A Primitive Operator for Similarity Joins in Data Cleaning

Surajit Chaudhuri Venkatesh Ganti Raghav Kaushik
Microsoft Research
{surajitc,vganti,skaushi}@microsoft.com

Abstract

Data cleaning based on similarities involves identification of "close" tuples, where closeness is evaluated using a variety of similarity functions chosen to suit the domain and application. Current approaches for efficiently implementing such similarity joins are tightly tied to the chosen similarity function. In this paper, we propose a new primitive operator which can be used as a foundation to implement similarity joins according to a variety of popular string similarity functions, and notions of similarity which go beyond textual similarity. We then propose efficient implementations for this operator. In an experimental evaluation using real datasets, we show that the implementation of similarity joins using our operator is comparable to, and often substantially better than, previous customized implementations for particular similarity functions.

1. Introduction

Data cleaning is an essential step in populating and maintaining data warehouses and centralized data repositories. A very important data cleaning operation is that of "joining" similar data. For example, consider a sales data warehouse. Owing to various errors in the data due to typing mistakes, differences in conventions, etc., product names and customer names in sales records may not match exactly with master product catalog and reference customer registration records respectively. In these situations, it would be desirable to perform *similarity joins*. For instance, we may join two products (respectively, customers) if the similarity between their part descriptions (respectively, customer names and addresses) is high. This problem of joining similar data has been studied in the context of record linkage (e.g. [6, 7]), of identifying approximate duplicate entities in databases (e.g., [5, 9, 11]). It is also relevant when identifying for a given record the best few approximate matches from among a reference set of records [4]. The similarity join is the fundamental operation upon which many of these techniques are built.

Current approaches exploit similarity between attribute values to join data across relations, e.g., similarities in part descriptions in the above example. A variety of string similarity functions have been considered, such as edit distance, jaccard similarity, cosine similarity and generalized edit distance ([4]), for measuring similarities. However, no single string similarity function is known to be the overall best similarity function, and the choice usually depends on the application domain [10, 13] (see also Section 6). For example, the characteristics of an effective similarity function for matching products based on their part names where the errors are usually spelling errors would be different from those matching street addresses because even small differences in the street numbers such as "148th Ave" and "147th Ave" are crucial, and the soundex function for matching person names.

The *similarity join* of two relations R and S both containing a column A is the join $R \bowtie_\theta S$ where the join predicate θ is $f(R.A, S.A) > \alpha$, for a given similarity function f and a threshold α. Although similarity joins may be expressed in SQL by defining join predicates through user-defined functions (UDFs), the evaluation would be very inefficient as database systems usually are forced to apply UDF-based join predicates only after performing a cross product. Consequently, specialized techniques have been developed to efficiently compute similarity joins. However, these methods are all customized to particular similarity functions (e.g., [1, 8, 9]).

A general purpose data cleaning platform, which has to efficiently support similarity joins with respect to a variety of similarity functions is faced with the impractical option of implementing and maintaining efficient techniques for a number of similarity functions, or the challenging option of supporting a foundational primitive which can be used as a building block to implement a broad variety of notions of similarity.

In this paper, we propose the SSJoin *operator* as a foundational primitive and show that it can be used for supporting similarity joins based on several string similarity functions—e.g., edit similarity, jaccard similarity, generalized edit similarity, hamming distance, soundex, etc.—as well as similarity based on cooccurrences [1]. In defining the SSJoin operator, we exploit the observation that set

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)
8-7695-2570-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

overlap can be used effectively to support a variety of similarity functions [13]. The SSJoin operator compares values based on "sets" associated with (or explicitly constructed for) each one of them. As we will show later, the design and implementation of this logical operator leverages the existing set of relational operators, and helps define a rich space of alternatives for optimizing queries involving similarity joins.

The SSJoin—denoting *set similarity* join—operator applies on two relations R and S both containing columns A and B. A group of $R.B$ values in tuples sharing the same $R.A$ value constitutes the set corresponding to the $R.A$ value. The SSJoin operator returns pairs of distinct values $\langle R.A, S.A\rangle$ if the overlap of the corresponding groups of $R[B]$ and $S[B]$ values is above a user specified threshold. We allow both weighted and unweighted versions. As an example, consider two relations $R[\mathsf{state},\mathsf{city}]$ and $S[\mathsf{state},\mathsf{city}]$. Setting $A = \mathsf{state}$ and $B = \mathsf{city}$, the SSJoin operator returns pairs of $\langle R.state, S.state\rangle$ values if the overlap between sets of cities which occur with each state is more than a threshold. So, it may return the pairs ('washington', 'wa') and ('wisconsin', 'wi') because the sets of cities within these groups overlap significantly. We will show in Section 3 that similarity joins based on a variety of similarity functions can be cast into a setting leveraging the SSJoin operator.

We then develop efficient implementations for the SSJoin operator. We first show that the SSJoin operator can be implemented in SQL using equi-joins. We further optimize the implementation for scenarios where the overlap has to be high based on the intuition that a high overlap between two sets implies that smaller subsets of the two sets also overlap. For example, if the overlap between two sets with 5 elements each has to be greater than 4, then size-2 subsets have a non-zero overlap. Based on this observation, we significantly reduce the number of candidate $\langle R.A, S.A\rangle$ groups to be compared. We observe that this implementation can also be carried out using traditional relational operators plus the groupwise processing operator [3], making it much easier to integrate with a relational engine. Not surprisingly, our proposed techniques are significantly better than using UDFs to compute similarities, which usually results in plans based on cross products.

The rest of the paper is organized as follows. In Section 2, we define the SSJoin operator. In Section 3, we instantiate similarity joins based on a variety of similarity functions. In Section 4, we describe an efficient physical implementation for the SSJoin operator. In Section 5, we show using several real datasets that our physical implementations are efficient, and sometimes substantially better than custom implementations. We discuss related work in Section 6, and conclude in Section 7.

2. Similarity Based on Set Overlap

In this section, we formally define the SSJoin operator by considering a simple notion of string similarity by mapping the strings to sets and measuring their similarity using set *overlap*. We then define the SSJoin operator that can be used to evaluate this notion of set overlap similarity.

There are several well-known methods of mapping a string to a set, such as the set of words partitioned by delimiters, the set of all substrings of length q, i.e., its constituent *q-grams*, etc. For example, the string "Microsoft Corporation" could be treated as a set of words {'Microsoft', 'Corp'}, or as a set of 3-grams, {'Mic', 'icr', 'cro', 'ros', 'oso', 'sof', 'oft', 'ft ', 't C', ' Co', 'Cor', 'orp'}. Henceforth, we refer to the set corresponding to a string σ as $Set(\sigma)$. This set could be obtained by any of the above methods. In this paper, we focus on multi-sets. Whenever we refer to sets, we mean multi-sets. Hence, when we refer to the union and intersection of sets, we mean the multi-set union and multi-set intersection respectively.

