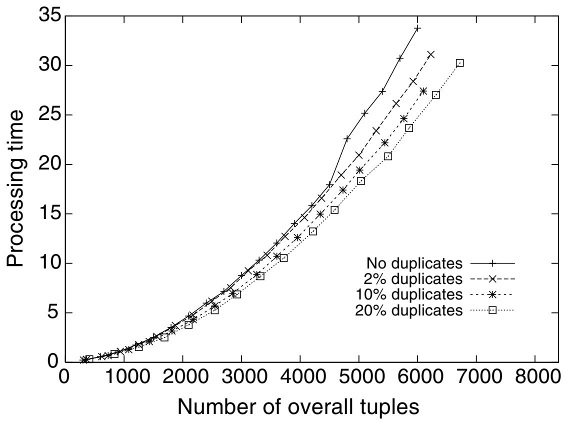


Fig. 8. Grouping with varying percentage of duplicates in the test data sets.

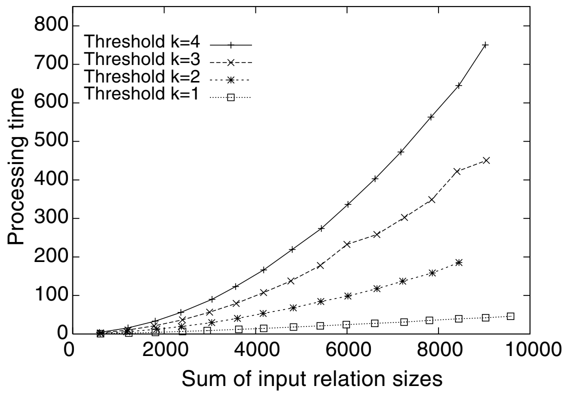


Fig. 9. Results for varying thresholds $k \geqslant 1$ for a similarity join.

of the word, and the letter can be changed to any other letter. Therefore, we have $(b-1)d$ paths through the trie representing all words which differ exactly in one letter from the search word. Averaging over all words, we can assume the difference position is in the middle of the word. All nodes before this 'switch' position are already visited (for the $k = 0$ case), all nodes of the path after this position are inspected newly only for the chosen alternative. Hence, for a path in average half of the nodes have to be counted. As a result, we have to visit $d/2(b-1)d$ nodes for inspecting the relevant part of the trie. Since $d = \log_b(n)$, we have a scan complexity of $O(m \log^2 n)$ for a threshold k.

For $k = 2$, we have two switch positions, and can argue that the complexity increases in a similar way when we locate the second switch position inside the second part of the words (however, we can not simply use the factor $1/2$ as for the remaining nodes but have to sum it up for all switch positions).

However, of course we do not have completely filled tries. For a good distribution of values, we should still have a logarithmic depth of the trie. Moreover, we will have more paths which have not to be followed to the end because the corresponding word is not in the trie. Our simplified scenario can therefore be used for a general complexity border. For the edit distance, the case of substituting a letter is the most expensive because we have to follow at any possible occurrence position $b-1$ alternatives. Therefore, this case dominates the other relevant word modifications.

7. Applications

In this section we give some examples of applications from real-world scenarios of the described similarity operations. These operations are used in a Internet database project for looted arts, that facilitates the registration of and the search for cultural assets. Registering new objects in the database can result in "duplicates" if the particular object was already reported by another user or institution but with slightly different descriptions. Furthermore, The data in the database can be enriched by external information, e.g. from artist catalogues. Due to possible different transcriptions for example of artist names, "approximate" join conditions are necessary.

The problem of duplicates can be solved by applying the similarity-based grouping operations. Using an appropriate similarity predicate (see below for a discussion) potential redundant objects can be identified. In our application a similarity predicate consisting of a combination of artist and title comparisons produces good results. For the artist name we have implemented a special similarity predicate taking different variants of first-name/last-name combinations into account. So, a typical query is formulated as follows:

```
select ...
from data
group by similarity on sim_person(artist)
                   and sim_edist(title)
threshold 0.8
```

However, this is only the first step towards "clean" data: From each group of tuples a representative object has to be chosen. This merging or reconciliation step is usually performed in SQL using aggregate functions. But, in the simplest case of the builtin aggregates one is able only to compute minimum, maximum, average etc. from numeric values. As an enhancement modern DBMS provide support for user-defined aggregation functions (UDA) which allow to implement application-specific reconciliation functions. However, these UDAs are still too restricted for reconciliation because they support only one column as parameter. Here, the problem is to choose or compute a merged value from a set of possible discrepant values without looking at any other columns. We can mitigate this problem by allowing more than one parameter or by passing a structured value as parameter to the function.

