Efficient Exact Set-Similarity Joins

Arvind Arasu
Microsoft Research
One Microsoft Way
Redmond, WA 98052
arvinda@microsoft.com

Venkatesh Ganti
Microsoft Research
One Microsoft Way
Redmond, WA 98052
vganti@microsoft.com

Raghav Kaushik
Microsoft Research
One Microsoft Way
Redmond, WA 98052
skaushi@microsoft.com

ABSTRACT

Given two input collections of sets, a *set-similarity join* (SSJoin) identifies all pairs of sets, one from each collection, that have high similarity. Recent work has identified SSJoin as a useful primitive operator in data cleaning. In this paper, we propose new algorithms for SSJoin. Our algorithms have two important features: They are exact, i.e., they always produce the correct answer, and they carry precise performance guarantees. We believe our algorithms are the first to have both features; previous algorithms with performance guarantees are only probabilistically approximate. We demonstrate the effectiveness of our algorithms using a thorough experimental evaluation over real-life and synthetic data sets.

1. INTRODUCTION

A data collection often has various inconsistencies which have to be fixed before the data can be used for accurate data analysis. The process of detecting and correcting such inconsistencies is known as *data cleaning*. A common form of inconsistency arises when a real-world entity has more than one representation in the data collection; for example, the same address could be encoded using different strings in different records in the collection. Multiple representations arise due to a variety of reasons such as misspellings caused by typographic errors and different formatting conventions used by data sources.

A *similarity join* is an important operation for reconciling different representations of an entity [9, 11, 16, 21]. Informally, a similarity join takes as input two relations, and identifies all pairs of records from the two relations that are syntactically similar. The notion of similarity is captured numerically using a string-based *similarity function*, and two records are considered similar if the value returned by the similarity function for these two records is greater than a threshold. For example, we can perform a similarity join on the `address` column of two customer tables to identify potential misspellings of the same physical address in the two tables.

A large number of different similarity functions such as edit distance, cosine similarity, jaccard similarity, and generalized edit distance [5] have been traditionally used in similarity joins. It is well-known that no single similarity function is universally applicable across all domains and scenarios [21]. For example, the characteristics of an effective similarity function for joining products based on their part names, where the errors are usually spelling errors, would be different from those joining street addresses because even small differences in the street numbers such as "148th Ave" and "147th Ave" are crucial, which would be different from similarity functions for joining person names based on their sounds.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

City	*State*
Los Angeles	CA
Palo Alto	CA
San Diego	CA
Santa Barbara	CA
San Francisco	CA
Seattle	WA
...	...

City	*State*
Los Angeles	California
San Diego	California
Santa Barbara	California
San Francisco	California
Sacramento	California
Seattle	Washington
...	...

Figure 1: SSJoins in data cleaning

A general-purpose data cleaning system is therefore faced with the daunting task of supporting a large number of similarity joins with different similarity functions. Recent work [6] has identified *set-similarity join (SSJoin)* as a powerful primitive for supporting (string-)similarity joins involving many common similarity functions. In other words, we can express these similarity joins using SSJoin as an sub-operator, and thereby avoid separate implementations for them from scratch.

Informally, SSJoin is defined as follows: Given two input collections of sets, identify all pairs of sets, one from each collection, that are highly similar. The similarity between two sets is captured using a general class of predicates involving the sizes of the sets and the size of their intersection. (We introduce the class of predicates formally in Section 2.) As a simple example, two sets can be considered similar if their intersection size is greater than a specified threshold.

Apart from string-based similarity, semantic relationships between entities can be exploited to identify different representations of the same entity [2, 10]. For example, in Figure 1, we can infer `CA` and `California` refer to the same state using the fact that there is a high similarity in their associated sets of cities. This approach is applicable even when there is no obvious syntactic similarity between different representations, as in this example. The SSJoin operator is naturally applicable here: By performing a set-similarity join over the sets of cities associated with the same state name in the two tables of Figure 1, we can join abbreviated and expanded representations of state names.

1.1 Our Contributions

In this paper, we present new algorithms—PARTENUM and WTENUM—for evaluating SSJoins. Our algorithms handle a large subclass of set-similarity predicates allowed by the definition of the SSJoin operator. In particular, this subclass includes threshold-

based predicates involving standard set-similarity measures such as jaccard and hamming (e.g., jaccard similarity greater than 0.9).

Our algorithms can be broadly characterized as *signature-based* algorithms that first generate *signatures* for input sets, then find all pairs of sets whose signatures overlap, and finally output the subset of these candidate pairs that satisfy the set-similarity predicate. Indeed, we observe in Section 3 that all previous algorithms for SSJoins are also signature-based, so we use the high-level structure of signature-based algorithms as a framework for comparing our algorithms with the previous ones.

One important feature of our algorithms is that for SSJoins involving jaccard and hamming, they provide a guarantee that two highly dissimilar sets will not appear as a candidate pair with a high probability. Consequently, they produce few false positive candidate pairs (candidate pairs that do not satisfy the set-similarity predicate), and are therefore efficient. In contrast, previous algorithms [6, 22] do not provide any such guarantee. Previous work [8, 15] has proposed probabilistic algorithms based on the idea of *locality-sensitive hashing (LSH)* [13] that have similar guarantees. However, these algorithms are *approximate* since they can miss some output set pairs; in contrast, all of our algorithms are *exact*, and never produce a wrong output. We believe our algorithms are the first exact ones with such performance guarantees.

Exact answers are important for data cleaning applications (the LSH-based algorithms have all been proposed in non-data-cleaning settings such as web informatics [15] and data mining [8]): In these applications, an SSJoin operator is typically used, not as a stand-alone operator, but as part of a larger query [6]. It is well-known [7] that it is hard to assign clean semantics to a query containing approximate operators. Recent work [12] has also explored an alternate setting, where data cleaning is performed on-the-fly during query evaluation, and not as a one-time offline process. Even here SSJoins appear as part of a larger query, so exact answers are important.

One drawback of the LSH-based algorithms is that they can be used only when the SSJoin predicate admits a locality-sensitive hash function; currently, locality-sensitive hash functions are known only for standard similarity measures such as jaccard. As we will show in Section 6, our algorithms handle a larger class of SSJoin predicates.

In addition to having better theoretical guarantees, our algorithms also empirically outperform previous exact algorithms [6, 22], often by more than an order-of-magnitude. More importantly, they exhibit superior scaling with input size compared to previous exact algorithms: Our algorithms scale almost linearly, while the previous ones achieve only quadratic scaling.

Our experiments also suggest that our exact algorithms are competitive with LSH-based algorithms. In many of the data cleaning scenarios that we consider, our algorithms perform better than the LSH-based ones, and in all other cases their performance is only marginally worse. This happens even though the LSH-based algorithms are set up to return only 95% of all results. Thus, the marginal performance advantage of the LSH-based algorithms is obtained at the cost of losing a substantial portion of the results.

Finally, our algorithms have the desirable property that they can be implemented over a regular DBMS using a small amount of application-level code, as we will describe in Section 8.

2. PRELIMINARIES

Formally, a set-similarity join (SSJoin) is specified using a binary predicate $pred$ over sets. It takes as input two set collections, $\mathcal{R}$ and $\mathcal{S}$, and produces as output all pairs (r, s), $r \in \mathcal{R}$, $s \in \mathcal{S}$, such that $pred(r, s)$ evaluates to $true$. Chaudhuri *et al* [6] identified the following general class of predicates for supporting various types of similarity joins in data cleaning:

$$pred(r, s) = \wedge_i (|\, r \cap s\,| \geq e_i)$$

Here, e_i is a numeric expression involving constants and sizes of r and s, i.e., $|\, r\,|$ and $|\, s\,|$. For example, an SSJoin with $pred(r, s) = |\, r \cap s\,| \geq 20$, computes all pairs (r, s) whose intersection is greater than or equal to 20.

For presentation simplicity, we assume that the domain of elements of all sets is $\{1, \ldots, n\}$ for some finite, but possibly large n. In other words, $r \subseteq \{1, \ldots, n\}$ and $s \subseteq \{1, \ldots, n\}$ for each $r \in \mathcal{R}$ and each $s \in \mathcal{S}$. None of our algorithms require the domain of elements be finite and integral, and can be generalized to handle infinite and non-integer domains.

In addition, while our algorithms apply to weighted sets when elements have associated weights, we restrict most of our discussion to unweighted sets. We revisit the weighted case in Section 7.

2.1 Threshold-based SSJoin

For most of this paper, we deal with simpler SSJoins whose predicates involve a set-similarity function and a threshold parameter. Specifically, the predicate $pred(r, s)$ of these threshold-based SSJoins is of the form $Sim(r, s) \geq \gamma$, where Sim denotes a similarity function and γ denotes a threshold parameter, or of the form $Dist(r, s) \leq k$, where $Dist$ denotes a distance function and k a threshold parameter. In particular, we focus on jaccard similarity and hamming distance (defined below), two common functions for defining similarity between sets. Threshold predicates involving each of these similarity functions can be expressed in the more general form introduced above, as we illustrate when we define these functions.

We proceed in this manner for ease of exposition: Our algorithms can be extended to handle a more general subclass of SSJoin predicates, as we describe in Section 6.