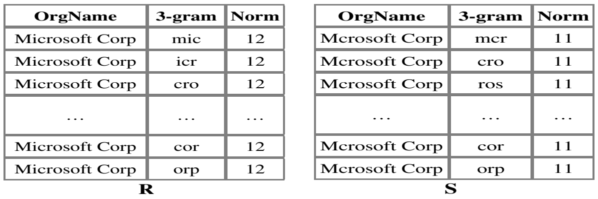
In general, elements may be associated with weights. This is intended to capture the intuition that different portions of a string have different importance. For example, in the string "Microsoft Corp", we may want to associate more importance to the portion "Microsoft". There are well-known methods of associating weights to the set elements, such as the notion of Inverse Document Frequency (IDF) commonly used in Information Retrieval. We assume that the weight associated with an element of a set, such as a word or q-gram, is fixed and that it is positive. Formally, all sets are assumed to be drawn from a universe $\mathcal{U}$. Each distinct value in $\mathcal{U}$ is associated with a unique *weight*. The weight of a set s is defined to be the sum of the weights of its members and is denoted as $wt(s)$. Henceforth, in this paper, we talk about weighted sets, noting that in the special case when all weights are equal to 1, we reduce to the unweighted case.

Given two sets s_1, s_2, we define their *overlap similarity*, denoted $Overlap(s_1, s_2)$, to be the weight of their intersection, i.e., $wt(s_1 \cap s_2)$. The overlap similarity between two strings, σ_1, σ_2, $Overlap(\sigma_1, \sigma_2)$ is defined as $Overlap(Set(\sigma_1), Set(\sigma_2))$.

Given relations R and S, each with string valued attribute A, consider the similarity join between R and S that returns all pairs of tuples where the overlap similarity between $R.A$ and $S.A$ is above a certain threshold. We expect that when two strings are almost equal, their overlap similarity is high, and hence this is a natural similarity join predicate to express. We next introduce the SSJoin operator that can be used to express this predicate.

In this paper, we assume the standard relational data model for simplicity. But, our techniques are also applicable to other models which allow inline representation of set-valued attributes. We assume that all relations are in the First

Normal Form, and do not contain set-valued attributes. Sets and hence the association between a string and its set are also represented in a normalized manner. For example, the set of rows in relation R of Figure 1 represents the association between the string "Microsoft Corp" and its 3-grams; the third *norm* column denotes the length of the string.



OrgName	3-gram	Norm
Microsoft Corp	mic	12
Microsoft Corp	icr	12
Microsoft Corp	cro	12
...	...	...
Microsoft Corp	cor	12
Microsoft Corp	orp	12

R

OrgName	3-gram	Norm
Mcrosoft Corp	mcr	11
Mcrosoft Corp	cro	11
Mcrosoft Corp	ros	11
...	...	...
Mcrosoft Corp	cor	11
Mcrosoft Corp	orp	11

S

Figure 1. Example sets from strings

We describe the SSJoin operator. Consider relations $R(A, B)$ and $S(A, B)$ where A and B are subsets of columns. Each distinct value $a_r \in R.A$ defines a group, which is the subset of tuples in R where $R.A = a_r$. Call this set of tuples $Set(a_r)$. Similarly, each distinct value $a_s \in S.A$ defines a set $Set(a_s)$. The simplest form of the SSJoin operator joins a pair of distinct values $\langle a_r, a_s \rangle$, $a_r \in R.A$ and $a_s \in S.A$, if the projections on column B of the sets $Set(a_r)$ and $Set(a_s)$ have a high overlap similarity. The formal predicate is $Overlap(\pi_B(Set(a_r), \pi_B(Set(a_s))) \geq \alpha$ for some threshold α. We denote $Overlap(\pi_B(Set(a_r), \pi_B(Set(a_s)))$ as $Overlap_B(a_r, a_s)$. Hence, the formal predicate is $Overlap_B(a_r, a_s) \geq \alpha$. We illustrate this through an example.

Example 1 : *Let relation $R(OrgName, 3\text{-}gram)$ and $S(OrgName, 3\text{-}gram)$ shown in Figure 1 associate the strings "Microsoft Corp" and "Mcrosoft Corp" with their 3-grams. Denoting* OrgName *by* A *and* 3-gram *by* B, *the* SSJoin *operator with the predicate* $Overlap_B(a_r, a_s) \geq 10$ *returns the pair of strings* $\langle$*"Microsoft Corp", "Mcrosoft Corp"*$\rangle$ *since the overlap between the corresponding sets of 3-grams is 10.*

In general, we may wish to express conditions such as: the overlap similarity between the two sets must be 80% of the set size. Thus, in the above example, we may wish to assert that the overlap similarity must be higher than 80% of the number of 3-grams in the string "Microsoft Corp". We may also wish to be able to assert that the overlap similarity be higher than say 80% of the sizes of *both* sets. We now formally define the SSJoin operator as follows, which addresses these requirements.

Definition 1: *Consider relations $R(A, B)$ and $S(A, B)$. Let $pred$ be the predicate $\bigwedge_i \{Overlap_B(a_r, a_s) \geq e_i\}$, where each e_i is an expression involving only constants and columns from either $R.A$ or $S.A$. We write R SSJoin_A^{pred} S to denote the following result: $\{\langle a_r, a_s \rangle \in R.A \times S.A | pred(a_r, a_s)$ is true $\}$.*

We also write $pred$ as $\{Overlap_B(a_r, a_s) \geq e_i\}$.

We illustrate this through the following examples based on Figure 1. The third column *Norm* denotes the length of the string. In general, the *norm* denotes either the length of the string, or the cardinality of the set, or the sum of the weights of all elements in the set. Several similarity functions use the norm to normalize the similarity.

Example 2 : *As shown in Figure 1, let relations $R(OrgName, 3\text{-}gram, Norm)$ and $S(OrgName, 3, Norm)$ associate the organization names with (1) all 3-grams in each organization name, and (2) the number of 3-grams for each name. The predicate in the* SSJoin *operator may be instantiated in one of the following ways to derive different notions of similarity.*

- *Absolute overlap: $Overlap_B(a_r, a_s) \geq 10$ joins the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since the overlap between the corresponding sets of 3-grams is 10.*
- *1-sided normalized overlap: $Overlap_B(\langle a, norm \rangle_r, \langle a, norm \rangle_s) \geq 0.8 \cdot R.norm$ joins the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since the overlap between the corresponding sets of 3-grams is 10, which is more than 80% of 12.*
- *2-sided normalized overlap: $Overlap_B(\langle a, norm \rangle_r, \langle a, norm \rangle_s) \geq \{0.8 \cdot R.norm, 0.8 \cdot S.norm\}$ also returns the pair of strings $\langle$"Microsoft Corp", "Mcrosoft Corp"$\rangle$ since 10 is more than 80% of 12 and 80% of 11.*

In the next section, we show how the intuitive notion of set overlap can be used to capture various string similarity functions. We discuss the implementation of the SSJoin operator in Section 4.

3. Using SSJoin Operator for Similarity Joins

In this section, we show illustrate the usage of the SSJoin operator to implement similarity joins based on a variety of previously proposed string similarity functions. Earlier techniques relied on distinct specialized implementations for each similarity function. In contrast, our approach relies on the SSJoin operator to perform bulk of the effort. Only a few checks have to be performed on the result of the SSJoin operator. Both the coding effort for programming these checks and the additional number of such checks is very small.

