Bridging Database Applications and Declarative Similarity Matching

Leonardo Andrade Ribeiro[1], Natália Cristina Schneider[2]
Andrei de Souza Inácio[3], Harley Michel Wagner[3], Aldo von Wangenheim[3]

[1] Universidade Federal de Goiás, Brazil
laribeiro@inf.ufg.br
[2] Universidade Federal de Lavras, Brazil
natalia_schneider@comp.ufla.br
[3] Universidade Federal de Santa Catarina, Brazil
{andrei, harley, awangenh}@inf.ufsc.br

Abstract. Effective manipulation of string data is of fundamental importance to modern database applications. Very often, textual inconsistencies render equality comparisons meaningless and strings have to be matched in terms of their similarity. Previous work has proposed techniques to express similarity operations using declarative SQL statements. However, the non-trivial issue of embedding similarity support into object-oriented applications has received little attention. Particularly, declarative similarity matching requires maintenance of a number of auxiliary tables for each combination of similarity function and string attribute in query predicates. As a result, persistence of application objects is severely complicated, thereby leading to excessively intricate and hardly reusable application code. In this article, we present SimDataMapper, an architectural pattern to provide easy and flexible integration of similarity matching with applications and programming environments. We developed an efficient implementation largely based on standard SQL, experimentally evaluate its performance, and compare it with alternative approaches.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems

Keywords: advanced query processing, architectural patterns, object persistence, similarity matching, string databases

1. INTRODUCTION

Virtually all modern database applications deal with string data. Indeed, important information stored in databases such as people names, addresses, and product descriptions are represented as strings. Very often, string data have inconsistencies owing to a variety of reasons, such as misspellings and different naming conventions. Querying such inconsistent data is problematic because the traditional query paradigm based on exact matching in equality comparisons is meaningless. As a result, users are unable to find desired information in the database; even worse, the same information can be reinserted into the database with a different representation, thereby leading to even more inconsistency.

A better approach would be to apply the more general paradigm of *similarity matching*, where strings are matched in terms of their similarity. A *similarity function* is used to quantify the underlying notion of similarity and two strings are considered similar if the value returned by the similarity function for them is not less than a threshold. Considering that tuples are represented by a descriptive string attribute, a *similarity selection* returns tuples that are similar to a given query string [Koudas et al. 2007], whereas a *similarity join* returns pairs of similar tuples from two input tables [Gravano et al. 2001]. Such operations are instrumental in dealing with inconsistent string data. For example,

We thank the anonymous reviewers for their helpful comments. This research was partially supported by FAPEMIG, FAPESC, CNPq, and INCT INCoD.

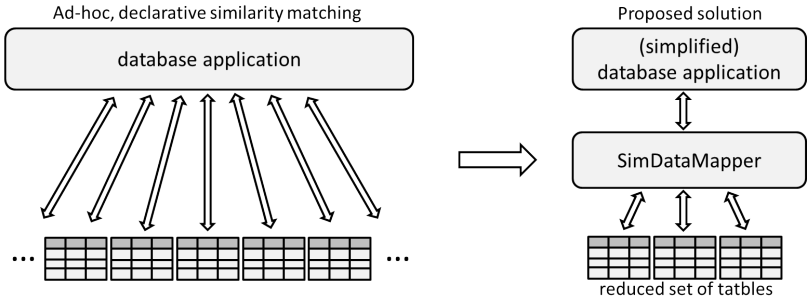


Fig. 1. Proposed software architecture for declarative similarity matching.

we can use similarity selections to find misspelled variants of patient names in a medical database and, likewise, similarity joins to identify tuples from two tables referring to the same patient.

Unfortunately, similarity matching is not directly supported by conventional Relational Database Management Systems (RDBMSs). The direct approach of implementing similarity functions as user defined functions (UDFs) can be very expensive. In similarity selections, the UDF would be evaluated over all tuples of the input table, even those bearing no similarity to the query string. In similarity joins, the situation is much worse as the UDF would be applied over the cross-product of two input tables. Over the last years, several proposals addressed this problem by expressing similarity functions declaratively in SQL [Gravano et al. 2001; Gravano et al. 2003; Koudas et al. 2007]. Besides covering a wide range of notions of similarity, these works employ a number of filters and index tables to reduce the comparison space, thus achieving significant performance gains.

However, the above approaches are largely ad-hoc, which complicates their integration with application programs. For example, enabling similarity operations over a string field on the application side requires creation and maintenance of several auxiliary tables on the database side. As a direct consequence, the well-known "impedance mismatch" problem [Cook and Ibrahim 2006], i.e., the problem caused by differences between programming language abstractions and persistent storage, is markedly exacerbated. The problem is much more severe when using similarity functions based on statistics, because these statistics need to be propagated to the indexes after updates on the original data [Koudas et al. 2007]. Moreover, similarity-aware database applications are likely to demand support of a variety of similarity functions. Such applications are thus faced with the daunting task of managing a huge number of indexes for each string attribute considered. We believe that these issues significantly hinder the widespread adoption of declarative similarity matching by applications beyond those for ad-hoc analysis over read-only data.

In this article, we present SimDataMapper, an architectural pattern to facilitate integration of declarative similarity matching into object-oriented database applications. SimDataMapper provides a general solution for loading in-memory objects based on similarity operations, while completely isolating the application from all details of the underlying similarity matching processing. Figure 1 illustrates our proposal. Instead of directly maintaining a number of auxiliary tables, a similarity-aware database application can delegate this task to the SimDataMapper. While our solution is based on auxiliary tables as previous work, those tables are now hidden from application programs, which can thus be greatly simplified. Furthermore, previous work used a different set of auxiliary tables for each similarity function under consideration. In contrast, SimDataMapper uniformly supports a variety of similarity functions using a common set of tables. In this context, additional index tables can be flexibly created or dropped at any time based on the query/update ratio in the current workload. We present an implementation of the SimDataMapper, which largely based on pure SQL. As a result, we can flexibly express various notions of similarity in a declarative way and easily reuse this implementation across different RDBMS platforms. We experimentally evaluate the performance

of our proposal on real-world datasets obtained from a production database and a publicly available repository and compare it with alternative approaches.

This article is an extended and revised version of an earlier conference paper [Schneider et al. 2015]. As part of the new material, we include support to similarity functions on unweighted sets and declarative statements for updating auxiliary tables after modification on the source tables. Furthermore, we provide a comprehensive set of new experiments covering several aspects not discussed in [Schneider et al. 2015] and including a comparison with alternative approaches.

The rest of the article is organized as follows. In Section 2, we provide necessary background. In Section 3, we present our architectural pattern and describe its implementation details. Performance results are reported in Section 4. We discuss related work in Section 5, before we wrap up with the conclusions in Section 6.

2. PRELIMINARIES

2.1 Mapping Strings to Sets

We map strings to *sets of tokens* using the popular concept of *q-grams*, i.e., substrings of length q obtained by "sliding" a window over the characters of an input string s. We extend s by prefixing and suffixing it with $q-1$ occurrences of a special character *"$"* not appearing in any string. Thus, all characters of s participate in exact q q-grams. For example, the string *"token"* can be mapped to the set of *2*-gram tokens {*$t*, *to*, *ok*, *ke*, *en*, *n$*}. As the result can be a multi-set, we append the symbol of a sequential ordinal number to each occurrence of a token to convert multi-sets into sets, e.g, the multi-set {a,b,b} is converted to {*a∘1*, *b∘1*, *b∘2*}. Given a string s, we denote by $Q_q(s)$ its token set of substrings with length q.