2.2 Hamming SSJoin

We can view a set $s \subseteq \{1, \ldots, n\}$ as an n-dimensional binary vector v, such that $v[i] = 1$ if $i \in s$, and $v[i] = 0$, otherwise ($v[i]$ denotes the value of vector v on the ith dimension). The *hamming distance* between two vectors v_1 and v_2, denoted $\mathcal{H}_d(v_1, v_2)$, is the number of dimensions on which the two differ. We often blur the distinction between sets and binary vectors: For example, we refer to the hamming distance between two sets to mean the hamming distance of their vector representations. Note that the hamming distance between two sets s_1 and s_2 is the size of their symmetric difference: $\mathcal{H}_d(s_1, s_2) = |\,(s_1 - s_2) \cup (s_2 - s_1)\,|$.

Example 1. Consider the 3-gram sets of the strings `washington` and `woshington` shown below:

$$s_1 = \{was, ash, shi, hin, ing, ngt, gto, ton\}$$
$$s_2 = \{wos, osh, shi, hin, ing, ngt, gto, ton\}$$

The hamming distance between s_1 and s_2 is 4. □

An SSJoin involving hamming distance (hereafter *hamming SSJoin*) takes as input $\mathcal{R}$ and $\mathcal{S}$ and produces as output all pairs $(r, s) \in \mathcal{R} \times \mathcal{S}$ such that $\mathcal{H}_d(r, s) \leq k$, for some integral threshold k. We note that hamming SSJoins belong to the general class of SSJoins since $\mathcal{H}_d(r, s) \leq k$ is equivalent to:

$$|\, r \cap s\,| \geq \frac{|\, r\,| + |\, s\,| - k}{2}$$

INPUT: Set collections $\mathcal{R}$ and $\mathcal{S}$ and threshold γ

1. For each $r \in \mathcal{R}$, generate signature-set $Sign(r)$
2. For each $s \in \mathcal{S}$, generate signature-set $Sign(s)$
3. Generate all candidate pairs (r, s), $r \in \mathcal{R}, s \in \mathcal{S}$ satisfying $Sign(r) \cap Sign(s) \neq \phi$
4. Output any candidate pair (r, s) satisfying $Sim(r, s) \geq \gamma$.

Figure 2: A signature-based algorithm for SSJoin

2.3 Jaccard SSJoin

The jaccard similarity of two sets r and s, denoted $\mathcal{J}_s(r, s)$, is defined as:

$$\mathcal{J}_s(r, s) = \frac{|\, r \cap s \,|}{|\, r \cup s \,|}$$

$\mathcal{J}_s(r, s)$ is a value between 0 and 1.

Example 2. Consider the sets shown in Example 1. The sets share 6 elements in common, and therefore their jaccard similarity is $6/10 = 0.6$. □

An SSJoin involving jaccard similarity (hereafter *jaccard SSJoin*) takes as input two set collections, $\mathcal{R}$ and $\mathcal{S}$, and produces as output all pairs $(r, s) \in \mathcal{R} \times \mathcal{S}$ such that $\mathcal{J}_s(r, s) \geq \gamma$, where γ denotes a threshold parameter. Again, jaccard SSJoin belongs to the general class of SSJoin introduced above since the predicate $\mathcal{J}_s(r, s) \geq \gamma$ is equivalent to the predicate:

$$|\, r \cap s \,| \geq \frac{\gamma}{1+\gamma}(|\, r \,| + |\, s \,|)$$

3. A FRAMEWORK FOR SSJOIN ALGORITHMS

As mentioned earlier, several algorithms have been proposed for SSJoin in previous work [6, 8, 15, 19, 22]. These algorithms often involve complicated implementation and engineering details, which raises the natural issue of how different algorithms can be compared. Interestingly, most of the previous algorithms have a common high-level structure that also happens to be shared by our algorithms, which can be used for comparison. In this section, we develop a framework based on this common structure, and we use this framework throughout the paper.

Based on their high-level structure, most of the previous algorithms and all of our algorithms can be viewed as belonging to a general class called *signature-based algorithms*. Figure 2 formalizes the steps in a general signature-based algorithm. A signature-based algorithm for SSJoin between $\mathcal{R}$ and $\mathcal{S}$ involving similarity function Sim and threshold γ operates as follows: It first generates a set of *signatures* for each input set in $\mathcal{R} \cup \mathcal{S}$. The signatures have the property that if $Sim(r, s) \geq \gamma$, then r and s share a common signature. Based on this property, the signature-based algorithm generates *candidate pairs* by identifying all $(r, s) \in \mathcal{R} \times \mathcal{S}$ such that the set of signatures of r and s overlap. Finally, in a *post-filtering* step, it checks the similarity join condition $Sim(r, s) \geq \gamma$ for each candidate pair (r, s), and outputs those that satisfy the condition. We emphasize that our view of previous algorithms as signature-based algorithms is purely conceptual. In particular, their original presentation was not along the lines suggested by Figure 2.

Note that Figure 2 provides only a high-level outline of a signature-based algorithm. Several engineering details are left unspecified. For example, a variety of indexing and join techniques can help speed up the generation of candidate pairs [22], and candidate pair generation and postfiltering can often be performed in a pipelined fashion [6]. However, the engineering details are relatively less important for comparing different signature-based algorithms, for two reasons. First, the engineering details are mostly orthogonal to the high-level outline: They do not benefit one high-level approach over another, so they are not very relevant for determining relative performance of signature-based algorithms. Second, the overall performance and the scalability of a signature-based algorithm is primarily determined by parameters such as number of signatures and number of candidate pairs that depend only on the high-level outline of Figure 2, as we will show using our experiments.

The primary difference between various signature-based algorithms lies in their *signature schemes*: the details of how they generate signatures for an input set. Therefore, we focus hereafter mostly on signature schemes. For example, we often refer simply to a signature scheme, while implicitly meaning the signature-based algorithm using the signature scheme. Also when we refer to a signature-based algorithm, we mean at the level of detail of Figure 2. We next introduce some terminology related to signature-based algorithms, present measures for evaluating signature-based algorithms, and briefly review the signature schemes of previous algorithms.

3.1 Signature Scheme

As in Figure 2, we use the notation $Sign(s)$ to denote the signatures generated by a signature scheme for an input set s. Note that this notation does not explicitly identify the signature scheme, which should be clear from the context. Any signature scheme has a basic correctness requirement: For any two sets r and s, $Sign(r) \cap Sign(s) \neq \phi$, whenever $Sim(r, s) \geq \gamma$; here Sim is the SSJoin similarity function and γ is the similarity threshold. This correctness requirement may be satisfied *probabilistically*, which is the case for LSH-based algorithms; an algorithm with such a signature scheme is approximate, i.e., it may miss some output pairs.

The notation $Sign(s)$ is slightly misleading, since the set of signatures for s is not a function of s alone. There are usually several "hidden parameters" which influence the set of signatures for s. These may include the SSJoin threshold γ, statistics collected from $\mathcal{R}$ and $\mathcal{S}$ such as frequency of elements, and random bits used for randomization. When we use $Sign(s)$, the hidden parameters should be clear from the context. Note that the same setting of hidden parameters is used for generating signatures of all input sets.

3.2 Evaluation

We now introduce measures for evaluating signature-based algorithms that depend only on the high-level outline of signature-based algorithms of Figure 2. (Our experiments are based on a complete implementation, and we will report total computation time for our experiments.)

1. *Intermediate result size (F_2):* One good indicator of the overall performance of a signature-based algorithm is given by the following expression:

$$\sum_{r \in \mathcal{R}} |Sign(r)| + \sum_{s \in \mathcal{S}} |Sign(s)| + \sum_{(r,s)} |Sign(r) \cap Sign(s)|$$

If we implement Step 3 of a signature-based algorithm (Figure 2) as a "join" between signatures, the above expression represents the total size of intermediate results produced by the algorithm, which is a well accepted basis for comparison. The first two terms capture the amount of work done for signature generation (steps 1 and 2 of Figure 2), and the third term captures the work done for candidate pair generation (step 3). We will show in Section 8 that the above

expression closely tracks the actual execution time of signature-based algorithms.

Most of the signature schemes that we propose in this paper involve setting various tuning parameters. We can identify the optimal parameters by estimating the value of the above expression for different settings of parameters. Note that for self-SSJoins, the above expression is within a factor 2 of F_2 measure of signatures of all input sets, and there exist well-known techniques for estimating F_2 measure using limited memory [1].

2. *Filtering Effectiveness:* All practical signature schemes produce false positive candidate pairs, i.e., (r, s) such that $Sign(r) \cap Sign(s) \neq \phi$ and $Sim(r, s) < \gamma$. The number of false positive candidate pairs linearly affects the costs of steps 3 and 4 of a signature-based algorithm, therefore it is desirable to minimize it. We use the term *filtering effectiveness* to indicate how good a signature scheme is in minimizing false positives. For most of our signature schemes and for those based on LSH, the filtering effectiveness can be quantified precisely: For any sets r and s, $Sign(r) \cap Sign(s) \neq \phi$ is a random event whose probability is a decreasing function of $Sim(r, s)$. No such quantification of filtering effectiveness exists for the signature schemes of [6, 22].