In this section, without loss of generality and for clarity in description, we fix unary relations Rbase(A) and Sbase(A) where A is a string-valued attribute. The goal

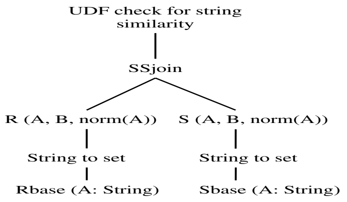


Figure 2. String Similarity Join using SSJoin

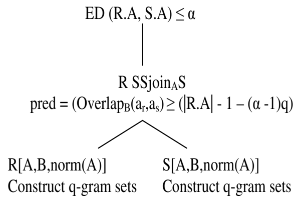


Figure 3. Edit distance join

is to find pairs $\langle \text{Rbase}.A, \text{Sbase}.A\rangle$ where the textual similarity is above a threshold α. Our approach (outlined in Figure 2) is to first convert the strings Rbase(A) and Sbase(A) to sets, construct normalized representations $R(A, B, norm(A))$ and $S(A, B, norm(A))$, and then suitably invoke the SSJoin operator on the normalized representations. The invocation is chosen so that all string pairs whose similarity is greater than α are guaranteed to be in the result of the SSJoin operator. Hence, the SSJoin operator provides a way to efficiently produce a small superset of the correct answer. We then compare the pairs of strings using the actual similarity function, declared as a UDF within a database system, to ensure that we only return pairs of strings whose similarity is above α.

Note that a direct implementation of the UDF within a database system is most likely to lead to a cross-product where the UDF is evaluated for all pairs of tuples. On the other hand, an implementation using SSJoin exploits the support within database systems for equi-joins to result in a significant reduction in the total number of string comparisons. This results in orders of magnitude improvement in performance, as we will discuss in Sections 4 and 5.

3.1. Edit Distance

The edit distance between strings is the least number of edit operations (insertion and deletion of characters, and substitution of a character with another) required to transform one string to the other. For example, the edit distance between strings ‘microsoft’ and ‘mcrosoft’ is 1, the number of edits (deleting ‘i’) required to match the second string with the first. The edit distance may be normalized to be between 0 and 1 by the maximum of the two string lengths. Hence, the notion of edit similarity can also be defined as follows.

Definition 2: *Given two strings σ_1 and σ_2, the* edit distance $\text{ED}(\sigma_1, \sigma_2)$ *between them is the minimum number of edit operations—insertion, deletion, and substitution—to transform σ_1 into σ_2. We define the* edit similarity $ES(\sigma_1, \sigma_2)$ *to be* $1.0 - \frac{\text{ED}(\sigma_1,\sigma_2)}{max(|\sigma_1|,|\sigma_2|)}$.

We consider the form of edit distance join addressed in [9], which returns all pairs of records where the edit distance is less than an input threshold α. This implementation can be easily extended to edit similarity joins.

We illustrate the connection between edit distance and overlap through the following example.

Definition 3: *Consider the strings “Microsoft Corp” and “Mcrosoft Corp”. The edit distance between the two is 1 (deleting ’i’). The overlap similarity between their 3-grams is 10, more than 80% of the number of 3-grams in either string.*

The intuition is all q-grams that are “far away” from the place where the edits take place must be identical. Hence, if the edit distance is small, then the overlap on q-grams must be high. The authors of [9] formalize this intuitive relationship between edit distance and the set of q-grams:

Property 4: *[9] Consider strings σ_1 and σ_2, of lengths $|\sigma_1|$ and $|\sigma_2|$, respectively. Let $QGSet_q(\sigma)$ denote the set of all contiguous q-grams of the string σ. If σ_1 and σ_2 are within an edit distance of ϵ, then $Overlap(QGSet_q(\sigma_1), QGSet_q(\sigma_2)) \geq max(|\sigma_1|, |\sigma_2|) - q + 1 - \epsilon \cdot q$*

Thus, in the above example, the edit distance is 1, and Property 4 asserts that at least 9 3-grams have to be common.

From the above property, we can implement the edit similarity join through the operator tree shown in Figure 3. We first construct the relations $R(A, B, norm(A))$ and $S(A, B, norm(A))$ containing the norms and q-gram sets for each string. We then invoke the SSJoin operator over these relations in order to identify $\langle R.A, S.A\rangle$ pairs which are very similar. Note that we further require a filter based on edit similarity (possibly as a user-defined function) in order to filter out pairs whose overlap similarity is higher than that given by Property 4 but edit similarity is still less than the required threshold.

3.2. Jaccard Containment and Resemblance

We define the Jaccard containment and resemblance between strings through the Jaccard containment and resemblance of their corresponding sets. We then illustrate the use of the SSJoin operator for Jaccard containment using the following example.

Definition 5: *Let s_1 and s_2 be weighted sets.*

1. *The Jaccard containment of s_1 in s_2,* $\mathsf{JC}(\mathsf{s}_1, \mathsf{s}_2)$ *is defined to be* $\frac{wt(s_1 \cap s_2)}{wt(s_1)}$.
2. *The Jaccard resemblance between s_1 and s_2,* $\mathsf{JR}(\mathsf{s}_1, \mathsf{s}_2)$*, is defined to be* $\frac{wt(s_1 \cap s_2)}{wt(s_1 \cup s_2)}$.

Example 3: *Suppose we define the Jaccard containment between two strings by using the underlying sets of 3-grams. Consider strings σ_1 = "Microsoft Corp" and σ_2 = "Mcrosoft Corp". We show how a Jaccard containment predicate on these strings translates to a* SSJoin *predicate. Suppose we want to join the two strings when the Jaccard containment of σ_1 in σ_2 is more than 0.8.*

As shown in Figure 1, let $R(OrgName, 3{-}gram, norm)$ and $S(OrgName, 3{-}gram, norm)$ associate the strings "Microsoft Corp" and "Mcrosoft Corp" with (1) the actual 3-grams in column $3 - gram$, and (2) the number of 3-grams in column $norm$.

We can see that the Jaccard containment predicate is equivalent to the following SSJoin *predicate: $Overlap_B(\langle a, norm\rangle_r, \langle a, norm\rangle_s) \geq 0.8 \cdot R.norm$.*

In general, we construct relations $R\langle A, B, norm(A)\rangle$ and $S\langle A, B, norm(A)\rangle$ from Rbase and Sbase respectively, that associates a string with (1) the weight of the underlying set, and (2) the set of elements in its underlying set. The Jaccard containment condition can then be expressed using the operator tree shown in Figure 4. Note that because Jaccard containment like the SSJoin operator measures the degree of overlap, this translation does not require a post-processing step.

Observe that for any two sets s_1 and s_2, $\mathsf{JC}(s_1, s_2) \geq \mathsf{JR}(s_1, s_2)$. Hence, $\mathsf{JR}(s_1, s_2) \geq \alpha \Rightarrow \mathsf{Max}(\mathsf{JC}(s_1, s_2), \mathsf{JC}(s_2, s_1)) \geq \alpha$. Therefore, as shown on the right hand side in Figure 4, we use the operator tree for Jaccard containment and add the check for Jaccard resemblance as a post-processing filter. In fact, we check for the Jaccard containment of $\mathsf{JC}(R.A, S.A)$ and $\mathsf{JC}(S.A, R.A)$ being greater than α.