We associate a weight with each token to obtain *weighted sets*. A widely adopted weighting scheme is the Inverse Document Frequency (*IDF*), which associates a weight $w(tk)$ with a token tk as follows: $w(tk)=ln(N/df(tk))$, where $df(tk)$ is the *document frequency*, i.e., the number of strings a token tk appears in a database of N strings. The intuition behind using IDF is that rare tokens are more discriminative and thus more important for similarity assessment. The weight of a set x, denoted by $w(x)$, is given by weight summation of its tokens, i.e., $w(x) = \sum_{tk \in x} w(tk)$. Note that we obtain *unweighted sets* by simply associating the value of 1 to each token; in such case, we have $w(x) = |x|$. For ease of notation, given a string s, we use simply $w(s)$ instead of $w(Q_q(s))$ to denote its weighted (or unweighted) token set. Similarly, $w(r \cup s)$ and $w(r \cap s)$ denotes, respectively, the union and overlap of the tokens sets derived from strings r and s. Finally, $|s|$ denotes the length of string s (i.e., number of characters).

2.2 Similarity Functions

We focus on two classes of similarity functions, namely, *token-based* and *edit-based* functions. The reasons for this choice are threefold: token-based and edit-distance similarity functions 1) have been extensively used in many database applications (e.g., [Chaudhuri et al. 2006; Koudas et al. 2007]), 2) can be declaratively expressed, and 3) share some common underlying principles that allow their support in a unified way.

The key idea behind token-based similarity functions is that most of the tokens derived from two significantly similar strings should agree accordingly. Thus, the token sets of these strings would have a large overlap, which, in turn, results in a high similarity value. Popular token-based similarity functions are defined as follows.

Definition 2.1. Let r and s be two strings. We have:

—*Jaccard similarity*: $J(r,s) = \frac{w(r \cap s)}{w(r \cup s)}$.
—*Dice similarity*: $D(r,s) = \frac{2 \times w(r \cap s)}{w(r)+w(s)}$.
—*Cosine similarity*: $C(r,s) = \frac{w(r \cap s)}{\sqrt{w(r) \times w(s)}}$.

Note that the above definitions as well as following definitions in Section 2.4 uniformly apply for unweighted and weighted sets. Accordingly, we refer to token-based similarity functions on (un)weighted sets as (un)weighted similarity functions. Such distinction is nevertheless omitted whenever unimportant for the discussion.

EXAMPLE 2.1. *Consider two strings r and s and their corresponding weighted token sets*

$w(r) = \{\langle \mathbf{c},\ \mathbf{9}\rangle, \langle \mathbf{b},\ \mathbf{7}\rangle, \langle \mathbf{d},\ \mathbf{7}\rangle, \langle \mathbf{e},\ \mathbf{7}\rangle, \langle \mathbf{h},\ \mathbf{5}\rangle, \langle \mathbf{i},\ \mathbf{5}\rangle\}$ *and*

$w(s) = \{\langle \mathbf{a},\ \mathbf{10}\rangle, \langle \mathbf{c},\ \mathbf{9}\rangle, \langle \mathbf{b},\ \mathbf{7}\rangle, \langle \mathbf{d},\ \mathbf{7}\rangle, \langle \mathbf{e},\ \mathbf{7}\rangle, \langle \mathbf{f},\ \mathbf{5}\rangle, \langle \mathbf{h},\ \mathbf{5}\rangle\}$.

Note the token-weight association $\langle tk, w(tk)\rangle$, where tokens are represented by single characters. Then, we have $J(r,s) = \frac{35}{40+50-35} \approx 0.64$; $D(r,s) = 2 \times \frac{35}{40+50} \approx 0.78$; and $C(r,s) = \frac{35}{\sqrt{40 \times 50}} \approx 0.79$.

The dual notion of similarity is *distance* that quantifies the degree to which two entities are "different" or "far away". A popular distance measure is the *edit distance*, which is defined by the minimum number of character-editing operations to make two strings equal [Navarro 2001]. Often, similarity and distance values can be easily converted into one another.

Definition 2.2. Let r and s be two strings. We have:

—*Edit distance*: $ED(r,s)$ is the least number of operations (i.e, character insertion, deletion, and substitution) to transform r into s.
—*Edit similarity*: $ES(r,s) = 1 - \frac{ED(r,s)}{max(|r|,|s|)}$.

EXAMPLE 2.2. *Consider the strings r ="toen" and s ="token". Then, we have $ED(r,s) = 1$, obtained by inserting the character "k" into r, and $ES(r,s) = 1 - \frac{1}{5} = 0.8$.*

2.3 Similarity Selection and Join

We are now ready to define similarity selection and join queries.

Definition 2.3 *Similarity Selection.* Given a table T with a string attribute $T.A$, a query string s, a similarity function Sim, and a threshold τ, retrieve all tuples $t \in T$ such that $Sim(t.A, s) \geq \tau$.

Definition 2.4 *Similarity Join.* Given two tables T_1 and T_2 with string attributes $T_1.A_i$ and $T_2.A_j$, respectively, a similarity function Sim, and a threshold τ, retrieve all pairs tuples $\langle t_1, t_2\rangle \in T_1 \times T_2$ such that $Sim(t_1.A_i, t_2.A_j) \geq \tau$.

The Similarity Selection and Join can be easily expressed in SQL using UDFs. For example, consider a UDF $Sim(r,s,\tau)$ implementing a similarity function and returning true if the similarity of the string arguments r and s according to the similarity function is not less than τ. Thus, given a query string s, a Similarity Selection can be declaratively performed as follows (the adaption of the query below to Similarity Join is obvious):

```
SELECT T.tid
FROM T
WHERE Sim(T.A, s, τ)
```

Table I. Overlap and size bounds.

Function	$O(r,s)$	$[min(r), max(r)]$
Jaccard	$\frac{\tau \times (w(r)+w(s))}{1+\tau}$	$\left[\tau \times w(r), \frac{w(r)}{\tau}\right]$
Dice	$\frac{\tau \times (w(r)+w(s))}{2}$	$\left[\frac{\tau \times w(r)}{2-\tau}, \frac{(2-\tau)\times w(r)}{\tau}\right]$
Cosine	$\tau \times \sqrt{w(r) \times w(s)}$	$\left[\tau^2 \times w(r), \frac{w(r)}{\tau^2}\right]$
ED	$max(\lvert r\rvert, \lvert s\rvert) - 1 - (\tau - 1) \times q$	$[\lvert r\rvert - \tau, \lvert r\rvert + \tau]$
ES	$max(\lvert r\rvert, \lvert s\rvert) - 1 - ((1-\tau) \times max(\lvert r\rvert, \lvert s\rvert)) \times q$	$[\lvert r\rvert \times \tau, \lvert r\rvert \times (2-\tau)]$

2.4 Filtering Techniques

A naïve approach, solely based on UDFs, would evaluate the similarity operations in the previous section by computing the similarity function for all input tuples. Obviously, the computational cost of this strategy is prohibitively high for large tables, particularly for joins where the time complexity would be quadratic in the size of the input tables. Next, we present filtering techniques that can substantially reduce the comparison space.

2.4.1 *Overlap and Size Bounds.* All similarity functions considered are closely related to the overlap measure between the token sets derived from the two input strings. For token-based similarity functions, such connection is straightforward: predicates involving these functions can often be equivalently represented in terms of an *overlap bound* [Chaudhuri et al. 2006]. Formally, the overlap bound between token sets derived from strings r and s, denoted by $O(r,s)$, is a function that maps a threshold τ and the token set weights to a real value, s.t. $sim(r,s) \geq \tau$ iff $w(r \cap s) \geq O(r,s)$[1].

Similarity operations using token-based functions can be reduced to the problem of identifying pairs r and s whose overlap of the corresponding token sets is not less than $O(r,s)$. For edit-based functions, we can employ overlap bound to prune string pairs that cannot satisfy the similarity predicate, since it can be shown that if $ED(r,s) \leq \tau$, then $Q_q(r) \cap Q_q(s) \geq O(r,s)$ [Gravano et al. 2001]. Table I shows the overlap bounds of the previous functions.