In particular for reconciliation purposes we have defined a set of such enhanced aggregate functions including the following:

- `pick_where_eq` (*v, col*) returns the value of column *col* of the first tuple, where the value of *v* is true, i.e., $\neq 0$. In case of a group consisting of only one tuple, the value of this tuple is returned independently of the value of *v*.
- `pick_where_min` (*v, col*) returns the value of column *col* of the tuple, where *v* is minimal for the entire relation or group, respectively.
- `pick_where_max` (*v, col*) returns the value of column *col* of the tuple, where *v* is maximal.
- `to_array` (*col*) produces an array containing all values from column *col*.

With the help of these functions several reconciliation policies can easily be implemented as shown in the following. In a first example, we assume that the final value for column `col` of each group has to be taken from the tuple containing the most current date, which is represented as column `m_date`:

```
select max(m_date), pick_where_max(m_date, col), ...
from data
group by ...
```

In the second example each tuple contains a column `src` describing the origin in terms of the source of the tuple and this way, imitating a source-aware integration view. Assuming a "preferred source" reconciliation strategy, where in case of a conflict the value from source S_P is selected, we could formulate the query as follows:

```
select pick_where_eq(src = 'S_P', col), ...
from data
group by ...
```

Finally, for allowing the user to decide about the resolved value in an interactive way, the `to_array` can be used to collect the list of conflicting values:

```
select to_array(col), ...
from data
group by ...
```

In summary, user-defined aggregation functions provide a viable way to implement specific reconciliation strategies, especially with the extension described above. Combined with powerful grouping operators they make it possible to support advanced cleaning tasks.

Another application-specific question is, how to specify the similarity predicate for similarity joins or grouping consisting of the similarity or distance measure itself and the threshold. If the chosen threshold has such a major impact on the efficiency of similarity-based operations, as described in Section 6, the question is how to specify a threshold to meet requirements regarding efficiency and accuracy. Actually, this adds complexity to the well studied problem of over- and under-identification, i.e. falsely qualified duplicates. Information about the distance or similarity distribution can be used for deciding about a meaningful threshold, as well as for refining user-defined similarity predicates. Distance distributions usually conform to some natural distribution, according to the specific application, data types and semantics. Inconsistencies, such as duplicates, cause anomalies in the distribution, e.g. local minima or points of extreme curvature. Fig. 10 depicts the edit distance distribution for one of our sample sets from Section 6 of 4000 tuples having approximately 20% duplicates with an equal distribution of 0, 1, or 2 edit operations to some original tuple, which is apparent in the chart. To actually choose a threshold based on such a distribution, aspects of efficiency as well as quality of the duplicate detection process have to be considered. Hence, setting $k = 2$ could be a reasonable result drawn from this chart alone.

While the previous anomaly in Fig. 10 was created intentionally, similar effects result from the integration of overlapping data sets in real applications. Fig. 11 shows a result for a sample consisting of approximately 1.600 titles starting with an "E" from integrated sources of data on cultural assets. Nevertheless, drawing the same conclusion of setting the edit distance threshold to receive a useful similarity predicate would lead to a great number of falsely identified tuples. For short titles there would be too many matches, and longer titles often do not match this way, because the length increases the number of typos etc.

Better results can be achieved by applying a relative edit distance

$$rdist(x,y) = 1 - \frac{edist(x,y)}{\max(x.length, y.length)}$$

as a similarity measure as introduced in Section 3. The algorithm introduced in Section 5 can easily be adjusted to this relative distance. Fig. 12 shows the distribution of relative edit distances in the previously mentioned example relation. Using the first global minimum around 0.8 as a

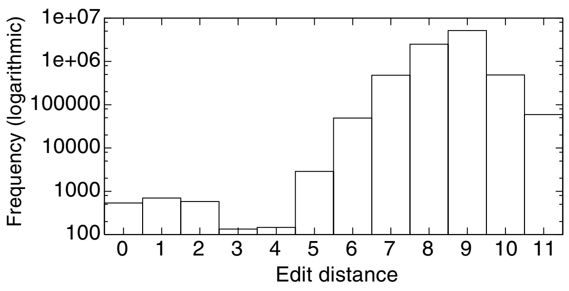


Fig. 10. Edit distance distribution of random strings in the test data set with 20% duplicates of $k_{\max} = 2$.

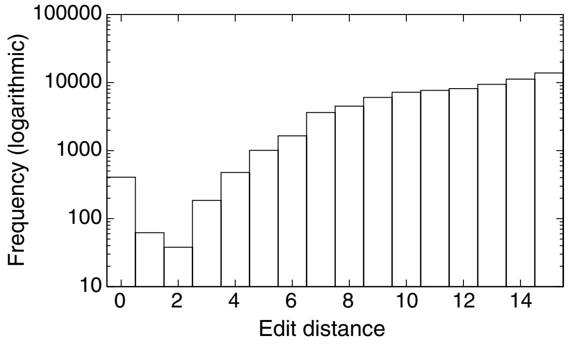


Fig. 11. Edit distance distribution in an integrated and sampled data set on cultural assets.