3.3. Generalized Edit Similarity

This similarity function, introduced in [4] is a weighted variant of edit distance. The idea is to address some limitations of plain edit distance, illustrated through the following example. Consider strings "microsoft corp", "microsft corporation" and "mic corp". The edit distance between "microsoft corp" and "mic corp" is less than that between "microsoft corp" and "microsft corporation". So is the case for Jaccard similarity because it only matches tokens which are identical.

To deal with these limitations, the generalized edit similarity (GES) function was proposed in [4]. Each string is interpreted as a sequence of tokens, through some tokenizing function. The edit operations that transform one sequence into another include insertion, deletion and replacement of one token with another. Each edit operation is associated with a cost dependent on the tokens (and their weights) involved in the edit. To illustrate, consider the above example strings. The strings "microsoft corp" and "microsft corporation" are close because 'microsoft' and 'microsft' are close according to edit distance and the weights of 'corp' and 'corporation' are relatively small owing to their high frequency. GES has been shown to be very effective for matching erroneous tuples with their correct counterparts [4]. Let $ed(\sigma_1, \sigma_2)$ denote the absolute edit distance normalized by the maximum of the strings lengths, i.e., $ed(\sigma_1, \sigma_2) = \frac{ED(\sigma_1, \sigma_2)}{max(|\sigma_1|, |\sigma_2|)}$.

Definition 6: *Let σ_1 and σ_2 be two strings. The cost of transforming a token t_1 in the set $Set(\sigma_1)$ of tokens corresponding to σ_1 to a token t_2 in $Set(\sigma_2)$ is $ed(t_1, t_2) \cdot wt(t_1)$. The cost of inserting or deleting a token t equals $wt(t)$. The cost $tc(\sigma_1, \sigma_2)$ of transforming σ_1 to σ_2 is the minimum cost transformation sequence for transforming σ_1 into σ_2. The generalized edit similarity $GES(\sigma_1, \sigma_2)$ is defined as follows.*

$$GES(\sigma_1, \sigma_2) = 1.0 - min(\frac{tc(\sigma_1, \sigma_2)}{wt(Set(\sigma_1))}, 1.0)$$

We now illustrate the connection between GES and the SSJoin predicate.

Example 4: *Consider strings σ_1 = "Microsoft Corp" and σ_2 = "Mcrosoft Corp". Consider the sets* $\mathsf{Set}(\sigma_1)$ = *{Microsoft, Corp} and* $\mathsf{Set}(\sigma_2)$ = *{Mcrosoft, Corp} obtained using the tokenizing function and ignoring the sequentiality among tokens. Suppose, we expand* $\mathsf{Set}(\sigma_1)$ *to* $\mathsf{ExpandedSet}(\sigma_1)$ = *{Microsoft, Mcrosoft, Macrosoft, Corp} by including tokens (say, from a dictionary) whose edit similarity with any token in* $\mathsf{Set}(\sigma_1)$ *is high. Then, the overlap between* $\mathsf{ExpandedSet}(\sigma_1)$ *and* $\mathsf{Set}(\sigma_2)$ *is high.*

The above example illustrates the basic intuition. Informally, the expansion adds to a set corresponding to $R.A$ all tokens from a dictionary (say, all tokens in any attribute value of $S.A$) whose edit similarity with any token in the set is greater than a threshold β ($< \alpha$). If the generalized edit similarity between the strings σ_1 and σ_2 is higher than α then the overlap between their expanded sets must be higher than $\alpha - \beta$. The intuition is that the cost of transforming any token t_1 in $\mathsf{Set}(\sigma_1)$ to a token t_2 in $\mathsf{Set}(\sigma_2)$ is either (i) less than β if there is an overlapping token t' between the expanded sets that is close to both t_1 and t_2, or (ii) greater than β, otherwise. Therefore, the similarity is bounded by $\alpha - \beta$ if the overlap is greater than α. In general, we can expand both sets $\mathsf{Set}(\sigma_1)$ and $\mathsf{Set}(\sigma_2)$ by including similar tokens. The details are intricate and require a generalization of our element weight model to allow an element having different weights as opposed to a fixed weight. In the interest

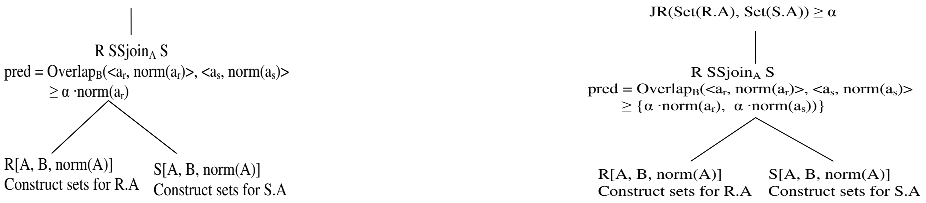

R SSjoin$_A$ S
pred = Overlap$_B$(<a$_r$, norm(a$_r$)>, <a$_s$, norm(a$_s$)> ≥ α ·norm(a$_r$)
R[A, B, norm(A)]
Construct sets for R.A
S[A, B, norm(A)]
Construct sets for S.A
JR(Set(R.A), Set(S.A)) ≥ α
R SSjoin$_A$ S
pred = Overlap$_B$(<a$_r$, norm(a$_r$)>, <a$_s$, norm(a$_s$)> ≥ {α ·norm(a$_r$), α ·norm(a$_s$))}
R[A, B, norm(A)]
Construct sets for R.A
S[A, B, norm(A)]
Construct sets for S.A

Figure 4. Jaccard containment and resemblance joins

of space, we omit the details and note that all techniques described in this paper can be generalized appropriately.

3.4. Beyond Textual Similarity

We now illustrate the applicability of the SSJoin operator to similarity joins based upon non-textual notions of similarity. The first notion of similarity is based on that of "co-occurrence" between columns and the second is based on soft functional dependencies. We illustrate these observations using examples. Both the following examples are based on an example publication database involving tables storing papers and authors.

Using Co-occurrence

Example 5: *Suppose we have two tables, say from different sources that are being integrated, of author names joined with the titles of the papers, say with the schema* ⟨ptitle, aname⟩. *Since we want a unified view of all authors, we are interested in identifying author names that are likely to represent the same author. Now, if the naming conventions in the two sources are entirely different, it is quite likely that the textual similarity between the author names is only a partial indicator of their similarity. We are forced to rely on alternative sources of information for identifying duplicate author entities.*

In this instance, we can use the set of paper titles associated with each author to identify authors. The idea is that if two authors are the same, then the set of paper titles co-occurring with them must have a large overlap. We can express this using Jaccard containment, for instance, which translates directly into the SSJoin *operator, as shown in the operator tree in Figure 5. This notion of similarity has been shown to be very effective for identifying approximate duplicates [1].*

Our next example illustrates how functional dependencies can be exploited for approximate equality.

