Robust and Efficient Fuzzy Match for Online Data Cleaning

Surajit Chaudhuri Kris Ganjam Venkatesh Ganti
Microsoft Research
{surajitc, krisgan, vganti}@microsoft.com

Rajeev Motwani
Stanford University
rajeev@cs.stanford.edu

ABSTRACT

To ensure high data quality, data warehouses must validate and cleanse incoming data tuples from external sources. In many situations, clean tuples must match acceptable tuples in *reference tables*. For example, product name and description fields in a sales record from a distributor must match the pre-recorded name and description fields in a product reference relation.

A significant challenge in such a scenario is to implement an efficient and accurate *fuzzy match* operation that can effectively clean an incoming tuple if it fails to match exactly with any tuple in the reference relation. In this paper, we propose a new similarity function which overcomes limitations of commonly used similarity functions, and develop an efficient fuzzy match algorithm. We demonstrate the effectiveness of our techniques by evaluating them on real datasets.

1. INTRODUCTION

Decision support analysis on data warehouses influences important business decisions; therefore, accuracy of such analysis is crucial. However, data received at the data warehouse from external sources usually contains errors, e.g., spelling mistakes, inconsistent conventions across data sources, missing fields. Consequently, a significant amount of time and money are spent on *data cleaning*, the task of detecting and correcting errors in data. A prudent alternative to the expensive periodic data cleaning of an entire data warehouse is to avoid the introduction of errors during the process of adding new data into the warehouse. This approach requires input tuples to be validated and corrected before they are loaded. There is much information that can be used to achieve this goal.

A common technique validates incoming tuples against *reference relations* consisting of known-to-be-clean tuples. The reference relations may be internal to the data warehouse (e.g., customer or product relations) or obtained from external sources (e.g., valid address relations from postal departments). An enterprise maintaining a relation consisting of all its products may ascertain whether or not a sales record from a distributor describes a valid product by matching the product attributes (e.g., Part Number and Description) of the sales record with the Product relation; here, the Product relation is the reference relation. If the product attributes in the sales record match exactly with a tuple in the Product relation, then the described product is likely to be *valid*. However, due to errors in sales records, often the input product tuple does not match exactly with any in the Product relation. Then, errors in the input product tuple need to be corrected before it is loaded. The information in the input tuple is still very useful for identifying the correct reference product tuple, provided the matching is resilient to errors in the input tuple. We refer to this error-resilient matching of input tuples against the reference table as the *fuzzy match operation.*

Suppose the enterprise wishes to ascertain whether or not the sales record describes an existing customer by *fuzzily matching* the customer attributes of the sales record against the Customer relation. The reference relation, Customer, contains tuples describing all current customers. If the fuzzy match returns a target customer tuple that is either exactly equal or "reasonably close" to the input customer tuple, then we would have validated or corrected, respectively, the input tuple. The notion of closeness between tuples is usually measured by a *similarity function*. As shown in Figure 1, if the similarity between an input customer tuple and its closest reference tuple is higher than some threshold, then the correct reference tuple is loaded. Otherwise, the input is routed for further cleaning before considering it as referring to a new customer. A fuzzy match operation that is resilient to input errors can effectively prevent the proliferation of fuzzy duplicates [13] in a relation, i.e., multiple tuples describing the same real world entity.

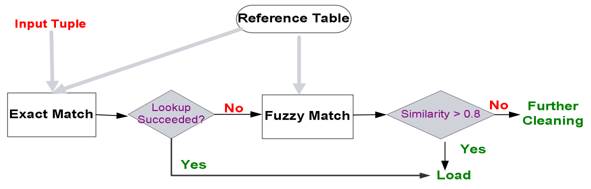


Figure 1: A Template for using Fuzzy Match

Our goal in this paper is to develop a robust and efficient fuzzy match algorithm, applicable across a wide variety of domains. We want a solution that provides a strong foundation for adding domain-specific enhancements. Most data warehouses are built atop database systems. Consequently, we require besides robustness and efficiency that the fuzzy match solution is implemented over standard database systems without assuming the persistence of complex data structures.