3.3 Signature schemes of previous algorithms

Probe-Count and Pair-Count Algorithms [22]: The signatures of a set is simply the elements in the set, i.e., $Sign(s) = s$. We call this the *identity* signature scheme.

Prefix-Filter Algorithm [6]: $Sign(s)$ contains the h elements of s with the smallest frequencies[1] in $(\mathcal{R} \cup \mathcal{S})$. The exact value of h depends on the similarity function and threshold. We call this signature scheme *prefix filter*. For illustration, consider a jaccard SSJoin between $\mathcal{R}$ and $\mathcal{S}$ with similarity threshold 0.8. Further, assume that the size of each set is 20, i.e., $|r|=|s|= 20$ for each $r \in \mathcal{R}$ and $s \in \mathcal{S}$. For this case, $Sign(s)$ contains the three elements of s with the smallest frequencies. We can show that if the jaccard similarity of r and s is at least 0.8, then $| r \cap s | \geq 18$, which in turn implies that r and s have at least one signature in common (recall $|r| = |s| = 20$).

LSH-based algorithm [8, 15, 19]: For jaccard SSJoins, each signature is a concatenation of a fixed number g of *minhashes* of the set, and there are l such signatures. The exact definition of minhashes is beyond the scope of this paper. The values g and l control the performance of the algorithm: Informally, the parameter g controls the filtering effectiveness, and for a fixed value of g, the parameter l controls the approximation factor: In order to achieve a false negative rate of δ, we require about $l = \frac{1}{\gamma^g} \log(\frac{1}{\delta})$ signatures.

4. HAMMING DISTANCE

We now present a signature scheme called PARTENUM (for *Partitioning and Enumeration*) for hamming SSJoins. In Section 5, we use PARTENUM as a building block to derive a signature scheme for jaccard SSJoins. Throughout this section, we represent sets as binary vectors, as indicated in Section 2. Therefore, we will focus on the following vector-based join problem in the remainder of this section: Given two vector collections $\mathcal{U}$ and $\mathcal{V}$, identify all pairs $(u, v) \in \mathcal{U} \times \mathcal{V}$ such that $\mathcal{H}_d(u, v) \leq k$. Recall that we have assumed $\{1, \ldots, n\}$ to be the domain of elements, so all vectors $u \in \mathcal{U}$ and $v \in \mathcal{V}$ are n-dimensional.

4.1 PartEnum: Overview

As the name suggests, PARTENUM is based on two ideas for signature generation called *partitioning* and *enumeration*. In this section, we informally introduce these ideas and PARTENUM; detailed specification of PARTENUM is provided in Section 4.2.

Partitioning: Consider a partitioning of the dimensions $\{1, \ldots, n\}$ into $k + 1$ equi-sized partitions. Any two vectors that have a hamming distance $\leq k$ must agree on at least one of these partitions, since the dimensions on which they disagree can fall into at most k partitions. Based on this observation, we can construct a simple signature scheme as follows: For each vector v, generate $k + 1$ signatures by projecting v along each of the $k + 1$ partitions. Unfortunately, this scheme has poor filtering effectiveness since two vectors often end up "accidentally" agreeing on a partition even if they are very dissimilar, and therefore end up sharing a signature.

Enumeration: More generally, consider a partitioning of the dimensions into $n_2 > k$ equi-sized partitions. Any two vectors with hamming distance $\leq k$ must agree on at least $(n_2 - k)$ partitions. Using this observation, we can construct the following signature scheme: For each vector v, pick $(n_2 - k)$ partitions in every possible way, and for each selection, generate a signature by projecting v along these $(n_2 - k)$ partitions. (There are $\binom{n_2}{k}$ signatures for each vector v.) This scheme has very good filtering effectiveness if we set $n_2 \approx 2k$, but the drawback is that it generates around $\binom{2k}{k} \approx 2^{2k}$ signatures per vector for this setting.

Hybrid(PARTENUM): Now consider a partitioning of the domain into $n_1 = (k + 1)/2$ partitions. Consider two vectors u and v with $\mathcal{H}_d(u, v) \leq k$. Using a simple counting argument, we can show that the projections of u and v along at least one of these partitions have hamming distance ≤ 1. Using this observation, we generate a signature scheme as follows: We project a given vector v along each of the n_1 partitions. For each projection, we generate a set of signatures using the enumeration scheme with a new threshold $k_2 = 1$. The signature set for v is the union of signatures corresponding to all projections. Informally, partitioning reduces the problem of generating signatures for a vector to that of generating signatures for vectors of smaller dimensions and for a smaller threshold; for a smaller threshold the number of signatures generated by enumerations becomes more tractable.

4.2 PartEnum: Details

Figure 3 contains the formal specification of PARTENUM. We first generate a random permutation π of the dimensions $\{1, \ldots, n\}$. We use π to define a two-level partitioning of the dimensions: There are n_1 first-level partitions, and within each first-level partition, there are n_2 second-level partitions. Therefore, there are $n_1 \times n_2$ second-level partitions overall. The values n_1 and n_2 are parameters that can be varied to control signature generation by PARTENUM. Each (first- or second-level) partition contains a set of dimensions contiguous under permutation π, i.e., it is of the form $\{\pi(i) : b \leq i < e\}$. We use p_{ij} to denote the jth second-level partition within the ith first level partition; see Figure 3 for the formal definition of p_{ij}. The random permutation π, and therefore the partitioning, is generated only once. The signatures of all vectors in $(\mathcal{U} \cup \mathcal{V})$ are generated using the same partitioning of the dimensions.

We now describe how the signatures for a vector v are generated. Fix $k_2 = \frac{k+1}{n_1} - 1$; recall that k is the hamming distance threshold. For each first-level partition, we generate all possible subsets of second-level partitions of size $(n_2 - k_2)$—there are exactly $\binom{n_2}{k_2}$ such subsets. We generate one signature corresponding to each subset. Fix a subset S, and let P denote the set of all dimensions belonging to partitions in S. The signature for S is the pair $\langle v[P], P \rangle$,

[1] Ties are broken arbitrarily but consistently for all sets.

PARAMETERS:
  SSJoin threshold: k
  Number of first-level partitions: n_1, $n_1 \leq k+1$
  Number of second-level partitions: n_2, $n_1 n_2 > k+1$
ONETIME:
  1. Generate a random permutation π of $\{1, \ldots, n\}$
  2. Define $b_{ij} = \frac{n(n_2 i - 1 + j - 1)}{n_1 n_2}$; $e_{ij} = \frac{n(n_2 i - 1 + j)}{n_1 n_2}$
  3. Define $p_{ij} = \{\pi(b_{ij}), \pi(b_{ij}+1), \ldots, \pi(e_{ij}-1)\}$
  4. Define $k_2 = \frac{k+1}{n_1} - 1$
SIGNATURE FOR v:
  1. $Sign(v) = \phi$
  2. For each i in $\{1, \ldots, n_1\}$
  3.   For each subset S of $\{1, \ldots, n_2\}$ of size $n_2 - k_2$
  4.   Let $P = \cup_{j \in S} p_{ij}$
  5. $Sign(v) = Sign(v) + \langle v[P], P \rangle$

Figure 3: PARTENUM: Formal Specification

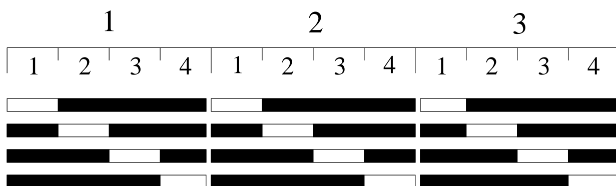


Figure 4: PARTENUM with $n_1 = 3$ and $n_2 = 4$ and $k = 5$

where $v[P]$ denotes the projection of vector v along dimensions in P. (For example, if $v = \mathbf{01}0101\mathbf{0}1$, $v[\{1, 2, 8\}] = 011$.) Note that two signatures $\langle v_1[P_1], P_1\rangle$ and $\langle v_2[P_2], P_2\rangle$ are equal only if both the projections ($v_1[P_1] = v_2[P_2]$) and the subsets ($P_1 = P_2$) are equal. The total number of signatures for v is therefore $n_1 \cdot \binom{n_2}{k_2}$. We will address practical issues concerning signature generation shortly.

Example 3. Figure 4 illustrates signature generation for $n_1 = 3$, $n_2 = 4$, and $k = 5$. By definition, $k_2 = 1$. For each first-level partition, we generate one signature corresponding to every second-level partition subset of size 3. Therefore there are $3 \times 4 = 12$ signatures for every vector. These signatures are represented as horizontal rectangles in Figure 4. The darkened portion of a rectangle indicates the dimensions along which a vector is projected for generating the corresponding signature. Note that the dimensions are ordered according to the permutation π. □

Example 4. Let $n_1 = 2$, $n_2 = 3$, and $k = 3$, implying $k_2 = 1$. Assume for simplicity that π is the identity permutation, i.e., $\pi(i) = i$, for all i. Figure 5 shows the six signatures for the vector 010110. □

The following theorem states the correctness of PARTENUM without proof.