Using Soft Functional Dependencies

Another source of identifying duplicate information is soft functional dependencies (FDs), which may not hold on the entire relation but over a large subset of the relation. The FDs may not hold exactly for a variety of reasons: they may not have been enforced due to efficiency reasons, or the relation may be the union of relations from several independent sources. For example, a large percentage of `emails` (if they are valid) uniquely determine the author tuple. In general, if we wish to use the functional dependency $X \rightarrow A$ to identify two similar values of $R.A$, then we can simply proceed by performing an equi-join on $R.X$.

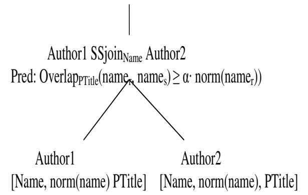

Author1 SSjoin$_{Name}$ Author2
Pred: Overlap$_{PTitle}$(name$_r$, name$_s$) ≥ α· norm(name$_r$))
Author1
[Name, norm(name) PTitle]
Author2
[Name, norm(name), PTitle]

Figure 5. Co-occurrence join using SSJoin

The question arises how we can exploit multiple FDs. Informally, two tuples agreeing on the source attributes of several FDs indicate that the target attribute values are the same. One natural way to aggregate the information from multiple functional dependencies is to use majority vote. We formalize this as follows. Let $\{X_1, \ldots, X_h, A\}$ be a set of columns in R and S. Each X_i is expected to functionally determine A.

Definition 7: *For two tuples* t_1 *and* t_2 *in* R*, we write* $t_1 \approx_{FD}^{k/h} t_2$ *if* t_1 *and* t_2 *agree on at least* k *out of the* h X_i*.*

Example 6 : *Consider two relations* `Authors1`, `Authors2`*, both with the schema* {`name, address, city, state, zip, email, phone`}. *We may want to join two author names if at least two of the following agree:* `address, email, phone`. *That would be expressed as* `Author1` $\approx_{\mathrm{FD}}^{2/3}$ `Author2`.

We illustrate how the SSJoin operator can be used to compute the $\approx_{\mathrm{FD}}^{k/h}$ predicate using the above example. By associating each author name with a set of ordered pairs ⟨`Column, Value`⟩ and normalizing the resulting relation, we get a relation with the schema `Name, AEP` (AEP for address-email-phone). We can implement the above predicate through the SSJoin operator as shown in Figure 6.

IEEE COMPUTER SOCIETY

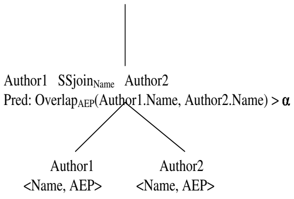

Author1 SSjoin$_{Name}$ Author2
Pred: Overlap$_{AEP}$(Author1.Name, Author2.Name) > α
Author1
<Name, AEP>
Author2
<Name, AEP>

Figure 6. FD-based join using SSJoin

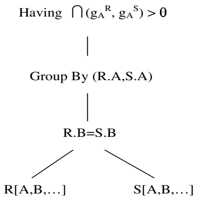


Figure 7. Basic implementation of SSJoin

4. Implementation of SSJoin

In this section, we discuss the implementation of the SSJoin operator. We consider various strategies, each of which can be implemented using relational operators. The idea is to exploit the property that SSJoin has to only return pairs of groups whose similarity is above a certain threshold, and that thresholds are usually high. In this section, we talk mostly about executing the operation R SSJoin$_A^{pred}$ S over relations $R(A, B)$ and $S(A, B)$ where the predicate is $Overlap_B(a_r, a_s) \geq \alpha$ for some positive constant α. The implementation extends to the case when $Overlap_B(a_r, a_s)$ is required to be greater than a set of expressions.

4.1. Basic SSJoin Implementation

Since $\alpha > 0$, we can conclude that for a pair $<a_r, a_s>$ to be returned, at least one of the values in the column B related to a_r and a_s must be the same. Indeed, by computing an equi-join on the B column(s) between R and S and adding the weights of all joining values of B, we can compute the overlap between groups on $R.A$ and $S.A$. Figure 7 presents the operator tree for implementing the basic overlap-SSJoin. We first compute the equi-join between R and S on the join condition $R.B = S.B$. Any $< R.A, S.A >$ pair whose overlap is non-zero would be present in the result. Grouping the result on $< R.A, S.A >$ and ensuring, through the having clause, that the overlap is greater than the specified threshold α would yield the result of the SSJoin.

The size of the equi-join on B varies widely with the joint-frequency distribution of B. Consider the case when the SSJoin operator is used to implement the Jaccard similarity between strings. Here, the values in the attribute B represent tokens contained in strings. Certain tokens like "the" and "inc" can be extremely frequent in both R and S relations. In such scenarios, which occur often, the size of the equi-join on B is very large (refer Section 5). The challenge, therefore, is to reduce the intermediate number of $<R.A, S.A>$ groups compared. Next, we describe our approach to address this problem.

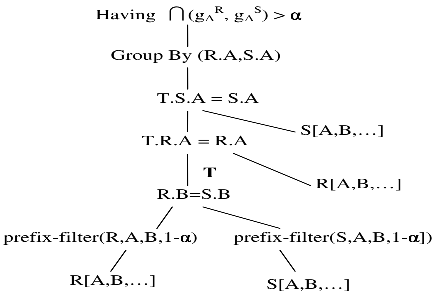


Figure 8. Prefix-filter implementation of SSJoin

4.2. Filtered SSJoin Implementation

The intuition we exploit is that when two sets have a *large* overlap, even smaller subsets of the base sets overlap. To make the intuition concrete, consider the case when all sets are unweighted and have a fixed size h. We can observe the following property.

Property 8: *Let s_1 and s_2 be two sets of size h. Consider any subset r_1 of s_1 of size $h - k + 1$. If $|s_1 \cap s_2| \geq k$, then $r_1 \cap s_2 \neq \phi$.*

For instance, consider the sets s_1={1,2,3,4,5} and s_2={1,2,3,4,6} which have an overlap of 4. Any subset of s_1 of size 2 has a non-zero overlap with the set s_2. Therefore, instead of performing an equi-join on R and S, we may ignore a large subset of S and perform the equi-join on R and a small filtered subset of S. By filtering out a large subset of S, we can reduce, often by very significant margins, the size of the resultant equi-join. We note here that this approach is similar to the OptMerge optimization in [13].

The natural question now is whether or not we can apply such a prefix-filter to both relations R and S in the equi-join. Interestingly, we find that the answer is in the affirmative. We illustrate this as follows. Fix an ordering $\mathcal{O}$ of the universe $\mathcal{U}$ from which all set elements are drawn. Define the k-prefix of any set s to be the subset consisting of the first k elements as per the ordering $\mathcal{O}$. Now, if $|s_1 \cap s_2| \geq k$, then their $(h - k + 1)$-prefixes must intersect. For example, consider $s_1 = \{1, 2, 3, 4, 5\}$ and $s_2 = \{1, 2, 3, 4, 6\}$ as before. Assume the usual ordering of natural numbers. Since the overlap between s_1 and s_2 is 4, their size $(5 - 4 + 1) = 2$-

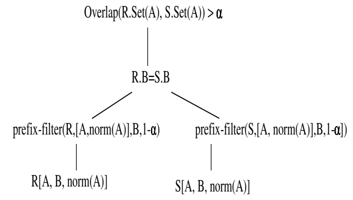

Overlap(R.Set(A), S.Set(A)) > α
R.B=S.B
prefix-filter(R,[A,norm(A)],B,1-α)
prefix-filter(S,[A, norm(A)],B,1-α])
R[A, B, norm(A)]
S[A, B, norm(A)]

Figure 9. Prefix-filter with inline set representation

prefixes must intersect, which is the case — the size-2 prefixes of both s_1 and s_2 is $\{1, 2\}$. Therefore, an equi-join on B on the filtered relations will return all pairs that satisfy the SSJoin predicate. The result would be a superset of all pairs of $< R.A, S.A >$ groups with overlap greater than the given threshold. And, the number of candidate groups of pairs is significantly (sometimes, by orders of magnitude) smaller than the number of pairs from the equi-join on the full base relations (refer Section 5).