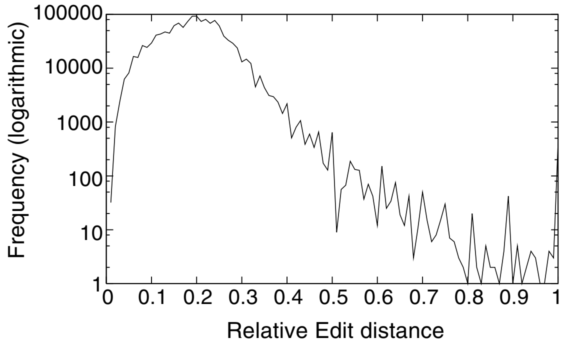


Fig. 12. Relative edit distance distribution.

threshold, and analyzing matches in this area shows that it provides a good ratio of very few over- and under-identified tuples.

A successive adjustment of similarity predicates using information from analytical data processing is also of interest for the creation of user-defined similarity predicates. For instance, using the edit distance on author names and their various representations will not yield any meaningful results. Combining analytical processing and a stepwise addition of techniques like tokenizing, weighted first letter matching etc., as mentioned in Section 3 quickly leads to more meaningful distributions, that can be used to derive a threshold value.

8. Conclusions

In this paper we presented database operators for finding related data and identifying duplicates based on user-specific similarity criteria. The main application area of our work is the integration of heterogeneous data where the likelihood of occurrence of data objects representing related or the same real-world objects though containing discrepant values is rather high. Intended as an extended grouping operation and by combining it with aggregation functions for merging/reconciling groups of conflicting values our grouping operator fits well into the relational algebra framework and the SQL query processing model. In a similar way, an extended join operator takes similarity predicates used for both operators into consideration. These operators can be utilized in ad-hoc queries as part of more complex data integration and cleaning tasks.

Furthermore, we have shown that efficient implementations have to deal with specific index support depending on the applied similarity measure. For one of the most useful measures for string similarity (particularly for shorter strings) we have presented a trie-based implementation. The evaluation results illustrate the benefit of this approach even for relatively large datasets. Though we focused in this paper primarily on the edit distance measure, the algorithm for similarity grouping is able to exploit any kind of index support.

A still open issue is the question how to find and specify appropriate similarity criteria. In certain cases, basic similarity measures like the edit distance are probably not sufficient. As described in Section 3, application-specific similarity measures implementing domain heuristics (e.g. permutation of first name and last name) based on basic edit distances is often a viable approach. However, choosing the right thresholds and combinations of predicates during the design phase of an integrated system often requires several trial-and-error cycles. This process can be supported by analytical processing steps as shown in Section 7 and according tools. Such tools should allow an interactive investigation of analytical results as well corresponding samples from the data level, and are part of our information fusion workbench [4]. Providing the similarity-based operators as query primitives instead of dedicated application tools simplifies this and opens the opportunity for optimization.

In future work, we plan to investigate support for further similarity measures, e.g. for vector representation of longer strings, in order to build a complete framework for similarity grouping.

References

[1] D. Calvanese, G. de Giacomo, M. Lenzerini, D. Nardi, 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] W. Cohen, Integration of heterogeneous databases without common domains using queries based on textual similarity, in: L.M. Haas, A. Tiwary (Eds.), SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, ACM Press, Seattle, Washington, USA, 1998, pp. 201–212.

[3] D. Dey, S. Sarkar, A probabilistic relational model and algebra, ACM Transactions on Database Systems 21 (3) (1996) 339–369.

[4] O. Dunemann, I. Geist, R. Jesse, K.-U. Sattler, A. Stephanik, A database-supported workbench for information fusion: infuse, in: C.S. Jensen, K.G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, M. Jarke (Eds.), Advances in Database Technology—EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, 25–27 March, Proceedings, volume 2287 of Lecture Notes in Computer Science, Springer, 2002, pp. 756–758.

[5] R. Elmasri, S.B. Navathe, Fundamentals of Database Systems, second ed., Benjamin/Cummings, Redwood City, CA, 1994.

[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, 1995, pp. 282–290.

[7] H. Galhardas, D. Florescu, D. Shasha, E. Simon, AJAX: an extensible data cleaning tool, in: W. Chen, J. Naughton, P.A. Bernstein (Eds.), Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, vol. 29(2), Dallas, Texas, 2000, p. 590.

[8] C.L. Giles, K.D. Bollacker, S. Lawrence, Citeseer: an automatic citation indexing system, in: DL'98: Proceedings of the 3rd ACM International Conference on Digital Libraries, 1998, pp. 89–98.

[9] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73–170.