The critical ingredient of a fuzzy match operation is the similarity function used for comparing tuples. In typical application domains, the similarity function must definitely handle string-valued attributes and possibly even numeric attributes. In this paper, we focus only on string-valued attributes, where defining similarity and performing fuzzy matching is more challenging. Given the similarity function and an input tuple, the goal of the fuzzy match operation is to return the *reference tuple*—a tuple in the reference relation—which is closest to the input tuple. An extension is to return the closest K reference tuples enabling users, if necessary, to choose one among them as the target, rather

SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06...$5.00.

than *the* closest. A further extension is to only output K or fewer tuples whose similarity to the input tuple exceeds a user-specified *minimum similarity threshold*. This formulation is essentially that of the nearest neighbor problem, but there the domain is typically a Euclidean (or other normed) space with well-behaved similarity functions [11]. In our case, the data are not represented in "geometric" spaces, and it is hard to map them into one because the similarity function is relatively complex.

Previous approaches addressing the fuzzy match operation either adopt proprietary domain-specific functions (e.g., Trillium's reference matching operation for the address domain [23]) or use the *string edit distance* function for measuring similarity between tuples [17]. A limitation of the edit distance is illustrated by the following example. The edit distance function would consider the input tuple I3 in Table 2 to be closest to R2 in Table 1, even though we know that the intended target is R1. Edit distance fails because it considers transforming 'corporation' to 'company' more expensive than transforming 'boeing' to 'bon.' However, we know that 'boeing' and '98004' are more informative tokens than 'corporation' and so replacing 'corporation' with 'company' should be considered cheaper than replacing 'boeing' with 'bon' and '98004' with '98014.' In yet another example, note that the edit distance considers I4 closer to R3 than to its target R1. This is because it fails to capture the notion of a token or take into account the common error of token transposition.

Table 1: Organization Reference Relation

ID	Org. Name	City	State	Zipcode
R1	*Boeing Company*	*Seattle*	*WA*	*98004*
R2	*Bon Corporation*	*Seattle*	*WA*	*98014*
R3	*Companions*	*Seattle*	*WA*	*98024*

Table 2: Input Organization Tuples

Id	Org. Name	City	State	Zipcode
I1	Beoing Company	Seattle	WA	98004
I2	Beoing Co.	Seattle	WA	98004
I3	Boeing Corporation	Seattle	WA	98004
I4	Company Beoing	Seattle	NULL	98014

We start by proposing a new *fuzzy match similarity (fms)* function, which views a string as a sequence of tokens and recognizes the varying "importance" of tokens by explicitly associating weights quantifying their importance. Tuples matching on high weight tokens are more similar than tuples matching on low weight tokens. We adopt the successful *inverse document frequency* (IDF) weights from the IR literature for quantifying the notion of token importance; informally, the importance of a token decreases with its *frequency*, which is the number of times a token occurs in the reference relation [3]. Even though the approach of weight association is common in the IR literature, the effective use of token weights in combination with data entry errors (e.g., spelling mistakes, missing values, inconsistent abbreviations) has not been considered earlier.

Our notion of similarity between two tuples depends on the minimum cost of "transforming" one tuple into the other through a sequence of transformation operations (replacement, insertion, and deletion of tokens) where the cost of each transformation operation is a function of the weights of tokens involved. For example, it may be cheaper to replace the token 'corp' with 'corporation' than to replace 'corporal' with 'corporation' even though edit distances suggest otherwise. This notion of similarity based on transformation cost is similar to edit distance except that we operate on tokens and explicitly consider their weights.

The goal of the fuzzy match algorithm is to efficiently retrieve the K reference tuples closest to an input tuple. It is well-known that efficiently identifying the exact K nearest neighbors even according to the Euclidean and Hamming norms in high-dimensional spaces is hard [14]. Since the Hamming norm is a special case of the edit distance obtained by allowing only replacements, the identification of the exact closest K matches according to our fuzzy match similarity—which generalizes edit distance by incorporating token weights—is essentially hard. Therefore, we adopt a *probabilistic* approach where the goal is to return the closest K reference tuples with high probability. We pre-process the reference relation to build an index relation, called the *error tolerant index (ETI) relation,* for retrieving at run time a small set of candidate reference tuples, which we then compare with the input tuple. Our retrieval algorithm is *probabilistically safe* because we retrieve (with high probability) a superset of the K reference tuples closest to the input tuple. It is *efficient* because the superset is significantly (often by several orders of magnitude) smaller than the reference relation. The index relation ETI is implemented and maintained as a standard relation, and hence our solution can be deployed even over current operational data warehouses.