This intuition can be extended to weighted sets. Consider any fixed ordering O of the domain from which $R.B$ and $S.B$ are drawn. Given a weighted set r drawn from this domain, define $prefix_\beta(r)$ to be the subset corresponding to the shortest prefix (in sorted order), the weights of whose elements add up to more than β. We have the following result:

Lemma 1: *Consider two weighted sets s_1 and s_2, such that $wt(s_1 \cap s_2) \geq \alpha$. Let $\beta_1 = wt(s_1) - \alpha$ and $\beta_2 = wt(s_2) - \alpha$. Then $prefix_{\beta_1}(s_1) \cap prefix_{\beta_2}(s_2) \neq \phi$.*

Suppose that for the set defined by value $a_r \in R.A$, $Set(a_r)$ (respectively for $a_s \in S.A$), we extract a $\beta_{a_r} = (wt(Set(a_r)) - \alpha)$ prefix under $\mathcal{O}$ (respectively, a β_{a_s} prefix). From the above lemma, performing the equi-join B on the resulting relations will result in a superset of the result of the SSJoin. We can then check the SSJoin predicate on the pairs returned. Since the filter is letting only a prefix under a fixed order to pass through, we call this filter the *prefix-filter*. We refer to the relation obtained by filtering R as *prefix-filter*(R, α).

The filtered overlap implementation of the SSJoin operator is illustrated in Figure 8. We first join the prefix-filtered relations to obtain candidate pairs $\langle R.A, S.A \rangle$ groups to be compared. We join the candidate set of pairs with the base relations R and S in order to obtain the groups so that we can compute the overlap between the groups. The actual computation of the overlap is done by grouping on $\langle R.A, S.A \rangle$ and filtering out groups whose overlap is less than α.

We need to extend this implementation to address the following issues.

- *Normalized Overlap Predicates:* Instead of a constant α as in the discussion above, if we have an expression of the form $\alpha \cdot R.Norm$, then we extract a $\beta_{a_r, norm(a_r)} = (wt(Set(a_r)) - \alpha \cdot norm(a_r))$ prefix of the set $Set(a_r)$. This generalizes to the case when we have an expression involving constants and $R.Norm$.
- For a 2-sided normalized overlap predicate $Overlap_B(a_r, a_s) \geq \alpha \cdot Max(R.Norm, S.Norm)$, we apply different prefix-filter to relations R and S. We apply the filter *prefix-filter*$(R, \alpha \cdot R.Norm)$ to R and *prefix-filter*$(S, \alpha \cdot S.Norm)$ to S.
- For the evaluation of a 1-sided normalized overlap predicate $Overlap_B(a_r, a_s) \geq \alpha \cdot R.Norm$, we can apply the prefix-filter only on sets in R.

4.3. Implementation Issues

We now discuss the implementation issues around the prefix-filter approach.

4.3.1 Mapping Multi-set Intersection to Joins

Observe that the form of predicate we consider here involves multi-set intersection when any $R.A$ (or $S.A$) group contains multiple values on the $R.B$ attributes. In order to be able to implement them using standard relational operators, we convert these multi-sets into sets; we convert each value in $R.B$ and $S.B$ into an ordered pair containing an ordinal number to distinguish it from its duplicates. Thus, for example, the multi-set $\{1, 1, 2\}$ would be converted to $\{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle\}$. Since set intersections can be implemented using joins, the conversion enables us to perform multi-set intersections using joins.

4.3.2 Determining the Ordering

Note that the prefix-filter is applicable no matter what ordering $\mathcal{O}$ we pick. The question arises whether the ordering picked can have performance implications. Clearly, the answer is that it does. Our goal is to pick an ordering that minimizes the number of comparisons that the ordering will imply. One natural candidate here is to order the elements by increasing order of their frequency in the database. This way, we try to eliminate higher frequency elements from the prefix filtering and thereby expect to minimize the number of comparisons. Since many common notions of weights (e.g., IDF) are inversely proportional to frequency, we can implement this using the element weights. Several optimization issues arise such as to what extent will prefix-filtering help, whether it is worth the cost of producing the filtered relations, whether we should proceed by partitioning the relations and using different approaches for different partitions, etc.

In our implementation, we order $R.B$ values with respect to their IDF weights. Since high frequency elements have lower weights, we filter them out first. Therefore, the size of the subset (and hence the subsequent join result) let through would be the smallest under this ordering.

4.3.3 Implementing the prefix-filter

The prefix-filter can be implemented using a combination of standard relational operators such as group by, order by, and join, and the notion of groupwise processing [2, 3] where we iteratively process groups of tuples (defined as in group-by, i.e., where every distinct value in a grouping column constitutes a group) and apply a subquery on each group. In our case, we would group the tuples of R on $R.A$ and the subquery would compute the prefix of each group it processes.

In our implementation, we use a server-side cursor which requires the scan of the base relation R ordered on A, B. While scanning, we mark the prefix of each group $\mathsf{Set}(a_r)$. Observe that ordering $R.B$ with respect to the fixed order $\mathcal{O}$ of $R.B$ may require an additional join of R with the "order" table.

4.3.4 Inlined Representation of Groups

A property of the prefix-filter approach is that when we extract the prefix-filtered relations, we lose the original groups. Since the original groups are required for verifying the SSJoin predicate, we have to perform a join with the base relations again in order to retrieve the groups, as shown in Figure 8. These joins can clearly add substantially to the cost of the SSJoin operation.

Next, we discuss a new implementation which can avoid these joins. The idea is to "carry" the groups along with each $R.A$ and $S.A$ value that pass through the prefix-filter. This way, we can avoid the joins with the base relations. The intuition is illustrated in Figure 9. In order to do so, we either require the capability to define a set-valued attribute or a method to encode sets as strings or clobs, say by concatenating all elements together separating them by a special marker.

In our implementation, we choose the latter option. Now, measuring the overlap between $\langle R.A, S.A\rangle$ groups can be done without a join with the base relations. However, we require a function, say a UDF, for measuring overlap between inlined sets. This implementation goes beyond the capabilities of standard SQL operators as it requires us to compute set overlaps. However, the UDF we use is a simple unary operator that does not perform very sophisticated operations internally, especially when the sets are bounded. Our experiments show that this alternative is usually more efficient than the prefix-filtered implementation since it avoids the redundant joins.