Further, similar strings have, in general, roughly similar sizes. (Here, we loosely use the term size to refer to both token set weights and string length; the former is used on token-based functions, whereas the latter on edit-based ones.) We can derive bounds for immediate pruning of candidate pairs whose sizes differ enough [Sarawagi and Kirpal 2004]. Formally, the size bounds of r, denoted by $min(r)$ and $max(r)$, are functions that map τ and $w(r)$ or $|r|$ to a real value s.t. $\forall s$, where s is a string, if $sim(r,s) \geq \tau$, then $min(r) \leq w(s) \leq max(r)$.

Given a string r, we can safely ignore all strings whose sizes do not fall within the interval $[min(r), max(r)]$. Table I shows size bounds of all similarity functions considered in this article.

2.4.2 *Prefix Filtering.* We can prune a large share of the comparison space by exploiting the *prefix filtering principle* [Sarawagi and Kirpal 2004; Chaudhuri et al. 2006]. Prefixes allow selecting or discarding candidate pairs by examining only a fraction of the original sets. First, fix a global ordering $\mathcal{O}$; tokens of all sets are then sorted according to this ordering. A set $x' \subseteq x$ is a prefix of x if x' contains the first $|x'|$ tokens of x. Further, $pref_\beta(x)$ is the shortest prefix of x, the weights of whose tokens add up to more than β. The prefix filtering principle is formally defined as follows.

[1]For ease of notation, the parameter τ is omitted in the functions defined in this section.

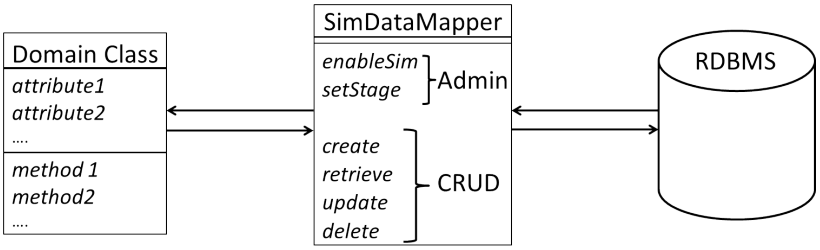


Fig. 2. The SimDataMapper architectural pattern.

Definition 2.5 *Prefix Filtering Principle.* Let r and s be two strings and α be an overlap threshold. If $w(r \cap s) \geq \alpha$, then $pref_{\beta_r}(r) \cap pref_{\beta_s}(r) \neq \varnothing$, where $\beta_r = w(r) - \alpha$ and $\beta_s = w(s) - \alpha$, respectively.

Note that prefix overlap is a condition necessary, but not sufficient to satisfy the original overlap constraint: an additional verification must be performed on the candidate pairs. Finally, the number of candidates can be reduced by using *document frequency ordering*, $\mathcal{O}_{df}$, as global token order to obtain sets ordered by increasing token frequency in the database. The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by shifting lower frequency tokens to the prefix positions.

Example 2.3. *Consider again the weighted sets $w(r)$ and $w(s)$ in Example 2.1 and assume that both sets are already sorted. For the Jaccard similarity and $\tau = 0.6$, we have $O(r,s) = 33.75$, $[min(r), max(r)] = [24, 66.7]$, $[min(s), max(s)] = [30, 83.3]$, $pref_{\beta_r}(r) = \langle$ `c, 9`$\rangle$ and $pref_{\beta_s}(s) = \{\langle$ `a, 10`$\rangle, \langle$ `c, 9`$\rangle\}$, for $\beta_r = \beta_s = O(r,s)$.*

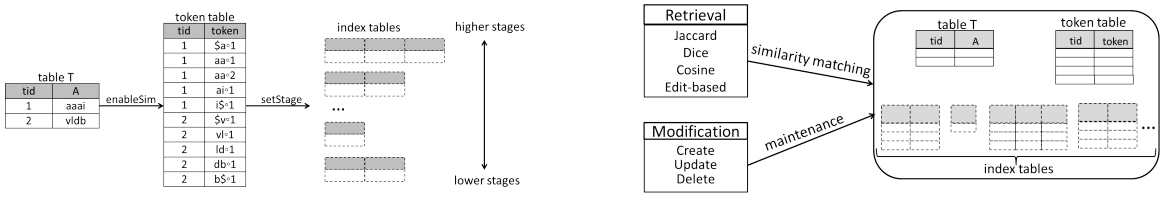
3. THE SIMDATAMAPPER PATTERN

3.1 General Overview

The SimDataMapper pattern is essentially a specialization of the well-known *Data Mapper* pattern [Fowler 2003] and, as such, inherits its fundamental features. Basically, it is a layer of software that insulates in-memory domain objects and the database from each other and handles the data transfer between them. SimDataMapper extends the Data Mapper pattern with the support of retrieval operations based on similarity matching on string attributes, while transparently handling modification operations on these attributes. Figure 2 illustrates the SimDataMapper pattern and its interfaces for Create, Retrieve, Update, and Delete (CRUD) operations and administrative tasks.

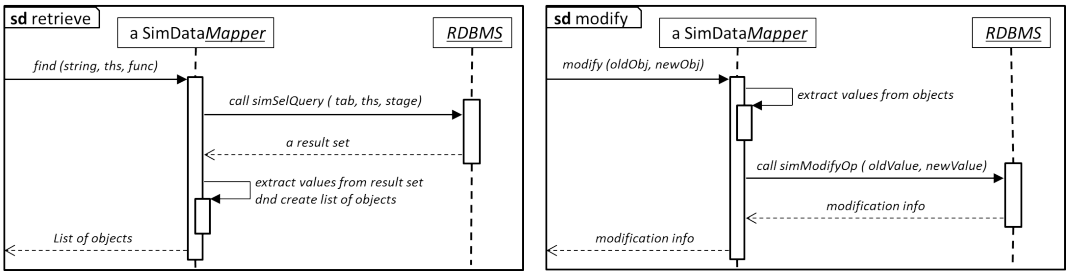
The method `enableSim` takes the specification of a string-valued attribute `T.A` of table `T` as input and creates a *token table* based on the values in this attribute; also, a number of index tables are created depending on the selected *processing stage* using the `setStage` method. Higher stages imply more index tables being used to support processing of similarity queries. The administrative operations are depicted in Figure 3(a). Afterwards, one or more application objects whose corresponding class is mapped to `T` can be loaded by invoking a special `find` method, which results in a similarity selection query against the corresponding auxiliary tables (i.e., token and index tables). Figure 3(b) illustrates the similarity operations within SimDataMapper over original and auxiliary tables.

Similarity matching requires adaption of all CRUD operations. For Retrieve operations, `find` methods take a query string s, a threshold, and a similarity function identifier as input. Figure 4(a) shows a sequence diagram capturing the behavior of such operations. Besides similarity selections, Retrieve also comprises similarity joins, in which case auxiliary tables for all attributes involved in the join condition must be available. Further, Update operations affecting $T.A$ as well as any Create and Delete operations translate into updates to the auxiliary tables. Figure 4(b) shows a general



(a) Administrative operations. (b) Similarity operations.

Fig. 3. Details of the SimDataMapper methods.



(a) Sequence diagram for retrieving. (b) Sequence diagram for modification.

Fig. 4. Sequence diagrams for CRUD operations based on similarity matching.

sequence diagram for those modification operations. In the `modify` method, either *oldObj* or *newObj* can be *null* for Create and Delete operations, respectively, or none of them for Update operations. Finally, SimDataMapper pattern acts as a regular Data Mapper for Retrieve operations based on exact matching or Update not involving any attribute considered for similarity matching.

3.2 Implementation Details

The integration of similarity matching functionality into a database mapping layer can be realized in different ways. In our implementation, procedural code is used to generate token sets from strings and calculate the edit distance between two strings. All other operations are performed using standard, declarative SQL statements.