Our main contributions are the following. We propose a new fuzzy match similarity function that explicitly considers IDF token weights and input errors while comparing tuples. We propose the error tolerant index and a probabilistic algorithm for efficiently retrieving the K reference tuples closest to the input tuple, according to the fuzzy match similarity function. Finally, we present a thorough empirical evaluation on real datasets. Our techniques are extensible to use specialized (possibly domain-specific) token weight functions instead of the IDF weights.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we define the new similarity function. In Section 4, we describe (i) our algorithm to build the ETI, and (ii) our retrieval algorithm for efficiently identifying the target reference tuples. In Section 5, we discuss a few extensions to the algorithm. In Section 6, we discuss a thorough empirical study on real datasets, and conclude in Section 7.

2. RELATED WORK

Several methods for approximate string matching over dictionaries or collections of text documents have been proposed (e.g., [12], [17]). All of the above methods use edit distance as the similarity function, not considering the crucial aspect of differences in importance of tokens while measuring similarity.

Approximate string matching methods [e.g., 2, 18] preprocess the set of dictionary/text strings to build *q-gram tables* containing tuples for every string s of length q that occurs as a substring of some reference text string; the record also consists of the list of identifiers (or locations) of strings of which s is a substring. The error tolerant index relation ETI we build from the reference relation is similar in that we also store q-grams along with the list of record identifiers in which they appear, but the ETI (i) is smaller than a full q-gram table because we only select (probabilistically) a subset of all q-grams per tuple, and (ii) encodes column-boundaries specific to relational domains.

The information retrieval community has successfully exploited inverse document frequency (IDF) weights for differentiating the importance of tokens or words. However, the IR application assumes that all input tokens in the query are correct, and does not deal with errors therein. Only recently, some search engines (e.g., Google's "Did you mean?" feature) are beginning to consider even simple spelling errors. In the fuzzy match operation, we deal with tuples containing very few tokens (many times, around 10 or less) and hence cannot afford to ignore erroneous input tokens, as they could be crucial for differentiating amongst many thousands of reference tuples. For example, the erroneous token 'beoing' in the input tuple [beoing corporation, seattle, wa, NULL] is perhaps the most useful token for identifying the target from among all corporations in Seattle. Clustering and reference matching algorithms [e.g., 7, 8, 9] using the cosine similarity metric with IDF weighting also share the limitation of ignoring erroneous input tokens. Further, Cohen et al. improve efficiency by choosing probabilistically a subset of tokens from each document under the correct input token assumption [9]. In this paper, we propose a similarity function that does not assume correctness of input tokens, and further improve efficiency by exploiting the variance in weights of input tokens.

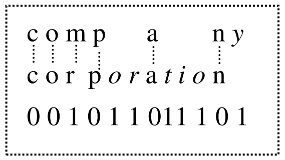
As discussed earlier, almost all solutions for the nearest neighbor problem are targeted at data in Euclidean/normed spaces [11] and hence inapplicable to our setting. There has been some recent work on general metric spaces [e.g., 5, 19], but their complexity and performance are not suitable for the high-throughput systems of interest here. Moreover, many of these solutions cannot be deployed easily over current data warehouses because they require specialized index structures (e.g., M-trees, tries) to be persisted.

Some recent techniques addressed the related problem of eliminating "fuzzy duplicates" in a relation by using a similarity function and identifying highly similar tuples as duplicates. Some are based on the use of edit distance [e.g., 13], some on cosine similarity with IDF weights [e.g., 8], some on learning similarity functions from training datasets [e.g., 10, 20], and some on the use of dimension hierarchies [1]. However, all such techniques are designed for use in an offline setting and do not satisfy the efficiency requirements of the online fuzzy match operation where input tuples have to be quickly matched with target reference tuples before being loaded into the data warehouse. A complementary use of solutions to both problems is to first clean a relation by eliminating fuzzy duplicates and then piping further additions through the fuzzy match operation to prevent introduction of new fuzzy duplicates.

Several commercial products (e.g., Trillium, Vality, Axciom) leverage characteristics peculiar to the address domain in their *proprietary* algorithms for matching addresses and individual or organization records. The record linkage literature—a survey can be found in [24]—also considers the problem of identifying matching records across relations (consisting mainly of census records of individuals), and employs a variety of (domain-specific) similarity functions. In contrast, our goal in this paper is to develop a domain-independent method.

3. THE SIMILARITY FUNCTION

In this section, we define the fuzzy match similarity (fms) function for comparing tuples. We start with a few definitions.