We note here that our current implementation is not geared for the case of unbounded sets. Dealing with large sets is left for future work.

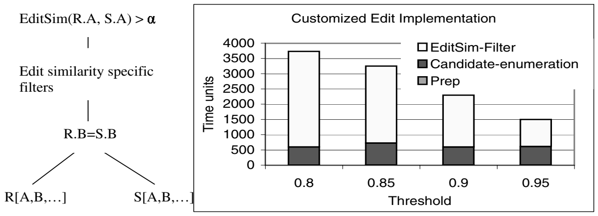


Figure 11. Customized edit similarity join

Threshold	SSJoin	Direct
0.80	546492	28252476
0.85	129925	21405651
0.90	16191	13913492
0.95	7772	5961246

Table 1. #Edit comparisons

5. Experiments

We now experimentally demonstrate the generality of the SSJoin operator in allowing several common similarity functions besides the efficiency of our physical implementations.

Datasets: All our experiments are performed using the *Customer* relation from an operational data warehouse. We evaluate the SSJoin operator by implementing similarity joins on a relation R of $25,000$ customer addresses with itself.

While performing Jaccard and generalized edit similarity joins between two relations R and S based on the attribute values in A, we assign IDF weights to elements of sets (tokens) as follows: $log(\frac{|R|+|S|}{f_t})$, where f_t is the total number of $R[A]$ and $S[A]$ values, which contain t as a token.

We implemented the SSJoin operator and the similarity join operations as client applications over Microsoft SQL Server 2005. We ran all our experiments on a desktop PC running Windows XP SP2 over a 2.2 GHz Pentium 4 CPU with 1GB main memory.

5.1. Evaluating the SSJoin Encapsulation

We compare our implementations with the best known customized similarity join algorithm for edit similarity [9]. No specialized similarity join algorithms have been proposed for Jaccard resemblance and generalized edit similarity. Therefore, we consider the basic SSJoin implementation as the strawman strategies for these similarity joins. Previous implementations for the generalized edit similarity were probabilistic [4]; hence, we do not compare our implementations with the earlier strategy.

The customized algorithm for edit similarity join is summarized by the operator tree on the left hand side of Figure 11: an equi-join on $R.B$ and $S.B$ along with additional filters (difference in lengths of strings has to be less, and the positions of at least one q-gram which is common to both

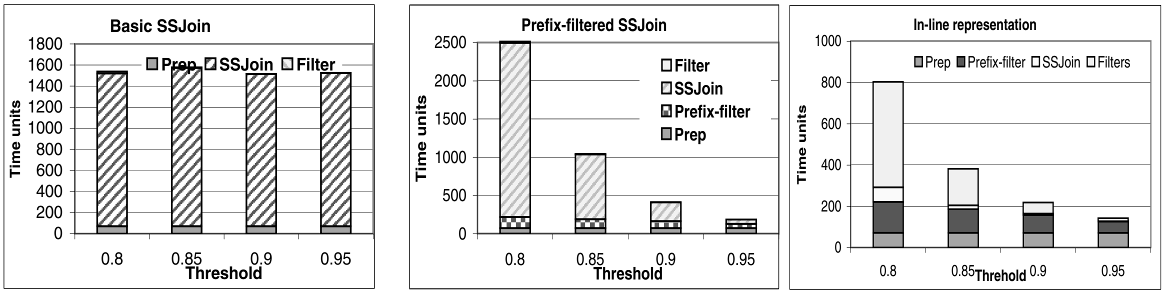


Figure 10. Edit similarity join: basic, prefix-filtered, inline-represented SSJoin

strings has to be close) followed by an invocation of the edit similarity computation.

Figure 11 plots the times required for implementing the edit similarity join using the customized algorithm. Comparing the times with those in Figure 10, we first note that the edit similarity joins based on our implementations (including the basic implementation) are faster than that using the custom implementation. The reason is that the custom implementation compares a very large number of strings. Table 1 plots the number of edit similarity computations through the SSJoin operator and that through the custom implementation. The custom implementation performs a much (by orders of magnitude) larger number of edit similarity comparisons. The SSJoin operator implementations rely on the overlap predicate in order to significantly reduce the number of edit similarity computations. Thus, using the more general SSJoin operator, we are able to implement the edit similarity join more efficiently than the previously known best customized solution.

Edit Similarity join: Figure 10 plots the times required for implementing a similarity join based upon edit similarity at various thresholds. The prefix-filtered implementations are significantly faster than the basic implementation at higher thresholds (greater than or equal to 0.85). However, at lower thresholds the basic implementation of the SSJoin operator is better than the prefix-filtered implementation using standard SQL operators. At lower thresholds, the number of $< R.A, S.A >$ group pairs to be compared is inherently high and any technique has to compare higher number of groups. So, even prefix-filters have to let a larger number of tuples to pass through. Therefore, the effectiveness of the prefix filters decreases as we decrease the thresholds. Therefore, prefix filtering is not as effective at lower thresholds. The additional cost of prefix filtering would not offset the savings resulting from reduced join costs.

That there is not always a clear winner between the basic and prefix-filtered implementations motivates the requirement for a cost-based decision for choosing the appropriate implementation. Because of our operator-centric approach, our SSJoin operator can be enhanced with such rules and integrated with a query optimizer. Observe that such cost-based choices are not possible for the framework proposed by Sarawagi et al. [13].

The prefix-filtered implementation with the inline set representation is still more efficient than the basic strategy even at lower thresholds. As expected, avoiding the overhead of joins with base relations to regroup elements significantly improves efficiency.

We also note that the plans chosen by the query optimizer only involved hash and merge joins. For no instance did the optimizer choose an index-based join even if we created clustered indexes on temporary tables. Therefore, we conclude that a fixed index-based strategy for similarity joins as in [13] and [6] is unlikely to be optimal always. Instead, we must proceed with a cost-based choice that is sensitive to the data characteristics.

Jaccard Resemblance: Figure 12 plots the times for implementing the Jaccard resemblance join at various thresholds through each implementation of the SSJoin operator. The prefix-filtered implementation is 5-10 times faster than the basic implementation. Therefore, prefix-filtering is very effective in reducing the cost of the SSJoin operator. Also, observe that the prefix-filtered implementation with inline representation is around 30% faster than the standard prefix-filtered implementation. As expected, avoiding joins with the base relations just to gather together all elements of groups to be compared is beneficial even if it means that additional information has to be carried through the prefix-filter for each tuple.

Observe that most of the time in the basic implementation is spent in the execution of the SSJoin. The preparation (denoted *Prep* in the figures) and the filtering (denoted *Filter* in the figures) take negligible fractions of the time. The time taken for the prefix-filtered implementation increases as we decrease the threshold. Such behavior is expected because the number of $< R.A, S.A >$ pairs pruned away decreases with the threshold. For the prefix-filtered implementations, a significant amount of time is spent in the prefix-filter due to which the subsequent steps are very efficient. These observations are true for our implementations of edit similarity and of generalized edit similarity joins.