3.2.1 *Creation of Auxiliary Tables.* For simplicity, consider a simple table `T(tid, A)`, where `tid` is the key attribute of `T` and `A` is a string-valued attribute. Given `T.A` as input, the method `enableSim` creates the token table `TTokens(tid, token)`, where for each tuple t of `T`, the q-grams in $Q_q(t.A)$ are represented as a separated tuple associated with $t.tid$ (recall Figure 3(a)) — the population of `TTokens` can be accelerated by using bulk loading functionality available in virtually all RDBMSs. The token table represents a "common ground" to the declarative specification of all similarity functions considered in Section 2.2. The storage overhead of `TTokens` is moderated for reasonable values of q [Gravano et al. 2001]. Formally, the storage size of `TTokens`, denoted by $S(TTokens)$, is given by:

$$S(TTokens) = n(q-1)(q+C) + (q+C)\sum_{j=1}^{n} |t_j.A|,$$

where n is the number of tuples in `T`, q the q-gram size, and C the size of the attribute `T.tid`.

We can further identify a clear staged evaluation strategy for weighted token-based similarity functions, which lends itself to the definition of index tables to improve query performance. Figure 5

Stage 1

```
INSERT INTO TDF(token, df)
SELECT token, COUNT(*)
FROM TTokens
GROUP BY token
```

(a) Table storing *df* values for each token.

```
INSERT INTO TSize(N)
SELECT COUNT(*)
FROM T
```

(b) Table storing the number of tuples in T.

Stage 2

```
INSERT INTO TTokenDF(tid, token, df)
SELECT TT.tid, TT.token, TDF.df
FROM TTokens TT, TDF
WHERE TT.token=TDF.token
```

(c) Table storing token and *df* values for each tuple.

```
INSERT INTO TRawSetWeight(tid, s1, s2)
SELECT tid, COUNT(*), SUM(LOG(df))
FROM TTokenDF
GROUP BY tid
```

(d) Table storing terms for set weight calculation.

Stage 3

```
INSERT INTO TSetWeight(tid, sw)
SELECT TR.tid, (LOG(TS.N)*TR.s1-TR.s2)
FROM TRawSetWeight TR, TSize TS
```

(e) Table storing set weight values.

Stage 4

```
INSERT INTO TWeights(tid, token, sw)
SELECT TT.tid, TT.token, TW.sw
FROM TTokens TT, TSetWeight TW
WHERE TT.tid=TW.tid
```

(f) Table storing token and set weight values.

Fig. 5. SQL statements to enable similarity matching on table `T`.

illustrates several index tables organized into 4 stages—tables `T` and `TTokens` constitute the Stage 0. In this context, the tables at Stage 1 help to calculate token weights: `TDF(token, df)` (Figure 5(a)) associates each token with its *df* value, whereas `TSize(N)` (Figure 5(b)) is a dummy table storing the number of tuples of `T`. On the other extreme, we have table `TWeights(tid, token, sw)` at Stage 4 (Figure 5(f)), where `sw` represents the token set weight. We do not need to store token weights, because they are calculated at query processing time, as we will see shortly.

While the `TWeights` table speeds up query evaluation, its maintenance can slow down update queries. In fact, `TWeights` would need to be recreated anew after any insertion or deletion on `T` because the weight of token sets change with the number of tuples of `T`. To mitigate this problem, we follow the approach of Koudas et al. [2007], where new terms independent of the number tuples of the original table were defined for the vector length—the similarity function considered by Koudas et al. [2007] is the Cosine similarity in the vector space model. We adapt this strategy to our context by rewriting the token set weight formula as follows:

$$
\begin{aligned}
w(x) &= \sum_{tk \in x} w(tk) \\
&= \sum_{tk \in x} log(N) - log(df(tk)) \\
&= log(N) \sum_{tk \in x} 1 - \sum_{tk \in x} log(df(tk)) \\
&= log(N) \times |x| - \sum_{tk \in x} log(df(tk)).
\end{aligned}
$$

We can define Stage 2 containing two tables: `TTokenDF(tid, token, df)` (Figure 5(c)), which is simply the join result between `TTToken` and `TDF`, and `TRawSetWeight(tid, s1, s2)` (Figure 5(d)), where `s1` represents $|x|$ and `s2` represents $\sum_{tk \in x} log(df(tk))$ in the above formula; the attribute `s1` can also be used for evaluation of unweighted similarity functions. From Stage 2, we can easily define Stage 3 containing the table `TRawSetWeight(tid, sw)` (Figure 5(e)), which associates each `tid` with the corresponding token set weight. We end up with a variety of alternative processing stages: higher stages favor query processing, whereas lower stages favor updates. Applications can flexibly specify the best stage to the query-update ratio of an anticipated workload using the `setStage` method. The storage overhead of index table `TDF` is within a constant factor of the number of distinct tokens (`TSize` is negligible) and the index tables in Stages 2–4 are within a constant factor of either $S(TTokens)$ (`TTokenDF` and `TWeights`) or the number of tuples in `T` (`TRawSetWeight` and `TSetWeight`).

```
INSERT INTO SCards(tid, token, sc)
SELECT   ST.tid, ST.token, SC.card
FROM     STokens ST, (SELECT COUNT(*) card FROM Stokens) SC
```
(a) Search table population for unweighted similarity functions.

```
SELECT   TT.tid, SC.tid
FROM     TTokens TT, TRawSetWeight TR, Scards SC
WHERE    TT.token=SC.token AND TT.tid=TR.tid AND
         TR.s1 >=(SC.card*τ) AND TR.s1 <=(SC.card/τ)
GROUP BY TT.tid, SC.tid, TR.s1, SC.card
HAVING   COUNT(*) ≥ (τ/(1+τ))*(TR.s1 + SC.card)
```
(b) Query for unweighted Jaccard.

```
INSERT INTO STokenIDF(tid, token, tw)
SELECT   ST.tid, ST.token, LOG(TS.N/TDF.df)
FROM     STokens ST, TTokenDF TDF, TSize TS
WHERE    ST.token = TDF.token
UNION ALL
SELECT   ST.tid, ST.token, LOG(TS.N)
FROM     STokens ST, TSize TS
WHERE    ST.token NOT IN (SELECT token FROM TTokenDF)

INSERT INTO Sweights(tid, token, tw, sw)
SELECT   SI.tid, SI.token, SI.tw, SW.sw
FROM     STokenIDF SI, (SELECT SUM(tw)sw FROM STokenIDF) SW
```
(c) Search table population for weighted similarity functions.

```
SELECT   TT.tid, S.tid
FROM     TTokens TT, TRawSetWeight TR, TSize TS, SWeights S
WHERE    TT.token=S.token AND TT.tid=TR.tid AND
         (LOG(TS.N)*TR.s1-TR.s2)>=(S.sw*τ) AND
         (LOG(TS.N)*TR.s1-TR.s2)<=(S.sw/τ)
GROUP BY TT.tid, S.tid, TR.s1, TR.s2, TS.N, S.sw
HAVING   SUM(S.tw)≥(τ/(1+τ))*((LOG(TS.N)*TR.s1-TR.s2)+S.sw)
```
(d) Query for weighted Jaccard at Stage 2.

```
SELECT   TD.tid, S.tid
FROM     TTokens TT, TSetWeight TW, SWeights S
WHERE    TT.token=S.token AND TT.tid=TW.tid AND
         TW.sw>=(S.sw*τ) AND TW.sw<=(S.sw /τ)
GROUP BY TT.tid, S.tid, TW.sw, S.sw
HAVING   SUM(S.sw)>=(τ/(1+τ))*(TW.sw+S.sw)
```
(e) Query for weighted Jaccard at Stage 3.

```
SELECT   T.tid, S.tid
FROM     TWeights T, SWeights S
WHERE    T.token = S.token AND T.sw>=(S.sw*τ) AND
         T.sw<=(S.sw/τ)
GROUP BY T.tid, S.tid, T.sw, S.sw
HAVING   SUM(T.tw)≥(τ/(1+τ))*(T.sw+S.sw)
```
(f) Query for weighted Jaccard at Stage 4.

```
SELECT   T.tid, S.tid
FROM     T, TTokens TT, S, STokens ST
WHERE    TT.token=ST.token AND T.tid = TT.tid AND
         S.tid=ST.tid AND ABS(LEN(T.A)-LEN(S.A))<=τ
GROUP BY T.tid, T.A, S.tid, S.A
HAVING   COUNT(*)>=LEN(T.A)-1-(τ-1)*q AND
         COUNT(*)>=LEN(S.A)-1-(τ-1)*q AND ED(T.A, S.A, τ)
```
(g) Query for edit distance.