Edit Distance: The *edit distance* $ed(s_1, s_2)$ between two strings s_1 and s_2 is the minimum number of character edit operations (delete, insert, and substitute) required to transform s_1 into s_2, normalized by the maximum of the lengths of s_1 and s_2. For the example shown in the adjacent figure the edit distance between the strings 'company' and 'corporation' is $7/11 \approx 0.64$, and the sequence of edit operations is shown. Vertical lines indicate either exact matches (cost is 0) or substitutions (cost is 1). Characters in italics are deleted or inserted and always have a unit cost.

Reference Relation: Let R[*tid*, $A_1, \ldots, A_n$] be a reference relation where A_i denotes the ith column. We assume that each A_i is a string-valued attribute (e.g., of type varchar). We also assume that *tid* (for *tuple identifier*) is a key of R. We refer to a tuple whose tid attribute assumes value r as the *tuple r*. We use v[i] to denote the value a_i in the tuple v[r, $a_1, \ldots, a_n$].

Tokenization: Let *tok* be a tokenization function which splits a string s into a set of tokens, *tok(s),* based on a set of delimiters (say, the white space characters). For example, tok(v[1]) of the tuple v = [R1, Boeing Company, Seattle, WA, 98004] is {boeing, company}. Observe that we ignore case while generating tokens. For tokens generated from attribute values of tuples, we associate the *column property*—the column from which a token originates. For example, the column property of tokens in tok(v[*col*]) is *col.* Consequently, the token 'madison' in the name column of a customer relation is considered different from the token 'madison' in the city column. The *token set tok(v)* is the multiset union of sets tok(a_1),…,tok(a_n) of tokens from the tuple v[r, $a_1, \ldots, a_n$]. That is, if a token t appears in multiple columns, we retain one copy per column in tok(v), distinguishing each copy by its column property. We say that a token *t is in tok(v)* if t is a member of some tok(a_i), for $1 \leq i \leq n$.

Weight Function: We now adapt the IDF weight function to the relational domain by treating each tuple as a document of tokens. The motivation for this definition is clear from the following example – we expect the weight of token 'corporation' in the organization-name column to be less than that of 'united' since corporation is a frequent token in that column. Let the frequency of token t in column i, denoted *freq(t, i),* be the number of tuples v in R such that tok(v[i]) contains t. The IDF value, *IDF(t, i),* of a token t with respect to the ith column in the schema of R is computed as follows, when freq(t, i) > 0,

$$w(t,i) = IDF\ (t,i) = \log \frac{|R|}{freq\ (t,i)}$$

For a token t whose frequency in column i is 0, our philosophy is that t is an erroneous version of some token in the reference tuple. Since we do not know the token to which it corresponds, we define the weight w(t, i) to be the average weight of all tokens in the ith column of relation R. For clarity in presentation, when the column property of a token is evident from the context, we use *w(t)* to denote *w(t, i).*

3.1 Fuzzy Similarity Function (*fms*)

Informally, the similarity between an input tuple and a reference tuple is the cost of *transforming* the former into the latter—the less the cost, the higher the similarity. We consider the following transformation operations: *token replacement, token insertion,* and *token deletion*. Each operation is associated with a *cost* that depends on the weight of the token being transformed. We now describe the cost of each transformation operation. Let u and v be two tuples with the schema $R[A_1,\ldots,A_n]$. We will be considering only the case where u is an input tuple and v is a reference tuple, and we are interested in transforming u into v.

(i) *Token replacement*: The cost of replacing a token t_1 in tok(u[i]) by token t_2 from tok(v[i]) is $ed(t_1,t_2)\cdot w(t_1,i)$. If t_1 and t_2 are from different columns, we define the cost to be infinite.

(ii) *Token insertion*: The cost of inserting a token t into u[i] is $c_{ins}\cdot w(t, i)$, where the *token insertion factor* c_{ins} is a constant between 0 and 1.

(iii) *Token deletion*: The cost of deleting a token t from u[i] is w(t,i).

Observe that the costs associated with inserting and deleting the same token may be different. We believe that this asymmetry is useful, since in many scenarios it is more likely for tokens to be left out during data entry than it is for spurious tokens to be inserted. Therefore, absence of tokens is not penalized heavily.