THEOREM 1. *If $\mathcal{H}_d(u, v) \leq k$, then $Sign(u) \cap Sign(v) \neq \phi$, where $Sign(u)$ and $Sign(v)$ are generated using the same random permutation π, and same parameters n_1, n_2, and k.*

Practical Issues: Since our vectors are representations of sets, they are typically sparse with a large dimensionality n. Therefore, generating signatures directly as suggested by Figure 3 is not efficient. Consider a signature $\langle v[P], P\rangle$ corresponding to a set S, generated in line 5 of Figure 3. In order to compute $v[P]$, we need to project v along the dimensions in P, which could be potentially

$\langle 10, \{2,3\}\rangle$	$\langle 00, \{1,3\}\rangle$	$\langle 01, \{1,2\}\rangle$
$\langle 10, \{5,6\}\rangle$	$\langle 10, \{4,6\}\rangle$	$\langle 11, \{4,5\}\rangle$

Figure 5: Signatures for 010110; $n_1 = 2$, $n_2 = 3$, $k = 3$

large in number. Instead, we could encode the signature using $\langle P_1(v), i, S\rangle$, where $P_1(v)$ denotes the set of dimensions in P for which v has a value 1, i.e., $\{d \in P : v[d] = 1\}$. Note that $P_1(v)$ uniquely encodes $v[P]$ and i and S uniquely identify P, and we can compute $P_1(v)$ more efficiently than $v[P]$ when v is sparse. Further, since the only operation that we perform on signatures is checking equality, we can simply hash these signatures into 4 byte values. Note that hash collisions do not affect the correctness of PARTENUM—Theorem 1 continues to be valid for hashed signatures—but the collisions could introduce additional false positive candidate pairs. In practice, the number of such false positives in negligible.

4.3 PartEnum: Performance

In this section, we cover various aspects of PARTENUM's performance. We begin by proving that PARTENUM has good asymptotic performance: For a particular setting of n_1 and n_2, we can prove that it provides good filtering effectiveness (i.e., generates only a few false positive candidate pairs) with few signatures per input set.

THEOREM 2. *Consider* PARTENUM *with $n_1 = k/\ln k$ and $n_2 = 2\ln k$. If $\mathcal{H}_d(u, v) > 7.5k$, then $Sign(u) \cap Sign(v) = \phi$ with probability $1 - o(1)$. For this setting of parameters, the number of signatures per vector is $O(k^{2.39})$.*

The constants in the statement of Theorem 2 are merely representative. We can get slightly different performance characteristics by suitably changing n_1 and n_2 values. Theorem 2 shows that we can achieve good filtering using $O(k^{2.39})$ signatures, which is independent of n. This property is crucial in order to be able to handle sparse vectors—which is the case when the vectors are representations of sets—where n could be very large.

Note that we do not recommend using the parameters of Theorem 2 for actual SSJoins. In fact, we will show in Section 8 that no single setting of the parameters is good for all SSJoin instances. For a fixed setting of parameters, we can show that increasing the number of input sets results in a quadratic increase in the number of signature collisions; therefore PARTENUM for a fixed setting of parameter scales only quadratically. The fact that we can control the behavior of PARTENUM using the parameters is actually crucial to ensure near-linear scalability of PARTENUM with increasing input size. Specifically, PARTENUM offers a tradeoff between the number of signatures per set and the filtering effectiveness: We can increase the number of signatures and improve filtering effectiveness, and vice-versa. The number of signatures can be increased either by decreasing the value of n_1 or increasing the value of $(n_2 - k_2)$. When the input size increases, we can avoid the quadratic increase in computation time by using a larger number of signatures per set. We cover this aspect in detail in Section 8. An important issue is determining the optimal setting of (n_1, n_2): Informally, we can determine optimal settings using some properties of the input data. This is covered again in Section 8.

5. JACCARD SIMILARITY

We now describe how we can use PARTENUM for jaccard SSJoins. An easy special case occurs when all input sets are equisized: Two sets can have jaccard similarity $\geq \gamma$ iff their hamming

OFFLINE:

(a) Define $I_1 = [l_0, r_0] = [1, 1]$

(b) Define $I_i = [l_i, r_i]$, where $l_i = r_{i-1} + 1$, $r_i = \left\lfloor \frac{l_i}{\gamma} \right\rfloor$

(c) Define $k_i = \left\lfloor 2\left(\frac{1-\gamma}{1+\gamma}\right) r_i \right\rfloor$

(d) For each $i = 1, 2, \ldots$, construct an instance of PARTENUM, denoted PE[i], with threshold k_i.

SIGNATURE FOR s:

1. Initialize $Sign(s) = \phi$
2. Let I_i denote the interval to which $|s|$ belongs.
3. For each signature $sg \in$ PE[i].$Sign(s)$
4. $\quad Sign(s) = Sign(s) + \langle i, sg \rangle$
5. For each signature $sg \in$ PE[$i+1$].$Sign(s)$
6. $\quad Sign(s) = Sign(s) + \langle i+1, sg \rangle$
7. Return $Sign(s)$.

Figure 6: Signature Scheme for Jaccard Similarity

distance is $\leq \frac{2\ell(1-\gamma)}{1+\gamma}$, where ℓ denotes the common set size. We can use this observation to convert a jaccard SSJoin with threshold γ to an equivalent hamming SSJoin with threshold $\frac{2\ell(1-\gamma)}{1+\gamma}$, and use PARTENUM for the latter.

For the general case, we observe that if two sets have high jaccard similarity, then they should have roughly similar sizes, as formalized in Lemma 1 below.

LEMMA 1. *If $\mathcal{J}_s(r, s) \geq \gamma$, then $\gamma \leq \frac{|r|}{|s|} \leq \frac{1}{\gamma}$.*

We use this observation to (conceptually) divide a general jaccard SSJoin instance into smaller instances, each of which computes an SSJoin on sets of roughly equal size. We use the technique described above to convert the smaller jaccard SSJoin instances to hamming SSJoin instances, and use PARTENUM for signature generation.

Formally, consider a jaccard SSJoin with threshold γ between $\mathcal{R}$ and $\mathcal{S}$. We define intervals I_i $(i = 1, 2, \ldots)$ that partition the set of positive integers (steps (a) and (b) of Figure 6). For each interval $I_i = [l_i, r_i]$, the right end of the interval $r_i = \left\lfloor \frac{l_i}{\gamma} \right\rfloor$. Using Lemma 1, we can show that if $|\,s\,| \in I_i$ and $\mathcal{J}_s(r, s) \geq \gamma$, then $|r| \in I_{i-1} \cup I_i \cup I_{i+1}$. Based on this observation, we (conceptually) construct subsets $\mathcal{R}_1, \mathcal{R}_2, \ldots$ of $\mathcal{R}$ as follows: For each $r \in \mathcal{R}$, if $|\,r\,| \in I_i$, we add the set r to both $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$. We construct subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots$ of $\mathcal{S}$, symmetrically. Then we perform a hamming SSJoin between each $\mathcal{R}_i$ and $\mathcal{S}_i$ with threshold $k_i = 2\left(\frac{1-\gamma}{1+\gamma}\right) r_i$, and take the duplicate-free union of all the SSJoin outputs. The correctness of this approach follows from the observation that if $\mathcal{J}_s(r, s) \geq \gamma$, and $r \in \mathcal{R}_i$, $s \in \mathcal{S}_i$, then we can show that $\mathcal{H}_d(r, s) \leq \frac{1-\gamma}{1+\gamma}(|r| + |s|) \leq 2\left(\frac{1-\gamma}{1+\gamma}\right) r_i = k_i$.

Example 5. Let $\gamma = 0.9$. Then we can verify that $I_1 = [1, 1]$, $I_8 = [8, 8]$, $I_9 = [9, 10]$, $I_{13} = [17, 18]$, and $I_{14} = [19, 21]$. Assume $\mathcal{R} = \{r_9, r_{10}, \ldots, r_{21}\}$, where the subscripts encode the size of a set, i.e., $|\,r_i\,| = i$. Similarly, assume that $\mathcal{S} = \{s_9, \ldots, s_{21}\}$. Figure 7 shows the composition of the subsets $\mathcal{R}_{10}$–$\mathcal{R}_{14}$; the squares from left-to-right represent the inputs $r_9, \ldots, r_{21}$, and the shaded rectangles represent the subsets $\mathcal{R}_i$. For example, $\mathcal{R}_{14} = \{r_{17}, \ldots, r_{21}\}$. Note that under our approach many input pairs in $\mathcal{R} \times \mathcal{S}$ are never considered for a join: For example, r_{10} which belongs to $\mathcal{R}_9$ and $\mathcal{R}_{10}$ is never considered for a join with s_{13} which belongs to $\mathcal{S}_{11}$ and $\mathcal{S}_{12}$, and correctly so, since we can infer using Lemma 1 that $\mathcal{J}_s(r_{10}, s_{13}) \leq \frac{10}{13} \approx 0.76$. □

Instead of explicitly constructing the subsets $\mathcal{R}_i$ and $\mathcal{S}_i$ and then computing the SSJoin between $\mathcal{R}_i$ and $\mathcal{S}_i$, we can get the same effect by computing the SSJoin directly between $\mathcal{R}$ and $\mathcal{S}$ using

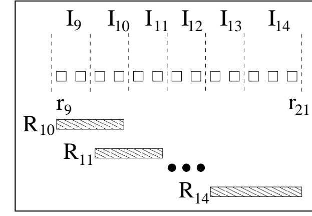


Figure 7: Size-based filtering (Example 5)

the signature scheme shown in Figure 6. Essentially, the signature scheme attaches the number i to the signatures of $\mathcal{R}_i$ and $\mathcal{S}_i$, which ensures that signatures of $\mathcal{R}_i$ and $\mathcal{S}_j$ $(i \neq j)$ do not collide. We call the signature scheme of Figure 6 also as PARTENUM, and it should be clear from the context whether we are referring to PARTENUM for hamming (Figure 3) or PARTENUM for jaccard (Figure 6).