Generalized Edit Similarity: Figure 13 plots the times required for implementing a similarity join based upon

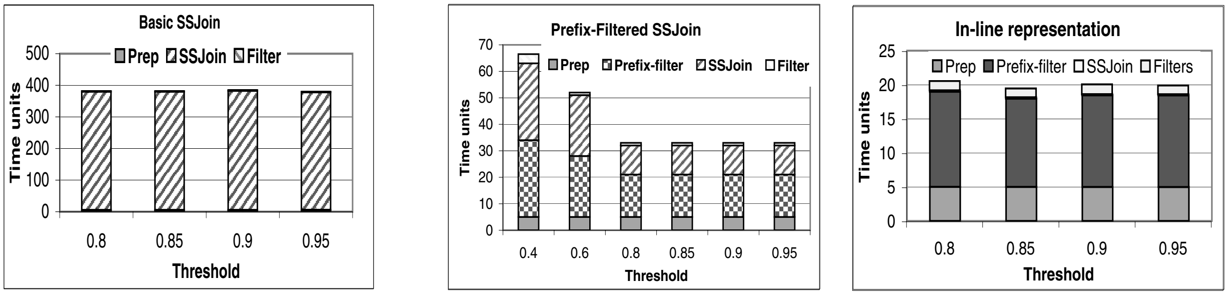


Figure 12. Jaccard similarity join: basic, prefix-filtered, and inline represented SSJoin

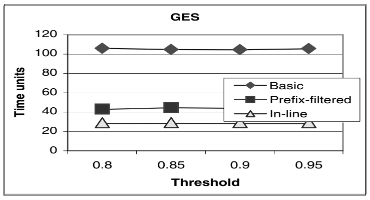


Figure 13. GES join

Input Size	SSJoin Input	Output	Time units
100K	288627 rows	2731	224
200K	778172 rows	2870	517
250K	1020197 rows	4807	649
330K	1305805 rows	3870	1072

Table 2. Varying input data sizes

the generalized edit similarity. The conclusions here are similar to those for the Jaccard similarity. The prefix-filtered implementations are better than those for the basic implementation by almost 2 times. Again, the inline representation is better by about 25% than the prefix-filtered implementation using standard SQL operators.

Beyond textual similarity: As discussed in Section 3, similarity notions based on agreements with respect to functional dependencies (say, at least k out of h FDs agree) and that of co-occurrence between attribute values with respect to a different attribute can both be reduced to Jaccard resemblance. The basic strategy for implementing the similarity based on agreements with respect to FDs using a disjunction of selection predicates results in a cross product plan being chosen by the optimizer. We have already seen that our physical implementations of the Jaccard resemblance join using the SSJoin operator can be significantly more efficient than the basic implementations and the cross product plans. Therefore, we do not discuss experiments based on these (non-textual) similarity functions.

Varying data sizes: Our implementations for the SSJoin operator rely primarily on the relational operators such as equi-join and group by. These operators have very efficient and scalable implementations within database systems. Hence, our implementations for the SSJoin operator are also scalable.

We perform a Jaccard similarity join of tables containing addresses with themselves; we vary the number of rows in each table by picking a random subsets from the original relation. We fix the threshold at 0.85. Table 2 presents the time required by the prefix-filtered implementation for increasing data sizes. We also report the sizes of the tables input to the SSJoin operator and the size of the output. As the input data size increases, the size of the prepared relations which are input to the SSJoin operator increases linearly. The output size is a characteristic of the data and it can vary widely. The time required depends crucially on the output size besides the input relation size. For instance, adding a large number of very similar pairs would increase the output size as well as the time significantly.

Summary: Based upon the above experiments implementing jaccard similarity, edit similarity and generalized edit similarity joins using the SSJoin operator, we conclude that (i) the SSJoin operator is general enough to implement a variety of similarity notions, both textual and non-textual, (ii) our algorithms to implement of the SSJoin operator are efficient — for the edit similarity join, our implementation is more efficient than the best known customized implementation — and (iii) the choice of physical implementation of the SSJoin operator must be cost-based.

6. Related Work

Sarawagi et al. [13] recognized that set overlap is an important measure for dealing with a variety of similarity functions. The main difference in our approach is our operator-centric focus. Sarawagi et al. [13] require plug-in functions in order to implement each similarity function, whereas our approach is to compose the SSJoin operator with other operators. Our design choice leads to the possibility of making cost-based decisions in choosing a physical implementation of similarity joins. For example, depending on the size of the relations being joined and the availability of indexes, the optimizer may choose either index-based plans or merge and hash joins in order to implement the SSJoin operator. This is in contrast to a fixed implementation based on inverted indexes, as in [13]. Further, our implementation of SSJoin operator is based upon relational operators present in a database system, making it much easier to integrate it into a database system.

Our notion of overlap similarity between groups is directly related to the notion of similarity which measures co-occurrence with respect to other attributes [1]. We build upon this notion of overlap and encapsulate it into a similarity join operator. Second, we further observe that the similarity join based on thresholded overlap similarity can be made into a primitive operator and applied to a variety of other textual similarity functions. We also propose efficient algorithms for implementing this primitive.

Custom join algorithms for particular similarity functions have been proposed for edit distance [9] and for cosine similarity [8, 6]. Top-K queries over string similarity functions have also received significant attention in the context of fuzzy matching where the goal is to match an incoming record against a reference table [6, 4]. However, there is no work yet on the design of primitive operators for top-K queries. However, we note that by composing the SSJoin operator with the top-k operator, we can address the form of top-K queries which ask for the best matches whose similarity is above a certain threshold.

Set containment joins in object-relational systems may also be used to express extreme forms of overlap queries where the degree of overlap has to be 1.0 (e.g., [12]). However, these techniques are not applicable for partial overlap queries, which is the focus of the SSJoin operator. Our techniques are also applicable for object-relational models which allow set-valued attributes. The prefix-filtered implementation with inline representation for sets can be directly implemented under these models.

7. Conclusions

In this paper, we introduce a primitive operator SSJoin for performing similarity joins. We showed that similarity joins based on a variety of textual and non-textual similarity functions can be efficiently implemented using the SSJoin operator. We then developed very efficient physical implementations for this operator mostly using standard SQL operators. In future, we intend to integrate the SSJoin operator with the query optimizer in order to make cost-conscious choices among the basic, prefix-filtered, and inline prefix-filtered implementations.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 28th international conference on very large databases (VLDB)*, pages 586–597, Hong Kong, August 20-23 2002.

[2] D. Chatziantoniou and K. Ross. Querying multiple features in relational databases. In *Proceedings of the VLDB Conference*, 1996.

[3] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *VLDB*, pages 476–485, 1997.

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD*, June 2003.

[5] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD*, pages 201–212, Seattle, WA, June 1998.

[6] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on information systems*, 18(3):288–321, July 2000.

[7] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64:1183–1210, 1969.

[8] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *In Proc. Intl. world Wide Web Conference*, pages 90–101, 2003.

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th international conference on very large databases (VLDB)*, pages 491–500, Roma, Italy, September 11-14 2001.

[10] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *Proceedings of the 30th international conference on very large databases (VLDB)*, pages 636–647, 2004.

[11] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM SIGMOD*, pages 127–138, San Jose, CA, May 1995.

[12] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, pages 351–362, 2000.

[13] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD*, pages 743–754, 2004.