We ignore the tid attribute while comparing tuples. Transforming u into v requires each column u[i] to be transformed into v[i] through a *sequence* of transformation operations, whose cost we define to be the sum of costs of all operations in the sequence. The transformation cost *tc(u[i], v[i])* is the cost of the *minimum cost transformation sequence* for transforming u[i] into v[i]. The cost *tc(u, v)* of transforming u into v is the sum over all columns i of the costs tc(u[i], v[i]) of transforming u[i] into v[i].

$$tc(u,v) = \sum_i tc(u[i], v[i])$$

The minimum transformation cost tc(u[i], v[i]) can be computed using the dynamic programming algorithm used for edit distance computation [22].

Consider the input tuple u[Beoing Corporation, Seattle, WA, 98004] in Table 2 and the reference tuple v[Boeing Company, Seattle, WA, 98004]. The minimum cost transformation of u[1] into v[1] requires two operations – replacing 'beoing' by 'boeing' and replacing 'corporation' by 'company'. The function tc(u[1], v[1]) is the sum of costs of these two operations; assuming unit weights on all tokens, this is 0.97 by adding 0.33 for replacing 'beoing' with 'boeing' which are at an edit distance 0.33, and 0.64 for replacing 'corporation' with 'company' which are at an edit distance 0.64. In this example, only tc(u[1], v[1]) is nonzero among column-wise transformation costs.

Definition of fms: We now define the *fuzzy match similarity* function *fms(u, v)* between an input tuple u and a reference tuple v in terms of the transformation cost *tc(u, v)*. Let *w(u)* be the sum of weights of all tokens in the token set tok(u) of the input tuple u. Similarity between u and v is defined as:

$$fms(u,v) = 1 - \min\left(\frac{tc(u,v)}{w(u)}, 1.0\right)$$

In the above example involving I3 and R1, w(I3) = 5.0 because there are five tokens in tok(I1) and the weight of each token is 1.0. Therefore, fms(I3, R1) = 1 - 0.97/5.0 = 0.806. We define fms asymmetrically because we believe the cost of transforming a dirty input tuple into a clean reference tuple is different from the reverse transformation. Also, in this paper, we only transform input tuples into clean reference tuples, and never the other way.

3.2 Edit Distance and fms

For a broad subclass of errors, we compare the weight assignment strategy implicitly adopted by the edit distance ed with that of the fuzzy match similarity fms, to isolate scenarios when they agree or disagree on fuzzy match. The comparison also justifies, although only informally, our belief that fms is the more appropriate choice in practice.

We consider the subclass of *order-preserving errors*. Under this class of errors, an input tuple and its target reference tuple are consistent in the ordering among tokens after each input token is mapped to the closest matching reference token, and each input token is transformed to its counterpart in the reference tuple. Let $u_1,\ldots,u_m$ be the list of tokens in the input tuple u ordered according to their position in u. Let $v_1,\ldots,v_m$ be the similarly ordered list of tokens in the reference tuple v. In the class of order-preserving errors, for all i, the input token u_i is transformed to the reference token v_i. Let *ed(u, v)* denote the total (minimum) number of edit operations for transforming each u_i into v_i, normalized by *max(L(u), L(v))* where the *length L(z)* of a tuple z is the sum of lengths of tokens $z_1,\ldots,z_p$ in tok(z), i.e., $L(z)=\sum|z_i|$. We now rewrite ed(u, v) to highlight the implicit weight assignment to the $u_i \rightarrow v_i$ token-mapping.

$$ed(u,v) = \frac{L(u)}{\max(L(u), L(v))}\sum_i \frac{\max(|u_i|, |v_i|)}{L(u)} ed(u_i, v_i) \quad \textbf{(1)}$$

Observe that the $u_i \rightarrow v_i$ mapping gets a weight proportional to $\max(|u_i|, |v_i|)/L(u)$. Therefore, ed implicitly assigns weights to token mappings in proportion to their lengths, i.e., longer tokens get higher weights. For example, 'corporation' to 'company' gets a higher weight than 'boeing' to 'bon' thus explaining why ed matches input tuple I3 (in Table 2) with R2 (in Table 1) instead of the correct target R1. Extensive empirical evidence from the IR application suggests the superiority of IDF weights to token lengths for capturing the notion of token importance [3]. Hence, we expect fms to be more beneficial than ed in practice.

4. FUZZY MATCH

We first formally define the fuzzy match problem before describing the algorithm.