[10] L. Gravano, P. Ipeirotis, N. Koudas, D. Srivastava, Text joins for data cleansing and integration in an rdbms, 2003.

[11] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: Proceedings of the 27th International Conference on Very Large Data Bases(*VLDB'01*, Morgan Kaufmann, Orlando, 2001, pp. 491–500.

[12] J.M. Hellerstein, M. Stonebraker, R. Caccia, Independent, open enterprise data integration, IEEE Data Engineering Bulletin 22 (1) (1999) 43–49.

[13] M.A. Hernández, S.J. Stolfo, The merge/purge problem for large databases, in: M.J. Carey, D.A. Schneider (Eds.), Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, 1995, pp. 127–138.

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

[15] K.P. Jantke, Nonstandard concepts of similarity in case-based reasoning, in: H.-H. Bock, W. Lenski, M.M. Richter (Eds.), Information Systems and Data Analysis: Prospects—Foundations—Applications, Proceedings of the 17th Annual Conference of the GfKl, University of Kaiserslautern, 1993, Kaiserslautern, 1994, Springer, Berlin, pp. 28–43.

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

[17] W.-S. Li, Knowledge gathering and matching in heterogeneous databases, in: AAAI Spring Symposium on Information Gathering, 1995.

[18] E.-P. Lim, J. Srivastava, S. Prabhakar, J. Richardson, Entity identification in database integration, in: International Conference on Data Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, 1993, pp. 294–301.

[19] S. Luján-Mora, M. Palomar, Reducing inconsistency in integrating data from different sources, in: M. Adiba, C. Collet, B.P. Desai (Eds.), Proceedings of the International Database Engineering and Applications Symposium (IDEAS 2001), IEEE Computer Society, Grenoble, France, 2001, pp. 219–228.

[20] A.E. Monge, C.P. Elkan, The field matching problem: algorithms and applications, in: E. Simoudis, J.W. Han, U. Fayyad (Eds.), Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), AAAI Press, London, 1996, p. 267.

[21] A.E. Monge, 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.

[22] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.

[23] M.M. Richter, Classification and learning of similarity measures. SEKI Report SR-92-18, Fachbereich Informatik, Universität Kaiserslautern, 1992.

[24] S. Santini, R. Jain, Similarity measures, IEEE Transactions on Pattern Analysis and Machine Intelligence 21 (9) (1999) 871–883.

[25] E. Schallehn, K. Sattler, G. Saake, Advanced grouping and aggregation for data integration, in: Proceedings of the 10th International Conference on Information and Knowledge Management, CIKM'01, Atlanta, GA, 2001, pp. 547–549.

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

[27] K. Shim, R. Srikant, R. Agrawal, High-dimensional similarity joins, in: Proceedings of the 13th International Conference on Data Engineering (ICDE'97), IEEE, Washington, Brussels, Tokyo, 1997, pp. 301–313.

[28] F. Tseng, A. Chen, 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, 1992, pp. 176–183.

[29] H. Wang, 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, K.-Y. Whang (Eds.), Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00), Morgan Kaufmann, Cairo, Egypt, 2000, pp. 166–175.
[30] T.W. Yan, H. Garcia-Molina, Duplicate removal in information dissemination, in: Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95), Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1995, pp. 66–77.
[31] G. Zhou, R. Hull, R. King, 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.

Eike Schallehn is a scientific assistant at the database research group of the University of Magdeburg, Germany. After receiving his diploma in computer science in 1996 he worked for two companies providing object-oriented database solutions and services in the field of object-oriented software development, respectively. Since 1999 he has been doing research in the field of information fusion and database integration focusing on query processing in heterogeneous environments, especially similarity-based operations. Other research and application fields of interest include self-tuning databases and digital libraries.

Kai-Uwe Sattler is an assistant professor at the database research group at the University of Magdeburg, Germany. He holds a Ph.D. in computer science from the University of Magdeburg. In 2003 he obtained his habilitation at the same University. He has been a visiting assistant professor at the University of California at Davis. He is co-author of several books on database systems and co-editor of the German database journal Datenbank-Spektrum. His research interests include query processing in heterogeneous databases, information integration, and Web databases.

Gunter Saake is full professor for the area "Databases and Information Systems" at the University of Magdeburg. He received the Ph.D. degree in Computer Science from the Technical University of Braunschweig, Germany, in 1988. From 1988 to 1989 he was a visiting scientist at the IBM Heidelberg Scientific Center where he joined the Advanced Information Management project. His research interests include conceptual design of data base applications, query languages for complex data base structures and languages, semantics and methodology for object-oriented system specification and application development in distributed and heterogeneous environments. Besides being author and co-author of scientific publications, he is author of a book "Object-oriented Modelling of Information Systems", co-author of three lecture books on Database Concepts, a book on Java and Databases, a lecture book on Object Databases and a book on Building efficient applications using Oracle8 (all in German).