Fig. 6. SQL statements for similarity selection.

3.2.2 *Similarity Selection Evaluation.* Similarity selections are evaluated by first populating the table *Stokens* with the token set generated from the query string. *Stokens* is then used to create a *search table* that is matched against the auxiliary tables derived from table `T`. For unweighted similarity functions, the search table is `SCards(sid, token, sc)`, where the attribute `sc` stores the cardinality of *Stokens*. Figure 6(a) shows the SQL query to populate `Scards`. The SQL query for unweighted Jaccard is shown in Figure 6(b). Size bounds are checked in the `WHERE` clause and overlap bounds in the `HAVING` clause. Adaption to other similarity functions is straightforward: one basically needs to change the filters in the `WHERE` and `HAVING` clauses to the corresponding definitions shown in Table I.

For weighted similarity similarity functions, the search table `SWeights(tid, token, tw, sw)` is created from the query string, where `tw` and `sw` represent the token weight and the token set weight, respectively. Figure 6(c) shows the SQL query to populate SWeights. The intermediate table `STokenIDF` is used to store the IDF weights of each token (the explicit creation of `STokenIDF` can be avoided by using *common table expressions*). In the SQL expression for populating `STokenIDF`, tokens in the query string not appearing in `TTokens` are treated as rare tokens with a df value of 1. This strategy results in a decreased similarity value, because these tokens receive maximum IDF weight and will not match with any token in the auxiliary tables. Of course, other strategies are possible, such as assigning an average df value to these tokens. Figure 6(d) shows the SQL query for weighted Jaccard at Stage 2. Queries for Stages 3 and 4 are shown in Figures 6(e) and 6(f), respectively; the main difference from Stage 2 is the simplified calculation of token and set weights. Finally, Figure 6(g) shows the SQL query for edit distance. The tables involved are the original tables and the token tables — note that the query string is represented by table `S`. Again, size and overlap bounds are checked in the `WHERE` and `HAVING` clauses, respectively. The UDF *ED* is finally applied on the remaining string pairs that survived the filter steps.

Prefix table generation

```
INSERT INTO T1Prefix (tid, token)
SELECT      tid, token
FROM        (SELECT TT.tid tid, TT.token token, TR.s1 sc (RANK() OVER
             (PARTITION BY TT.tid ORDER BY TD.df  ASC, TT.token)) as p
             FROM T1Tokens TT, T1T2DF TD, T1RawSetWeight TR
             WHERE TT.tid = TFD.tid AND TT.tid = TR.tid) as P
WHERE       p <= sc - CEIL(sc * τ) + 1
```

(a) Table storing unweighted Jaccard prefixes.

```
INSERT INTO T1Prefix (tid, token)
SELECT      tid, token
FROM        (SELECT tid, token, sw, (SUM(tw) OVER
             (PARTITION BY tid ORDER BY tw DESC, token)-tw) p
             FROM T1JoinWeights) P
WHERE       p >= sw-(sw * τ)
```

(b) Table storing weighted Jaccard prefixes.

```
INSERT INTO T1Prefix (tid, token)
SELECT      tid, token
FROM        (SELECT TT.tid tid, TT.token token, (RANK() OVER
             (PARTITION BY TT.tid ORDER BY TD.df  ASC, TT.token)) p
             FROM T1Tokens TT, T1T2DF TD WHERE TT.tid = TFD.tid) P
WHERE       p <= τ*q + 1
```

(c) Table storing edit distance prefixes.

Summation table generation

```
INSERT INTO T1T2DF(tid, token)
SELECT      D.tid, SUM(D.df)
FROM        ((SELECT * FROM T1DF)
            UNION ALL
            (SELECT * FROM T2DF)) as D
GROUP BY    D.token
```

(d) Table storing aggregated *df* values from two input tables.

```
INSERT INTO T1T2Size(N)
SELECT      TS1.N + TS2.N
FROM        T1Size TS1, T2Size TS2
```

(e) Table storing the total number of tuples in the two input tables .

Candidate generation

```
CREATE VIEW      CAND AS
SELECT DISTINCT  P1.tid, P2.tid
FROM             T1Prefix P1, T2Prefix P2
WHERE            P1.token=P2.token
```

(f) View for candidate generation.

Fig. 7. SQL statements for similarity join.

3.2.3 *Similarity Join Evaluation.* In principle, similarity joins can be implemented using similar queries as those for selections. However, only the filters based on size and overlap bounds are insufficient to deal with the high cost of joining two large tables. Thus, a more aggressive pruning approach based on prefix filtering is needed. To this end, the following sequence of operations is executed: 1) creation of *summation tables*; 2) creation of new auxiliary tables (if joining weighted sets); 3) creation of *prefix tables*; 4) generation of candidates; 5) similarity join evaluation over the candidates.

Summation tables store aggregated statistics from the two input tables. Table `T1T2DF` associates all tokens in tables `T1` and `T2` with the corresponding `df` value; for tokens appearing in both tables, the summation of each `df` value is stored (Figure 7(d)). For unweighted similarity functions, the new `df` values determine the global token order for prefix creation. Note that we do not to create summation tables for self-joins. For weighted similarity functions, we also need to derive new IDF weights for each token and new set weights accordingly. First, the `T1T2Size` table is created, which stores the sum of the input table sizes ((Figure 7(d)). Then, texttttT1T2Size together with `T1T2DF` are used to create the table `TiJoinWeights(tid, token, tw, sw)` for each input table. The queries for populating `TiJoinWeights` are similar to the queries used to create the `TWeights` table (Figure 6).

The next step is the creation of the prefix tables storing only tokens in the prefix of each token set. This step can be implemented in pure SQL using *window functions*. The SQL statements to create prefix tables for unweighted Jaccard, weighted Jaccard, and edit distance, are respectively shown in Figures 7(a-c). Previous implementations of prefix filtering in the RDBMS were either procedural [Chaudhuri et al. 2006] or did not consider token weights [Xiao et al. 2011; Augsten et al. 2014].

After having created the prefix tables, pairs of tuples whose associated prefixes have at least a token in common can be identified by a simple equi-join. This operation is illustrated by a simple view definition in Figure 7(f). The similarity join is then performed using queries similar to those used for selections (Figure 6), with additional join with the *CAND* view. All tables created for join processing are deleted upon completion of the operation. Alternatively, if there is no intervening data update, the `TiJoinWeights` table can be reused on subsequent similarity join evaluations; the prefix tables can also be reused in such case for joins with the same or lower threshold values.

```
--Stage 0
INSERT INTO Ttokens (SELECT new_tid, token FROM NewTokens);
--Stage 1
UPDATE TDF SET df = df + 1
WHERE token IN (SELECT token FROM NewTokens);

INSERT INTO TDF
SELECT token, 1 FROM NewTokens
WHERE token NOT IN (SELECT token FROM TDF);

UPDATE Tsize SET N = N + 1;
--Stage 2
INSERT INTO AffectedTids
SELECT tid FROM TTokenDF
WHERE token IN (SELECT token FROM NewTokens);

UPDATE TTokenDF SET df = df + 1
WHERE tid IN (SELECT token FROM AffectedTids);

UPDATE TRawSetWeight TR
SET s2=(SELECT SUM(LOG(df)) FROM TTokenDF TD  WHERE TR.tid=TD.tid)
WHERE tid IN (SELECT token FROM AffectedTids);

INSERT INTO TTokenDF
SELECT new_tid, token, df FROM TDF NATURAL JOIN NewTokens;

INSERT INTO TRawSetWeigt
SELECT tid, COUNT(*), SUM(LOG(df)) FROM TTokenDF TD
WHERE tid = new_tid GROUP BY tid;
```