The K-Fuzzy Match Problem: Given a reference relation R, a *minimum similarity threshold c* ($0 < c < 1$), the similarity function f, and an input tuple u, find the set *FM(u) of fuzzy matches* of at most K tuples from R such that

(i) $fms(u, v) \geq c$, for all v in FM(u)

(ii) $fms(u, v) \geq fms(u, v')$ for any v in FM(u) and v' in R−FM(u)

Observe that by setting the minimum similarity threshold c to be zero, we can simulate the scenario where a user is interested in all closest K reference tuples. When more than K−i+1 reference tuples are tied for the ith, …, Kth (i > 1) best fuzzy matches, we break ties by choosing an arbitrary subset of the tied reference tuples such that the total number of returned fuzzy matches is K.

Given an input tuple u, the goal of the fuzzy match algorithm is to identify the *fuzzy matches*—the K reference tuples closest to u. A naïve algorithm scans the reference relation R comparing each tuple with u. A more efficient approach is to build an "index" on the reference relation for quickly retrieving a superset of the target fuzzy matches. Standard index structures like B+-tree indexes cannot be deployed in this context because they can only be used for exact or prefix matches on attribute values. Therefore, we gather, during a pre-processing phase, additional indexing information for efficiently implementing the fuzzy match operation. We store the additional information as a standard database relation, and index this relation using standard B+-trees to perform fast exact lookups. We refer to this *indexed relation* as the *error tolerant index (ETI).* The challenge is to identify and to effectively use the information in the indexed relation. Our solution is based on the insight of deriving from fms an easily indexable similarity function *fms^{apx}* with the following characteristics. (i) fms^{apx} upper bounds fms with *high probability*. (ii) We can build the error tolerant index (ETI) relation for efficiently retrieving a small *candidate set* of reference tuples whose similarity with the input tuple u, as per fms^{apx}, is greater (probabilistically) than the minimum similarity threshold c. Therefore, with a high probability the similarity as per fms between any tuple in the candidate set and u is greater than c. From this candidate set, we return the K reference tuples closest to u as the *fuzzy matches*.

In Section 4.1, we define fms^{apx}. In Section 4.2, we describe the ETI relation as well as an algorithm for building it. In Section 4.3, we present an efficient algorithm to process fuzzy matching queries, and we discuss their resource requirements in Section 4.4.

4.1 Approximation of fms

Our goal in this section is to derive fms^{apx} an approximation of fms for which we can build an indexed relation. fms^{apx} is a pared down version of fms obtained by (i) ignoring differences in ordering among tokens in the input and reference tuples, and (ii) by allowing each input token to match with the "closest" token from the reference tuple. Since disregarding these two distinguishing characteristics while comparing tuples can only increase similarity between tuples, fms^{apx} is an upper bound of fms. For example, the tuples [boeing company, seattle, wa, 98004] and [company boeing, seattle, wa, 98004] which differ only in the ordering among tokens in the first field are considered identical by fms^{apx}. In fms^{apx}, we measure the closeness between two tokens through the similarity between *sets of substrings—called q-gram sets*—of tokens (instead of edit distance between tokens used in fms). Further, this q-gram set similarity is estimated well by the commonality between small probabilistically chosen subsets of the two q-gram sets. This property can be exploited, like we do later, to build an indexed relation for fms^{apx} because for each input tuple we only have to identify reference tuples whose tokens share a number of chosen q-grams with the input tuple. We first define the approximation of the q-gram set similarity between tokens. In Lemma 4.2, we relate this similarity with the edit distance between tokens using an "adjustment term" which only depends on the value of q introduced below.

Q-gram Set: Given a string s and a positive integer q, the set *$QG_q(s)$ of q-grams* of s is the set of all size q substrings of s. For example, the 3-gram set QG_3("boeing") is {boe, oei, ein, ing}. Because we fix q to be a constant, we use *QG(s)* to denote $QG_q(s)$.

Jaccard Coefficient: The *Jaccard coefficient $sim(S_1, S_2)$* between two sets S_1 and S_2 is $\frac{|S1 \cap S2|}{|S1 \cup S2|}$.