We call this approach of using size information to reduce the number of set pairs that need to be considered as *size-based filtering*. Size-based filtering is not specific to PARTENUM: It can be combined with any other signature scheme to improve the performance of the scheme. (However, for the case of PARTENUM, it is essential in order to be make PARTENUM work for jaccard SSJoins.)

6. GENERAL SSJOINS

Recall from Section 2 that we defined SSJoins for a general class of predicates involving the sizes of sets and the size of their intersection. The main ideas behind our signature scheme for jaccard SSJoins can be generalized to derive a signature scheme for a large subclass of the general class of predicates. Informally, our techniques can be used to handle an SSJoin with predicate *pred* if:

1. For any set r, we can derive an upper- and lower-bound on the size $|\,s\,|$ of sets s that join with r, i.e., $pred(r, s)$ evaluates to $true$, and

2. For any set r, we can derive an upper-bound on the hamming distance between r and s for any s that joins with r.

The first condition helps us conceptually partition the given SSJoin instance into smaller instances as we did for jaccard SSJoins, and the second condition enables us to use a hamming PARTENUM for each of the smaller instances. For example, an SSJoin with predicate $|\,r \cap s\,| \geq 0.9 \max\{|\,r\,|, |\,s\,|\}$ satisfies both conditions: Given a set r with size 100, we can show that only sets s with sizes between 90 and 111 can possibly join with r, and further, for any s that joins with r, $\mathcal{H}_d(r, s) \leq 20$. On the other hand, an SSJoin with predicate $|\,r \cap s\,| \geq 20$ satisfies neither condition. Formalizing the exact class of predicates that satisfy the above requirements is beyond the scope of this paper.

7. WEIGHTED SSJOIN

We now consider the weighted version of SSJoin, where there is a weight $w(e)$ associated with each element e. For many applications using set-similarity, some elements are more important than others for the purpose of defining similarity, and these differences in the importance are naturally captured using element weights. A well-known example is the use of weights based on *inverse document frequency (IDF)* in Information Retrieval [3], which essentially captures the intuition that less frequent words are more significant than frequent words for determining document similarities.

We can use PARTENUM for the weighted case by converting a weighted SSJoin instance to an unweighted one: We convert a weighted set into an unweighted bag by making $w(e)$ copies of

PARAMETERS:
SSJoin threshold: T
Pruning threshold: TH
SIGNATURE FOR s:
1. $Sign(v) = \phi$
2. For each minimal subset s' of s with weighted size $\geq T$
3. Order elements of s' in decreasing order of IDF weights.
4. Define PREF(s'): smallest prefix of s', such that the sum element weights in the prefix >= TH
5. $Sign(v) = Sign(v) \cup \{\text{PREF}(s')\}$.
6. Remove duplicates from $Sign(v)$ and return.

Figure 8: WTENUM: Formal Specification

each element e, using standard rounding techniques if weights are nonintegral. All of our guarantees from Section 4 continue to hold for this approach. However, this approach is unsatisfactory for the following reason: Consider an unweighted hamming SSJoin with threshold k. Theorem 2 states that using $O(k^{2.39})$ signatures we can achieve good filtering. Next, consider a weighted hamming SSJoin with threshold αk, where all element weights are α. Clearly, the weighted SSJoin is identical to the unweighted one above. However, if we use the approach above, PARTENUM requires $O(\alpha^{2.39} k^{2.39})$ signatures for (almost) the same filtering effectiveness, which is undesirable since α can be made arbitrarily large. This example suggests that the theoretical guarantees of Theorem 2 are not very meaningful for the weighted case. We do not know what a good theoretical guarantee for weighted case should look like for exact SSJoin computation.

We now present a heuristic signature scheme called WTENUM (for *Weighted Enumeration*) that works well in practice. It is convenient to present WTENUM for *intersection* SSJoins: Given $\mathcal{R}$ and $\mathcal{S}$, identify all pairs $(r, s) \in \mathcal{R} \times \mathcal{S}$ such that $| r \cap s | \geq T$, for a threshold T. We can adapt WTENUM for jaccard SSJoins using ideas from Section 5. We initially assume that weights are IDF-based; we later describe how WTENUM can be extended for general weights.

Figure 8 contains the formal specification of WTENUM. WTENUM generates signatures for a set s as follows: It conceptually enumerates all *minimal* subsets s' of s with weighted size at least T. A subset s' is minimal if no proper subset of s' has weighted size $\geq T$. For each subset s', it orders the elements in descending order of weights, and picks the smallest prefix of this ordering such that the weights of the elements in the prefix add upto at least TH; TH is a parameter that can be used to control WTENUM. The signature set for s' is the set of all such distinct prefixes. (As before, we can hash the prefixes into integer values.) The correctness of WTENUM is straightforward: if $| r \cap s | \geq T$ for some r and s, then r and s share some minimal subset; the prefix of this subset is a common signature for both r and s.

Example 6. Consider the weighted set $s = \{a_8, b_4, c_3, d_2, e_1, f_1, g_1\}$, where the subscripts indicate the IDF weights of elements. Let the intersection SSJoin threshold be 17, and let the TH parameter be 14. Figure 9 shows the minimal subsets of s considered in Step 2 and the prefixes for each minimal subset. The final signature set for s is $\{\langle a, b, d\rangle, \langle a, b, c\rangle\}$. Any set that has a weighted intersection of 17 with s has to contain both a and b and at least one of c or d, and therefore shares a signature with s. □

The intuition behind WTENUM is simple: Recall that the IDF weight of an element is defined as $\log(1/f_e)$, where f_e denotes the fraction of input sets in which e occurs. Therefore, if TH $= \log \max\{| \mathcal{R} |, | \mathcal{S} |\}$, any subset of elements whose weights add up to TH occurs in only one input set in expectation, if we assume

Minimal Subset	Prefix
$\{a, b, d, e, f, g\}$	$\langle a, b, d\rangle$
$\{a, b, c, d\}$	$\langle a, b, c\rangle$
$\{a, b, c, e, f\}$	$\langle a, b, c\rangle$
$\{a, b, c, e, g\}$	$\langle a, b, c\rangle$
$\{a, b, c, g, f\}$	$\langle a, b, c\rangle$

Figure 9: Example signature generation using WTENUM

elements occur independently in the input sets. Therefore, using TH $= \log \max\{|\mathcal{R}|, |\mathcal{S}|\}$ produces very few signature collisions. The number of signatures per set could possibly be undesirably high, but it is usually very small in practice. Finally, when the weights are non-IDF, we explicitly generate IDF weights for elements; we use the non-IDF weights in Step 2 and the IDF weights in Step 3 of Figure 8.

8. EXPERIMENTS

We now present our experimental results. The main goal of our experiments is to measure the performance of our algorithms and to compare the performance against those of previous algorithms. Our experiments covered jaccard SSJoins, weighted jaccard SSJoins, and string similarity joins involving edit distance. Edit-distance based string similarity joins use hamming SSJoins as an underlying primitive, and therefore indirectly measure the performance of the algorithms for this type of SSJoin. All of our experiments involved only self-joins; we expect the relative performances to be similar for binary SSJoins as well.

The details of the various algorithms used in our experiments are as follows:

1. LSH: We used the classic LSH based on minhashes for our experiments involving jaccard and weighted jaccard SSJoins. LSH does not map naturally to the edit distance measure, so we did not include LSH in our experiments involving edit-distance based string similarity joins. In most of the experiments, we used LSH with accuracy (false-negative rate) 0.95. Note that this means LSH produces only about 95% of the correct output. When we refer to LSH instances with a different false-negative rate, we explicitly quantify the rate within paranthesis; for example, LSH(0.90) refers to an instance of LSH with false-negative rate 0.90. In all of our experiments involving LSH, we used the optimal settings of parameters g and l (recall Section 3) for the given accuracy. The observed accuracy of LSH in all our experiments was very close to the predicted accuracy, so we do not report these numbers explicitly.

2. Prefix Filter: Prefix filter represents the best previous exact algorithm. The performance of the original prefix filter as proposed in [6] was very poor relative to LSH and our algorithms. Therefore, we augmented it with size-based filtering of Section 5. We report experimental results only for this augmented version of prefix filter. In our experimental charts, we abbreviate prefix filter to PF to save space.