(a) Update of the auxiliary tables at Stage 2 after insertion

```
--Stage 0
DELETE FROM Ttokens WHERE tid = old_tid;
--Stage 1
UPDATE TDF SET df = df - 1
WHERE token IN (SELECT token FROM OldTokens);

DELETE FROM TDF WHERE df = 0;

UPDATE TSize SET N = N - 1;
--Stage 2
DELETE FROM TTokenDF WHERE tid = old_tid;

DELETE FROM TRawSetWeight WHERE tid = old_tid;

INSERT INTO AffectedTids
SELECT tid FROM TTokenDF
WHERE token IN (SELECT token FROM OldTokens);

UPDATE TTokenDF SET df = df - 1
WHERE tid IN (SELECT tid FROM AffectedTids);

UPDATE TRawSetWeight TR
SET s2=(SELECT SUM(LOG(df)) FROM TTokenDF TD
        WHERE TR.tid=TD.tid)
WHERE tid IN (SELECT tid FROM AffectedTids);
```

(b) Update of the auxiliary tables at Stage 2 after deletion

Fig. 8. SQL statements for updating the auxiliary tables at Stage 2 after modification on the source table.

3.2.4 *Auxiliary Table Updating.* Figures 8(a) and 8(b) show the sequence of SQL statements for updating the auxiliary tables at Stage 2 after Create and Delete operations, respectively. The statements can be encapsulated into a stored procedure and easily extended to handle multiple insertions and deletions. The token set generated from the inserted (deleted) string in the SimDataMapper is stored in the temporary table `NewTokens` (`OldTokens`). Alternatively, an array can be used to directly send the token set to the DBMS. Because insertion and deletion alter the value of `TSize.N`, the auxiliary tables at Stage 3 and Stage 4 have to be created anew after these operations. It is possible to trade accuracy for efficiency on modification operations by using techniques such as *blocking* and *thresholding* to postpone the propagation of updates [Koudas et al. 2007]. The Update operation is implemented using a combination of the SQL statements for insertion and deletion with the following differences. The temporary tables `NewTokens` and `OldTokens` store the difference between the token sets generated from the new and old string values. No tuple needs to be inserted into or deleted from table `TRawSetToken`. Finally, because the value of `TSize.N` is not modified, we do not need to recreate the auxiliary tables at Stage 3 and Stage 4: only tuples related to tokens in `NewTokens` and `OldTokens` need to be updated (the ids of these tuples are stored in the temporary table `AffectedTids`.

4. EXPERIMENTS

4.1 Experimental Setup

We used two real world datasets: *Medicine* containing medicine names, which were obtained from a production medical database and *DBLP*, containing author names of computer science publications [2]. We initially extracted 20K strings from each dataset. The average string length is 10 characters for *Medicine* and 14 for *DBLP*; owing to naming variations and misspellings, a large number of strings in *Medicine* have similar content referring to a same medicine. We then doubled each dataset to 40K

[2]dblp.uni-trier.de/xml. Accessed January 19, 2016.

by producing an erroneous copy of each string, i.e., a copy to which we applied 1–3 character-level transformations (insertion, deletion, and substitution).

All datasets were stored in relational tables with schema composed of an integer identifier and the string attribute. We built B-tree indexes on the identifier column of the source and the auxiliary tables (Figure 5), but not on the temporary tables created for join processing (Figure 7). Index construction time is not included in the results. We generated q-grams of size 3, which were hashed into four byte values before being stored in the relational tables. We evaluated selection, join, and modification operations, the latter formed by insertion and deletion operations. We generated the query workload for selections and modifications by randomly selecting strings from the datasets — note that the query workload contains both original strings and erroneous copies. We also used strings not present in any dataset for insertions or to produce empty results for selections. Example of strings in the workload for the DBLP dataset are "Robert Sedgewick", "Gotz Greafe", and "no matching author". For joins, we generated another dataset containing only erroneous copies. We report the RDBMS processing cost measured in average runtime over repeated executions. The time required to process application objects was not considered in the measurements (of course, this time can significantly vary along different applications). The default threshold and distance values are 0.7 and 1, respectively.

We ran our experiments on an Intel Xeon E3-1240 Quad-core, 3,3 GHz, 8MB CPU cache, and 8 GB of main memory. We used PostgreSQL 9.4 as RDBMS platform, with 2GB of main memory allocated to shared buffers and 64MB allocated to each internal operation such as sorting and hashing. SimDataMapper were implemented using Java JDK 8 (Oracle) and JDBC Driver version 4.

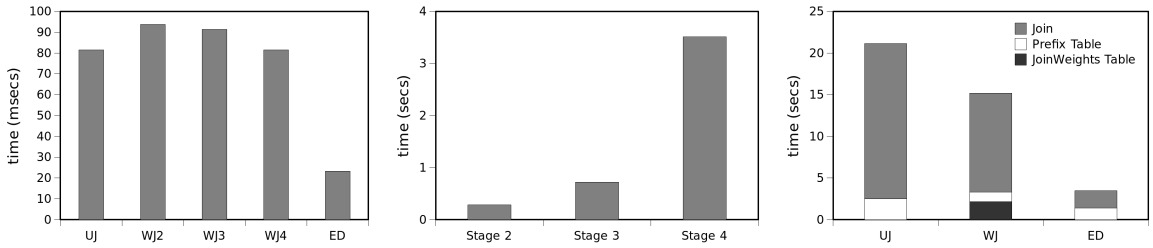
We compared our approach with two specialized implementations of string similarity matching present in additional PostgreSQL modules, namely, *fuzzystrmatch*[3] and *pg_ trmg* [4]. The *fuzzystrmatch* module provides an optimized edit distance implementation, which stops earlier as soon as is identified that the two input strings cannot meet the distance constraint. We also used this implementation as the *ED* UDF in the query statements for edit distance (see Figure 6(g)). The *pg_ trmg* module provides an implementation of the Jaccard similarity based on *3*-grams (trigrams in their nomenclature). In contrast to the method described in Section 2.1, *pg_ trmg* eliminates repeated tokens, which results in smaller token sets — on *DBLP*, the average token set sizes produced by SimDataMapper and *pg_ trmg* are 16,17 and 14,72, respectively. Also, this method actually leads to a different notion of similarity. For example, all strings matching the regular expression "aaa*" are considered as equal. The *pg_ trmg* module can be combined with inverted indexes for efficient processing. Here, we used GIN indexes[5], which favors query performance over updates and building time.

4.2 Performance Results for CRUD Operations

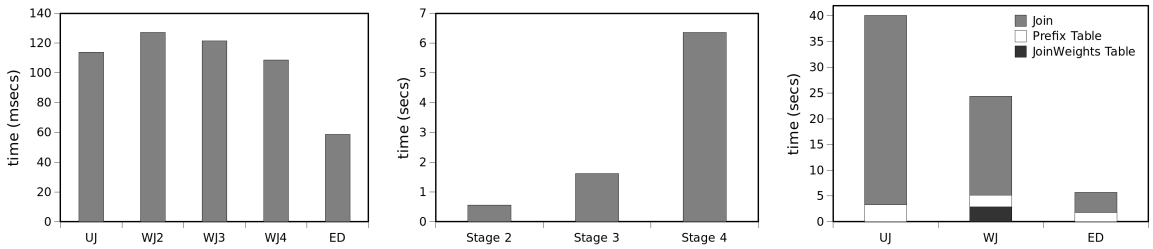
We first report performance results of SimDataMapper operations only; comparison with the alternative approaches is presented in the next section. In the experimental charts, unweighted Jaccard and edit distance are abbreviated to *UJ* and *ED*, respectively. Weighted Jaccard includes the indication of the processing stage; for example, *WJ2* is the abbreviation to weighted Jaccard at Stage 2.