Min-hash Similarity: Let *U* denote the universe of strings over an alphabet $\sum$, and $h_i: U \rightarrow N$, i = 1,…,H be H hash functions mapping elements of U uniformly and randomly to the set of natural numbers N. Let S be a set of strings. The *min-hash signature mh(S)* of S is the vector *$[mh_1(S), \ldots, mh_H(S)]$* where the *$i^{th}$ coordinate $mh_i(S)$* is defined as $mh_i(S) = \arg\min_{a \in S} h_i(a)$. Let I[X] denote an *indicator variable* over a boolean X, i.e., I[X] = 1 if X is true, and 0 otherwise. Then (as shown in [4, 6]),

$$E[sim(S_1, S_2)] = \frac{1}{H}\sum_{i=1}^{H} I[mh_i(S_1) = mh_i(S_2)]$$

Computing the min-hash signature is like throwing darts at a board and stopping when we hit an element of S. Hence, the probability that we hit an element in $S_1 \cap S_2$ before another element in $S_1 \cup S_2$ is equal to $sim(S_1,S_2)$. We now define token similarity in terms of the min-hash similarity between their q-gram sets. Let q and H be positive integers. The *min-hash similarity $sim_{mh}(t_1,t_2)$* between tokens t_1 and t_2 is:

$$sim_{mh}(t_1, t_2) = \frac{1}{H}\sum_{i=1}^{H} I[mh_i(QG(t_1)) = mh_i(QG(t_2))]$$

We define the similarity function fms^{apx} and then show (i) its *expectation* is greater than fms, and (ii) the probability of fms^{apx} being greater than fms can be made arbitrarily large by choosing an appropriate min-hash signature size.

Definition of fms^{apx}: Let u, v be two tuples, and let $d_q = (1-1/q)$ be an *adjustment term.*

$$fms^{apx}(u,v) = \frac{1}{w(u)}\sum_i \sum_{t \in tok(u[i])} w(t) \cdot \max_{r \in tok(v[i])} \left(\frac{2}{q} sim_{mh}(QG(t), QG(r)) + d_q\right)$$

Consider the tuple I4 in Table 2 and the tuple R1 in 1. Suppose q=3 and H=2. We use the notation *t:w* to denote a token with weight w. Suppose the tokens and their weights in I4 are company:0.25, beoing:0.5, seattle:1.0, 98004:2.0; their total weight is 3.75. Suppose their min-hash signatures are [eoi, ing], [com, pan], [sea, ttl], [980, 004], respectively. The tokens in R1 are boeing, company, seattle, wa, 98004. Suppose their min-hash signatures are [oei, ing], [com, pan], [sea, ttl], [wa], [980, 004], respectively. Then, 'company' matches with 'company', 'beoing' with 'boeing', 'seattle' with 'seattle', '98004' with '98004'. The score from matching 'beoing' with 'boeing' is: w(beoing)*(⅔*0.5 + (1-1/3))=w(beoing). Since every other token matches exactly with a reference token, fms^{apx}(I4, R1) = 3.75/3.75. In contrast, fms(I4, R1) will also consider the cost of reconciling differences in order among tokens between I4 and R1, and the cost of inserting token 'wa'. Hence, fms(I4, R1) is less than fms^{apx}(I4, R1).

Lemma 4.1: Let $0 < \delta < 1, \varepsilon > 0, H \geq 2\delta^{-2} \log \varepsilon^{-1}$. Then

(i) E[fms^{apx}(u, v)] ≥ fms(u, v)

(ii) $P(fms^{apx}(u,v) \leq (1-\delta) fms(u,v)) \leq \varepsilon$

Sketch of Proof: We require the following definitions.

$$f_1(u,v) = \frac{1}{w(u)} \sum_i \sum_{t \in tok(u[i])} w(t) \cdot \underset{r \in tok(v[i])}{Max} (1 - ed(t,r))$$

$$f_2(u,v) = \frac{1}{w(u)} \sum_i \sum_{t \in tok(u[i])} w(t) \cdot \underset{r \in tok(v[i])}{Max} \left(\frac{2}{q} sim(QG(t), QG(r)) + d_q\right)$$

Result (i) falls out of the following sequence of observations.

(i) Ignoring the ordering among tokens while measuring f, and allowing tokens to be replaced by their best matches always results in over estimating fms. Therefore, $f_1(u, v) \geq fms(u, v)$.

(ii) Edit distance between strings is approximated by the similarity between the sets of q-grams (Lemma 4.2 below), and $\max(|t|, |r|) \geq |QG(t) \cup QG(r)|/2$. Hence, $f_2(u, v) \geq f_1(u, v)$.