3. PARTENUM, WTENUM: Like in LSH, we used the optimal settings of parameters for PARTENUM and WTENUM in our experiments. In our experimental charts, we abbreviate PARTENUM to PEN and WTENUM to WEN to save space.

Our discussion so far has focused mostly on a high-level view of signature-based algorithms, comprising of Steps 1–4 shown in Figure 2. In particular, we did not deal with the implementation of these steps. For our experiments, we use an implementation that uses a general purpose DBMS for the most part, with a small amount of application-level code. (We will describe the details,

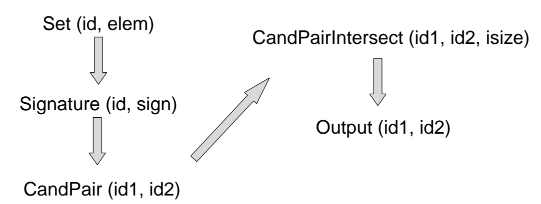


Figure 10: Jaccard SSJoin implementation

which are specific to the type of SSJoin, later.) Other implementations are possible: However, we measured various implementation independent performance measures such as F_2 (recall from Section 3) size of signatures, number of candidate pairs, and so on, and these strongly suggest that the relative performance of various algorithms will not change with a different implementation.

All of our experiments were performed on a 3.2 GHz Pentium 4 machine with 2GB RAM running Windows Server 2003. We used Microsoft SQL Server 2005 for database support.

8.1 Jaccard Similarity

We first describe our implementation of jaccard SSJoins, and then present experiment results for real and synthetic datasets.

Implementation

Figure 10 shows the implementation that we use for computing jaccard SSJoins. The input is provided as a relation `Set(id, elem)` in first normal form; `id` denotes the identifier for a set and `elem` denotes an element belonging to the set. In the first step, we scan the relation `Set`, generate signatures for each input set using an appropriate signature scheme, and generate a new relation `Signature(id, sign)` containing the signatures. We perform this step using application-level code, so data crosses DBMS boundaries during both the reading and the writing. The remaining steps are all performed within the DBMS using SQL queries. We next generate the candidate pairs by performing a self-join over the signature relation, and these are stored in a relation `CandPair(id1,id2)`; here `id1` and `id2` denote the identifiers of a candidate pair. The postfiltering step, where we check the SSJoin predicate, is performed using two queries. The first query computes the intersection size of each candidate pair by performing a join with the input relation `Set`, and stores the result in the table `CandPairIntersect(id1,id2,isize)`. The second query joins `CandPairIntersect` with another relation `SetLen(id,len)` containing the size of each input set, and checks the SSJoin predicate. For the purpose of our experiments, we materialize this relation in advance, so the time required to compute it is not factored in our results. The actual DBMS queries used in the implementation are shown in Figure 11. We built a clustered index over the input relation `Set` since it significantly improved the time to computer `CandPairIntersect`. We construct the index in advance, so the index construction time is not factored in our experimental results. No other index was used in our implementation. We used 32 bit integers for all the columns, with appropriate hashing wherever necessary.

Experiments on real data sets

We performed experiments on two real data sets: the first is a proprietary address data set and the second is the publicly available DBLP data set. The address data set contains 1 million strings, each a concatenation of an organization name and address (street, city, zip, state). The DBLP data set contains around 0.5 million strings, each a concatenation of authors and title of a publication. To generate sets, we tokenized the strings based on white space separators, and hashed the resulting words into 32 bit integers. The average size of a set in the address data is 11, while that in the DBLP data is 14. Since the results for both datasets were similar qualitatively, we only report results for the address data. Note that both data sets are highly relevant for the data cleaning applications that we are interested in. We used different sized (100K, 500K, and 1M) subsets of the address data as input to the SSJoin to understand the scalability properties of different algorithms. (Input size refers to the number of input sets to SSJoin—recall we are dealing only with self-joins here.)

```
CandPair (id1, id2):
   Select Distinct S1.id as id1, S2.id as id2
   From Signature as S1, Signature as S2
   Where S1.Sign = S2.Sign and S1.id < S2.id

CandPairIntersect (id1, id2, isize):
   Select C.id1, C.id2, Count(*) as isize
   From CandPair as C, Set as S1, Set as S2
   Where C.id1 = S1.id and C.id2 = S2.id and S1.elem = S2.elem
   Group By C.id1, C.id2

Output (id1, id2):
   Select C.id1, C.id2
   From CandPairIntersect as C, SetLen as S1, SetLen as S2
   Where C.id1 = S1.id and C.id2 = S2.id and
         C.isize ≥ (S1.len + S2.len − C.isize) ∗γ
```

Figure 11: Jaccard SSJoin Implementation: Queries

Figure 12 shows the total SSJoin computation time for the three algorithms for different input sizes and similarity thresholds. The results indicate that the performance of PARTENUM and LSH are roughly similar for all input sizes and similarity thresholds that we considered. The performance of PARTENUM is actually slightly better for 0.9 and 0.85 similarity thresholds. This happens because PARTENUM uses size information to ensure that two sets with very different sizes do not share a signature, while LSH does not. As we indicated in Section 4, the performance of PARTENUM falls steeply with decreasing similarity, and this is reflected in Figure 12: the LSH has slightly better performance than PARTENUM at similarity threshold 0.8.

The results also indicate that the gap between prefix filter and the other two increases sharply with increasing input size. Since the scales used for subfigures (a), (b), and (c) are different, the scalability aspects of different algorithms are not immediately obvious from Figure 12. In fact, the scalability of prefix filter is almost quadratically, while that PARTENUM and LSH is almost linear. For example, at 0.85 similarity, when we move from 100K input size to 1M input size, the computation time for PARTENUM increases from 23 seconds to 240 seconds (a tenfold increase), while that for prefix filter increases from 36 seconds to about 2500 seconds (a 70 fold increase).

Figure 13 shows the F_2 size of signatures for the three algorithms. The F_2 values closely track the actual running times, indicating that the observed relative performance is not specific to our implementation. In all our experiments, the setting of parameters for PARTENUM and LSH for optimizing F_2 was identical to the setting of parameters for optimizing the actual running time. This suggests that we can use well-known techniques for F_2 estimation for automatically determining the optimal setting of parameters for PARTENUM and LSH.

Experiments on synthetic data sets

We performed a variety of experiments using synthetic data, and we report a representative one here. As part of this experiment, we

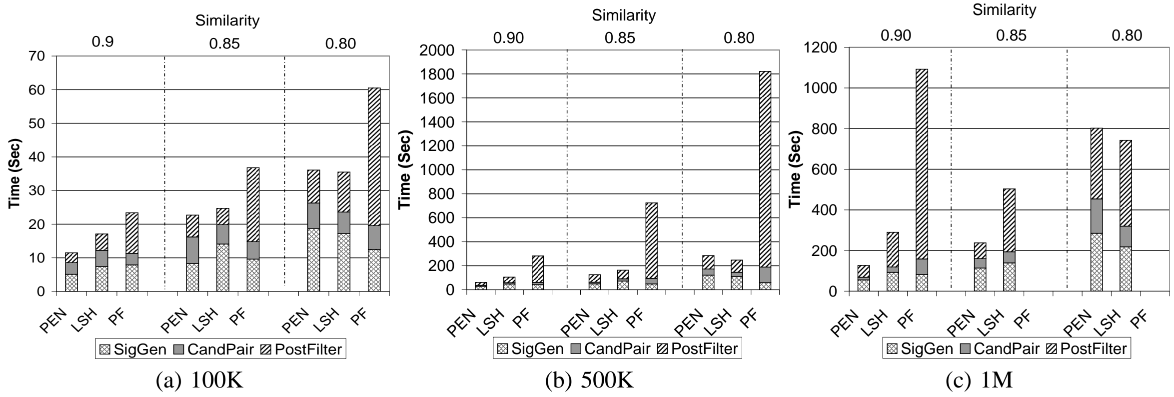


Figure 12: Total jaccard SSJoin computation time for address data

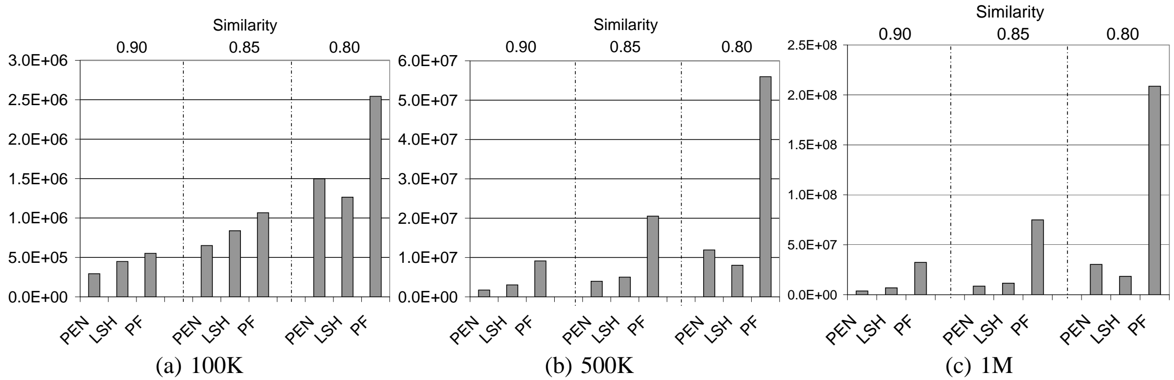


Figure 13: Jaccard SSJoin F_2 size for address data

generated synthetic data comprising of sets with the same size. This means that we no longer require the size-based filtering of Section 5 for PARTENUM, and therefore this experiment serves to better illustrate the scalability of different algorithms without the effects of partitioning. Further, note that PARTENUM cannot get any filtering effectiveness from the varying set sizes, and so does not get any "unfair" advantage over LSH. We generated input sets with 50 elements each; the elements of each set were drawn uniformly at random from a domain of size 10000 elements. We added a few additional sets highly similar to existing ones to generate valid output. Our data generation is similar to the one used in [8]. Again, we measured the performance of PARTENUM, LSH, and prefix filter for varying input sizes (50K, 100K, 500K, 1M), and for varying similarity.