Figure 9 shows results for CRUD operations based on Jaccard and edit distance on both datasets. All similarity selection queries took less than 100 milliseconds to complete on the *Medical* dataset (Figure 9(a)), thereby providing truly interactive response times. As expected, performance increases with the processing stage for weighted Jaccard. *ED* is faster than Jaccard, despite the higher cost of the edit distance calculation, which indicates better effectiveness of the filters based on overlap and size bounds. For modifications (Figure 9(b)), the results are just the opposite for weighted Jaccard: now performance decreases at higher stages. Modification timing at Stage 4 is markedly higher owing to

[3]https://www.postgresql.org/docs/9.4/static/fuzzystrmatch.html. Accessed May 13, 2016.
[4]https://www.postgresql.org/docs/9.4/static/pgtrgm.html. Accessed May 13, 2016.
[5]https://www.postgresql.org/docs/9.4/static/gin-intro.html. Accessed May 13, 2016.

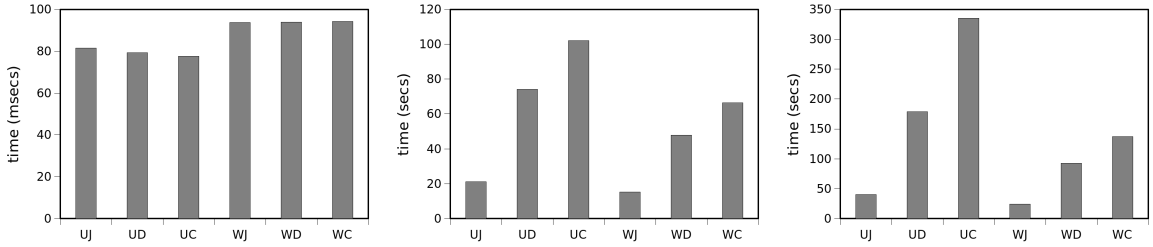


(a) Selection, Medicine dataset. (b) Modification, Medicine dataset. (c) Join, Medicine dataset.



(d) Selection, DBLP dataset. (e) Modification, DBLP dataset. (f) Join, DBLP dataset.

Fig. 9. Query and update results for Jaccard and Edit Distance on Medical and DBLP datasets.



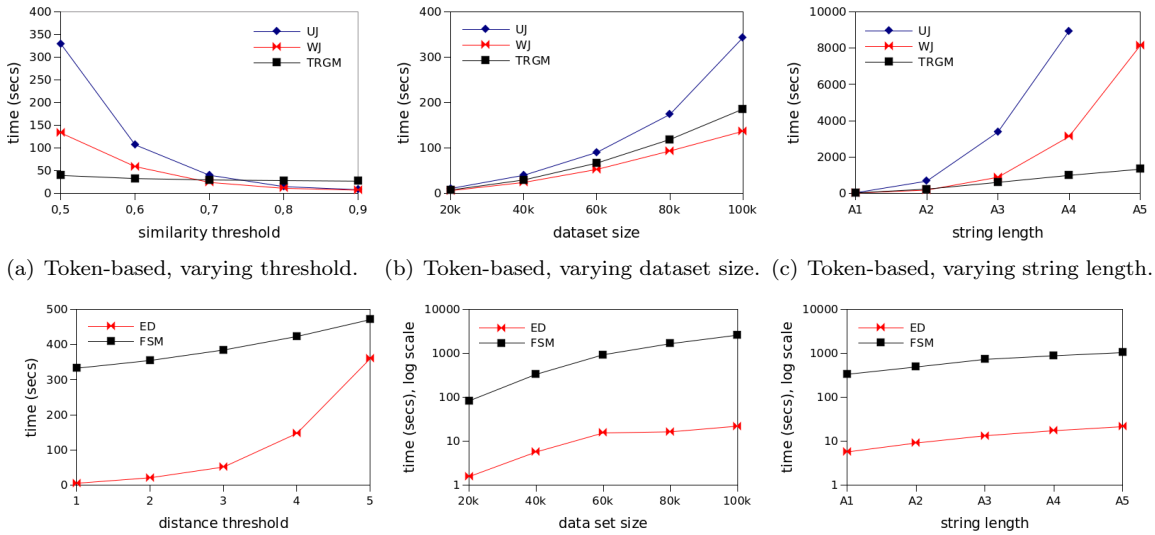
(a) Selection, Medical dataset. (b) Join, Medical dataset. (c) Join, DBLP dataset.

Fig. 10. Query results for different token-based similarity functions on Medical and DBLP datasets.

the costly recreation of the `TWeights` table. Figure 9(c) shows the results for similarity joins. The time of the actual join largely dominates the overall execution time. WJ is faster than UJ because prefixes based on unweighted sets typically exhibit worse filtering effectiveness as compared to weighted ones [Ribeiro and Härder 2011]. As for selections, ED is the best performing join operation. Figures 9(d)–(f) show the results on the *DBLP* dataset. The trends are similar to those observed on the *Medical* dataset; the timings are overall higher because strings in *DBLP* are larger in average size.

We note that the above results are significantly better than those previously presented by Schneider et al. [2015]. We had reimplemented all algorithms and adjusted configuration settings, which led to much better query execution plans. As a result, we obtained an order of magnitude (or more) better performance for most operations. The main refinements have already been mentioned: simplification of query statements (e.g., see Figure 6(f)), use of B-tree indexes, integer representation of q-grams, adjusted memory settings, and optimized edit distance implementation.

Figures 10(a) and (a) plot the results for queries using different token-based similarity functions on the *Medical* dataset. Unweighted (weighted) variants of Dice and Cosine are abbreviated to UD (WD) and UC (WC), respectively. We considered only the weighted Jaccard (WJ) execution on Stage 2. Differences are more prominent in the join results. Again, weighted variants perform better owing to higher prefix filtering effectiveness on weighted sets. Similarly, Jaccard is faster than Dice and Cosine



(a) Token-based, varying threshold. (b) Token-based, varying dataset size. (c) Token-based, varying string length.

(d) Edit-based, varying threshold. (e) Edit-based, varying dataset size. (f) Edit-based, varying string length.

Fig. 11. Join results: varying threshold values, dataset sizes, and string lengths.

unweighted and weighted sets because its overlap and size bounds are tighter, which results in better pruning power. Join results on *DBLP* exhibit similar trend (Figure 10(c)).

4.3 Comparison with Alternative Approaches and Scalability Experiments

We now compare SimDataMapper with implementations based on *pg_ trgm* and *fuzzystrmatch* (abbreviated to TRGM and FSM, respectively). We analyze performance with varying thresholds, dataset sizes and string lengths; for the two latter, we used the default similarity and distance thresholds of 0.7 and 1, respectively. We focus on the heavier join operation to better distinguish intrinsic characteristics of the algorithms under comparison. For TRGM, we built GIN indexes on both input tables and the optimizer always chose an Indexed Nested Loop Join using one of them; FSM joins were executed using regular Nested Loop Joins. In the charts, Ai indicates datasets containing strings formed by the concatenation of i author names. The resulting average string lengths of A2–A5 are 30.17, 45.94, 61.55, and 77,08, respectively.

We first present results for joins based on Jaccard similarity. Figure 11(a) plots results with varying threshold values. UJ and WJ are faster than TRGM at higher thresholds. The performance difference decreases with the threshold and TRGM becomes the fastest for thresholds ≤ 0.6. This result is expected because the filtering techniques used by UJ and WJ exploit high thresholds. TRGM is not base on such filters and, thus, its performance exhibits little variation across different threshold values. Figure 11(b) shows the results on increasing dataset size. WJ is the fastest on all datasets, whereas UJ is the slowest. Results on increasing string length are shown in Figure 11(c). WJ is still faster than TRGM on dataset A2, but slower on the following datasets. In fact, the performance of UJ and WJ severely degrades on larger strings; UJ took more than three hours to complete on A5 and we omit this result from the plot. The explanation lies in the underlying processing model based on relational, unnested set representation. In particular for larger strings, this model provides less scope for optimization as compared to the ones based on inverted indexes, thereby leading to slower processing — the performance gap between the two models can be substantially narrowed by using optimized merge join algorithms [Rae et al. 2014].

Our final set of experiments compares similarity joins based on edit distance. Figure 11(d) plots results with varying distance threshold. Our ED operator is faster than FSM on all distances. Fil-

tering is less effective at higher distance thresholds and the performance advantage of ED decreases accordingly. With low distance threshold, ED is two order of magnitude faster than FSM on increasing dataset sizes (Figure 11(e)) and string lengths (Figure 11(f)). Finally, we observe that both ED and FSM scales well with the string length owing to the optimized edit distance implementation, which avoids complete comparison between dissimilar strings.

5. RELATED WORK

In recent years, many different approaches have been proposed for string similarity matching, which fall into three broad, mostly complementary, categories: 1) *stand-alone algorithms*; 2) *integration with query engines*; and 3) *declarative matching on top of query engines*.

There is a large body of research on stand-alone, specialized algorithms for similarity selection and join [Sarawagi and Kirpal 2004; Xiao et al. 2011; Ribeiro and Härder 2011]. Numerous techniques have been proposed, including prefix- and size-based filters, specialized data structures based on inverted lists, and many others.

Several proposals to integrate string similarity matching into query engines focus on the design of new physical operators. The SSJoin operator is based on composing regular and similarity-specific operators into query evaluation plans [Chaudhuri et al. 2006]; SSJoin supports the same class of similarity functions considered in this article. The work of Augsten et al. [2014] defines a tokenizer operator to allow evaluation of similarity queries without precomputed tokens and indexes– the idea of using a tokenizer operator to enable on-the-fly evaluation first appeared in the context of XML DBMSs [Ribeiro and Härder 2007]. Tight coupling of such approaches with full-fledged RDBMSs requires adaptions on all query optimizer components, including algebraic operators, cost models, cardinality estimates, and index infrastructure. Thus, deep integration of similarity processing into general-purpose RDBMSs remains a long-term research goal.

Declarative similarity matching through SQL statements permits readily use of existing RDBMS machinery. Our work builds upon previous efforts in this area [Gravano et al. 2001; Gravano et al. 2003; Koudas et al. 2007]. But, instead of considering each similarity predicate in isolation, we focus on identifying key data structures and operations and assembling them into an architectural pattern to provide common support to a variety of similarity predicates.

In some sense, our work can be classified into a new category of approaches to string similarity matching: 4) *integration with object-oriented applications*. Popular products offer a data mapping layer to bridge object-oriented applications and relational databases, such as Hibernate and ADO.Net. Declarative similarity matching is not directly supported by these solutions. Frameworks such as Hibernate Search provide similarity matching capability to object domain models via integration with full-text search engines. In such approaches, data resides in two different and independent systems, i.e., an RDBMS and a search engine, and the corresponding data models need to be "glued together" somehow at the application level. In contrast, our solution requires no support of specialized systems.

Finally, there is a long and fruitful line of research on similarity operations over image and other complex data types (e.g., see [Bedo et al. 2014] and references therein). Most techniques developed in this area are quite different from the ones considered here for string data. Integration of similarity matching techniques for image and string data into a single framework is an interesting direction for future research.

6. CONCLUSION

This article addressed the problem of bridging database applications and string similarity operations on RDBMSs, which has received very little attention in the existing literature. We proposed the SimDataMapper, an architectural pattern that facilitates the incorporation of similarity matching

into database applications by isolating in-memory objects from details of similarity processing in the RDBMS. A flexible and efficient implementation of SimDataMapper was presented, which supports a rich variety of similarity functions and employs state-of-the-art optimization techniques. Our implementation is largely based on standard, declarative SQL statements and, thus, can be easily reused across different RDBMs platforms.

The current article extends a prior conference paper [Schneider et al. 2015] by including support to similarity functions on unweighted sets, detailed description of the strategy for handling modifications on source tables, and a comprehensive set of new experiments covering several aspects not discussed previously. In this context, we compared our approach with platform-specific solutions based on customized implementations for particular similarity functions. The results showed that our implementation is efficient, often substantially faster than the custom implementations. For token-based similarity functions, the cases where our approach was slower mostly involved lower threshold values and large strings. The reasons are that the optimization techniques employed are less effective for low thresholds and the current limitation of the underlying token-set representation and processing model in dealing with large strings. For edit distance, our approach significantly outperformed the custom implementation in all settings.

Future work is mainly oriented towards extending SimDataMapper to support complex similarity operations involving multiple tables and attributes. We also plan to investigate cost models to improve the quality of the evaluation plans picked by query optimizer and the use of more efficient physical operators to increase overall performance on large strings.

REFERENCES

Augsten, N., Miraglia, A., Neumann, T., and Kemper, A. On-the-fly Token Similarity Joins in Relational Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Snowbird, USA, pp. 1495–1506, 2014.

Bedo, M. V. N., Traina, A. J. M., and Traina Jr., C. Seamless Integration of Distance Functions and Feature Vectors for Similarity-queries Processing. *Journal of Information and Data Management* 5 (3): 308–320, 2014.

Chaudhuri, S., Ganti, V., and Kaushik, R. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the IEEE International Conference on Data Engineering*. Atlanta, USA, pp. 1–12, 2006.

Cook, W. R. and Ibrahim, A. H. Integrating Programming Languages & Databases: What's the Problem? http://www.odbms.org/2006/10/integrating-programming-languages-and-databases-what-is-the-problem/, 2006.

Fowler, M. *Pattern of Enterprise Application Architecture*. Addison-Wesley, 2003.

Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. Approximate String Joins in a Database (Almost) for Free. In *Proceedings of the International Conference on Very Large Data Bases*. Rome, Italy, pp. 491–500, 2001.

Gravano, L., Ipeirotis, P. G., Koudas, N., and Srivastava, D. Text Joins in an RDBMS for Web Data Integration. In *Proceedings of the International World Wide Web Conferences*. Budapest, Hungary, pp. 90–101, 2003.

Koudas, N., Marathe, A., and Srivastava, D. Propagating Updates in SPIDER. In *Proceedings of the IEEE International Conference on Data Engineering*. Istanbul, Turkey, pp. 1146–1153, 2007.

Navarro, G. A Guided Tour to Approximate String Matching. *ACM Computing Surveys* 33 (1): 31–88, 2001.

Rae, I., Halverson, A., and Naughton, J. F. In-RDBMS Inverted Indexes Revisited. In *Proceedings of the IEEE International Conference on Data Engineering*. Chicago, USA, pp. 352–363, 2014.

Ribeiro, L. A. and Härder, T. Embedding Similarity Joins into Native XML Databases. In *Proceedings of the Brazilian Symposium on Databases*. João Pessoa, Brazil, pp. 285–299, 2007.

Ribeiro, L. A. and Härder, T. Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems* 36 (1): 62–78, 2011.

Sarawagi, S. and Kirpal, A. Efficient Set Joins on Similarity Predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Paris, France, pp. 743–754, 2004.

Schneider, N. C., Ribeiro, L. A., de Souza Inácio, A., Wagner, H. M., and von Wangenheim, A. SimDataMapper: An Architectural Pattern to Integrate Declarative Similarity Matching into Applications. In *Proceedings of the Brazilian Symposium on Databases*. Petrópolis, Brazil, pp. 125–136, 2015.

Xiao, C., Wang, W., Lin, X., Yu, J. X., and Wang, G. Efficient Similarity Joins for Near-duplicate Detection. *ACM Transactions on Database Systems* 36 (3): 15:1–15:41, 2011.