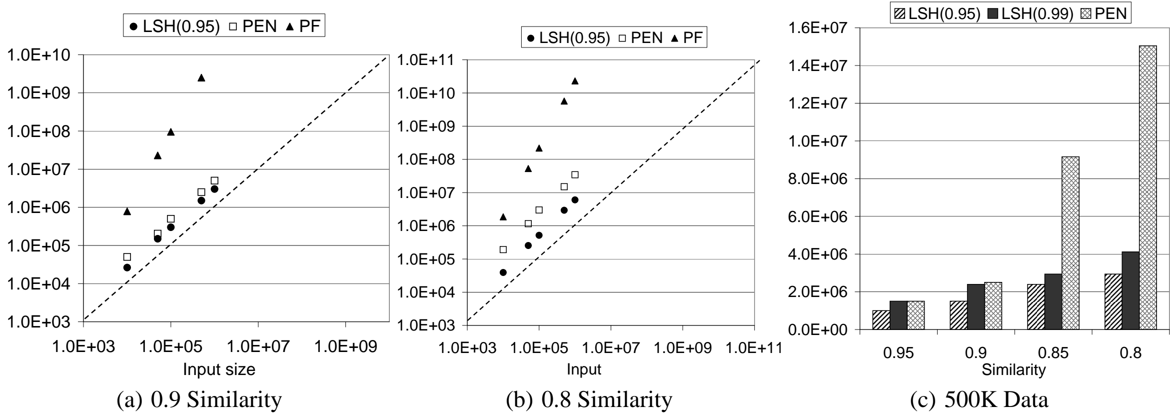
Overall the results for this data were qualitatively similar to the results for real data presented above. LSH is now slightly faster than PARTENUM (1.5x for 0.9 similarity and 5x for 0.8 similarity). We present some results from this experiment in a slightly different format to highlight aspects not illustrated in the earlier experiment. Figures 14(a) and (b) present the F_2 measure (y-axis) for the three algorithms for SSJoins with 0.9 and 0.8 similarity, respectively, for varying input sizes (x-axis). Both axes are in logarithmic scale, meaning that the F_2 vs. input size plot for a perfectly scaling algorithm would be a straightline with slope 1 (parallel to the dotted line in the figures). Figures 14(a) and (b) show that this is indeed the case for PARTENUM and LSH. The F_2 vs. input size slope for prefix filter is almost 2, illustrating that it scales nearly quadratically with input size. Finally, Figure 14(c) plots the F_2 measure (y-axis) of PARTENUM and LSH for varying similarity thresholds. We use LSH with two different accuracy settings: 0.95 and 0.99.

Input Size	Optimal (n1, n2)	Num. of signatures/set
10K	(9,3)	13
50K	(6,3)	16
100K	(4,4)	22
500K	(4,4)	22
1M	(3,5)	30

Table 1: Optimal PARTENUM parameters vs. input size. Similarity threshold: 0.8

We do not plot the performance of prefix filter, in order to more accurately bring out the contrast between LSH and PARTENUM.

The main reason for the near-linear scalability of PARTENUM is the availability of control parameter n_1 and n_2. For a *fixed setting* of parameters, PARTENUM has quadratic scaling: increasing input size causes a quadratic increase in the number of signature collisions. However, we are able to avoid the quadratic increase by moving to a different setting of parameters that generates more signatures per input set, and therefore has better filtering effectiveness. Table 1 illustrates this argument: It shows the optimal setting of (n_1, n_2) for varying input sizes of our synthetic data (for 0.80 similarity threshold). In general, we can increase the number of signatures and improve filtering effectiveness by reducing n_1 or increasing $(n_2 - k_2)$ or both. Figure 15 illustrates the tradeoff between the number of signatures and filtering effectiveness: for varying values of n_1, keeping $(n_2 - k_2)$ constant, we plot the total number of signatures corresponding to all input sets and total number of signature collisions, which is essentially F_2 minus the total number of signatures.



(a) 0.9 Similarity (b) 0.8 Similarity (c) 500K Data

Figure 14: Jaccard SSJoin: Synthetic data

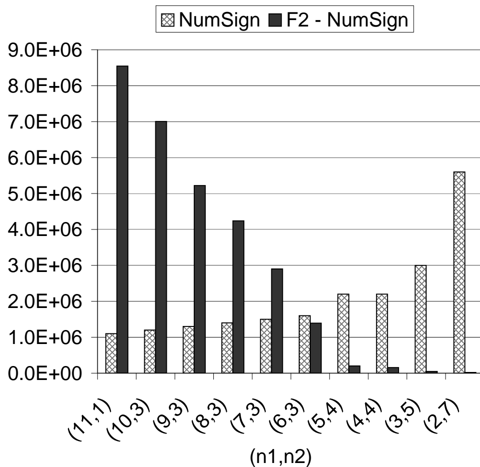


Figure 15: Tradeoff between number of signatures and filtering effectiveness

8.2 Edit Distance

As we mentioned in Section 1, one of the main uses of SSJoin is as a primitive for string similarity joins. In this section, we report on our experiments on string similarity joins using edit distance, which is one of the most common distance functions for strings. We provide brief background on string similarity joins, and then describe our implementation, before presenting our experimental results.

Background

The basic idea behind using SSJoins for string similarity joins is the observation that if two strings have small edit distance, then their n-gram sets are similar. Specifically, if the edit distance between strings s_1 and s_2 is $\leq k$, then we can show that the hamming distance of their n-gram sets $\leq nk$. Therefore, in order to compute a string similarity join with edit threshold k, we first generate n-gram sets (bags) for each string, and then compute an SSJoin with hamming threshold nk. The output of the SSJoin can contain false positives, since the n-gram sets of two strings can have a hamming distance $\leq nk$, even if the edit distance of the strings is $> k$. These false positives are removed using a postprocessing step.

One interesting issue is the choice of n, the n-gram value. Increasing the value of n, results in a weaker SSJoin threshold, nk, making the SSJoin harder. On the other hand, a smaller value of n, means that the elements of the SSJoin input sets are drawn from a smaller domain; as we indicated earlier, previous exact algorithms [6, 14] perform poorly with smaller element domains, since their signatures are drawn from the domain of elements. Interestingly, small element domains is not a problem for PARTENUM, so setting $n = 1$ gives the best performance, especially for relatively small strings.

Implementation

Figure 16 shows our implementation for string similarity joins. We start off with an input relation `String(id,str)` containing input strings and their identifiers. In the first step, we scan this relation, and for each input string, we generate their n-grams on-the-fly, generate signatures for the n-gram bags using an appropriate signature scheme, and finally write the signatures into a new relation `Signature (id,sign)`. All these steps are performed in application-level code; note in particular that we do not explicitly materialize the n-gram bags. Next, we generate the candidate pairs in a relation `CandPair(id1,id2)` exactly as we did for jaccard SSJoins. Finally, we retrieve the strings corresponding to the identifiers in each candidate pair by joining with the input relation `String(id,str)`, and for each such pair of strings we check if their edit distance is smaller than the join threshold. We perform the edit distance checking in application code. Note that we do not perform the SSJoin postfiltering step (Step 4 of Figure 2), i.e., check if the hamming distance of n-gram sets of two candidate pairs is less than nk, since, as mentioned earlier, this step does not remove all false positives from the string similarity join point of view. This step would have reduced the number string pairs for which we have to compute edit distance, but our experiments indicated that it did not improve overall performance.

Experiments

We use the same address data we used for jaccard SSJoins, but now we do not tokenize the strings into sets. The average length of a string is 58. We compared the performance of PARTENUM and prefix filter for small edit distance (1–3) thresholds. LSH does not map naturally for edit distances, so we do not include it in our experiments. For PARTENUM, we use $n = 1$, and for prefix filter we manually picked the optimal value of n (which was 4–6 depending on the edit threshold). Figure 18 shows the total computation time (y-axis) for string similarity joins with the two approaches for varying input sizes and varying edit thresholds. Again, the overall nature of the results is qualitatively similar to the results of jaccard SSJoins. Note that the y-axis for Figure 18 is "cut" at two points. The F_2 measures also closely mirrored the total computation time;

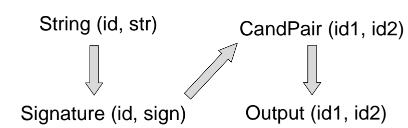


Figure 16: String similarity join implementation

```
CandPair (id1, id2):
   Select Distinct S1.id as id1, S2.id as id2
   From Signature as S1, Signature as S2
   Where S1.Sign = S2.Sign

Output (id1, id2):
   Select C.id1, C.id2
   From CandPair as C, Strings as S1, Strings as S2
   Where C.id1 = S1.id and C.id2 = S2.id and
         EDIT(S1.Str, S2.Str) < k
```

Figure 17: String similarity join: Queries

we do not show them due to space constraints.

8.3 Weighted Jaccard Similarity

Our final set of experiments is on weighted jaccard SSJoins with IDF-based weights. We use the same address data as previous experiments, tokenized as in the case of jaccard SSJoins. The implementation details are almost identical to those for unweighted jaccard, with minor variations for handle weights. We measured the performance of WTENUM, prefix filter, and LSH (0.95). Figure 19 shows the total computation time (y-axis) for these algorithms for varying input sizes and SSJoin thresholds. We highlight only the important qualitative differences from unweighted experiments. The performance of WTENUM is actually significantly better than LSH for this data set. This is primarily because WTENUM exploits the frequency information in the IDFs, while LSH does not. Also, the performance of WTENUM does not fall steeply when SSJoin thresholds are lowered, as in the case of PARTENUM. The overall scalability characteristics and relative performance of the algorithms is similar to the unweighted case (with PARTENUM replaced by WTENUM).

9. RELATED WORK

Previous work on set-similarity joins broadly fall into two categories. In the first category, set-similarity joins occur as an implicit operation as part of some application [4, 8, 15, 19]. The focus of this category of work is not in solving general purpose set-similarity joins, and they often involve implementation tricks and details that are highly specific to the application. For example, reference [19] only considers fixed size sets, since these sets correspond to n-grams of fixed length genome subsequences. For most of these applications, SSJoin occurs as a standalone operator, so approximate answers often suffice. Not surprisingly, all the above work uses the idea of locality sensitive hashing [13] in some form or the other.

The second category of work, which is more closely related to this paper, considers the problem of supporting SSJoins within a regular DBMS [22, 6]. Exact answers to SSJoins are important in this setting. Reference [22] proposes a variety of algorithms. The signature schemes used by these algorithms are all fairly simple, and the focus is on detailed implementation issues. An important feature of these algorithms is that they represent monolithic implementations of the SSJoin operator from scratch. Many of the algorithms also assume the availability of a large amount of main memory, comparable to the input data size. The other work [6] in this category is more closely related to this paper. This paper proposes prefix filtering (some ideas of prefix filtering are also present in the algorithms of [22]), and also studies the alternate implementation strategy that uses the processing capabilities of a DBMS for most of SSJoin computation.

SSJoins are closely related to set-containment joins, which has been the subject of several previous work [17, 18, 20]. Partial set-containment joins, a generalization, is covered by [6]. Supporting general string similarity joins within a DBMS has been studied in [14]. Interestingly, their implementation also uses existing operators within a DBMS for most of the join computation, just like the implementation that we studied in this paper.

General similarity joins are closely related to proximity search, where the goal is to retrieve, given a *lookup* object (set or vector), the closest object from a given collection; the challenge is to index the collection so that the lookup can be efficient. In fact, LSH was proposed originally for proximity search [13]. We have not yet explored if our signature schemes would be applicable to proximity search.

10. CONCLUSIONS

In this paper, we presented new algorithms for computing exact set-similarity joins. Some of our algorithms have precise theoretical guarantees and they are the first algorithms for set-similarity joins with this property. Our experiments indicate that our algorithms outperform previous exact algorithms by more than an order of magnitude in many cases. Also, they have excellent scaling properties with respect input size, which previous exact algorithms lack. The performance of our algorithm is comparable to that of LSH-based approximate algorithms for many scenarios, especially the data cleaning ones we are most interested in, and in many cases they even outperform LSH-based algorithms. Finally, our algorithms can be implemented on top of a regular DBMS with very little coding effort.

11. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the 28th Annual ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[2] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 586–597, Aug. 2002.

[3] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

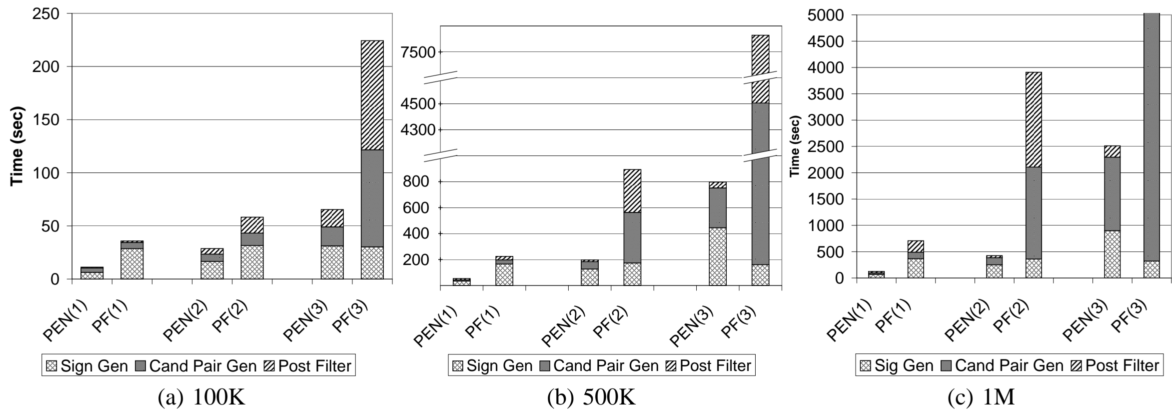
[4] K. Bharat and A. Z. Broder. Mirror, mirror on the web. *Computer Networks*, 31(11-16):1579–1590, May 1999.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 313–324, June 2003.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. of the 22nd Intl. Conf. on Data Engineering*, Apr. 2006. (To Appear).

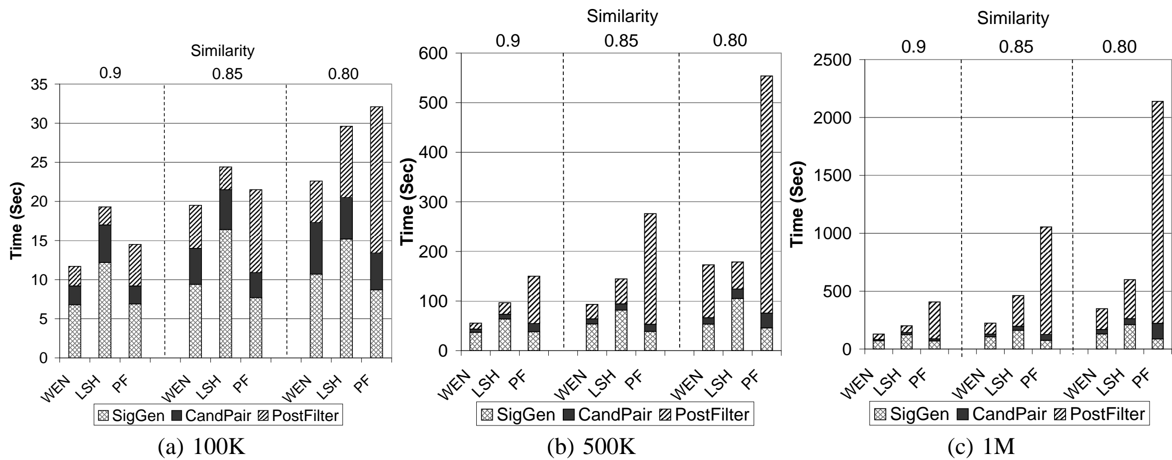
[7] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. of the 1999ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 1999.

[8] E. Cohen, M. Datar, S. Fujiwara, et al. Finding interesting associations without support pruning. In *Proc. of the 16th Intl. Conf. on Data Engineering*, pages 489–499, Mar. 2000.



(a) 100K (b) 500K (c) 1M

Figure 18: String similarity join computation time



(a) 100K (b) 500K (c) 1M

Figure 19: Total weighted SSJoin computation time for address data

[9] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 201–212, June 1998.

[10] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–96, June 2005.

[11] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64(328):1183–1210, 1969.

[12] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, pages 155–166, June 2005.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of the 25th Intl. Conf. on Very Large Data Bases*, pages 518–529, Sept. 1999.

[14] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, et al. Approximate string joins in a database (almost) for free. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 491–500, Sept. 2001.

[15] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the 3rd Intl. Workshop on Web and Databases*, pages 129–134, May 2000.

[16] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 127–138, May 1995.

[17] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 157–168, June 2003.

[18] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. on Database Systems*, 28(1), Mar. 2003.

[19] M. Narayanan and R. M. Karp. Gapped local similarity search with provable guarantees. In *Proc. of the 4th Intl. Workshop on Algorithms in Bioinformatics*, pages 74–86, Sept. 2004.

[20] K. Ramaswamy, J. M. Patel, J. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 251–262, Sept. 2000.

[21] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of the 8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 269–278, July 2002.

[22] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 743–754, June 2004.