(iii) Min-hash similarity between tokens is an unbiased estimator of the Jaccard coefficient between q-gram sets of tokens. Therefore, $E[fms^{apx}(u, v)] = f_2(u, v) \geq fms(u, v)$.

Since $E[fms^{apx}(u, v)] = f_2(u, v) \geq fms(u, v)$ for all $H > 0$, splitting $fms^{apx}(u, v)$ into the average of H independent functions f_1', …, f_H' one for each min-hash coordinate such that f_i' has the same expectation as fms^{apx} and using Chernoff bounds [16], we have the following inequality, which yields Result (ii).

$$E[X < (1-\delta) f(u,v)] \leq E[X < (1-\delta) E[X]) \leq e^{-\frac{\delta^2 H f_2(u,v)}{2}}$$

Lemma 4.2 [15]: Let t_1, t_2 be two tokens, and $m = \max(|t_1|, |t_2|)$. Let $d = (1-1/q)\cdot(1-1/m)$. Then,

$$1 - ed(s_1, s_2) \leq \frac{|QG(s_1) \cap QG(s_2)|}{mq} + d$$

Because the probability $P(fms^{apx}(u, v) \geq (1-\delta)fms(u, v))$ can be increased arbitrarily, we loosely say that fms^{apx} upper bounds fms.

4.2 The Error Tolerant Index (ETI)

The primary purpose of ETI is to enable, for each input tuple u, the efficient retrieval of a *candidate set* S of reference tuples whose similarity with u is greater than the minimum similarity threshold c. Recall from the definition of fms^{apx} that $fms^{apx}(u, v)$ is measured by comparing min-hash signatures of tokens in tok(u) and tok(v). Therefore, for determining the candidate set, we need to *efficiently* identify for each token t in tok(u), a set of reference tuples sharing min-hash q-grams with that of t. Consider the example input tuple [Beoing Company, Seattle, WA, 98004] shown in Figure 2. The topmost row in the figure lists tokens in the input tuple, the next row lists q-gram signatures of each token, and the lowest row lists sets (S_1 through S_9) of tids of reference tuples with tokens whose min-hash signatures contain the corresponding q-gram. For example, the set $S_1 \cup S_2$ is the set of tids of reference tuples containing a token in the Org. Name column that shares a min-hash q-gram with 'beoing'. Extending this behavior to q-gram signatures of all tokens, the union of all S_i's contains the candidate set S. In order to identify such sets of tids, we store in ETI each q-gram s along with the list of all tids of reference tuples with tokens whose min-hash signatures contain s.

We now formally describe the ETI and its construction. Let R be the reference relation, and H the size of the min-hash signature. ETI is a relation with the following schema: [QGram, Coordinate, Column, Frequency, Tid-list] such that each tuple e in ETI has the following semantics. e[Tid-list] is a list of tids of all reference tuples containing at least one token t in the field e[Column] whose e[Coordinate]-th min-hash coordinate is e[QGram]. e[Frequency] is the number of tids in e[Tid-list]. Constructing a tuple [s, j, i, frequency, tid-list] in ETI requires that we know the list of all reference tuple tids containing ith column tokens with s as their jth min-hash coordinate. The obvious method of computing all ETI tuples in main-memory by scanning and processing each reference tuple is not scalable because the combined size of all tid-lists is usually larger than the amount of available main memory. To build the ETI efficiently, we leverage the underlying database system by first building a temporary relation called the *pre-ETI* with sufficient information and then construct the ETI relation from the pre-ETI using SQL queries.

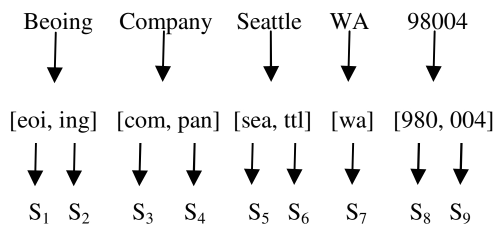


Figure 2: Candidate set generation

The schema of the pre-ETI is: [QGram, Coordinate, Column, Tid]. We scan the reference relation R processing each tuple v as follows. We tokenize v, and for each ith column token t in tok(v), we determine its min-hash signature mh(t) of size H. We insert into the pre-ETI a row [q, j, i, r] for the jth min-hash coordinate in mh(t). For example, if the size-2 signature of the token 'company' belonging to column 1 of the tuple R1 is [com, pan], then we insert the rows [com, 1, 1, R1], [pan, 1, 1, R1] into the pre-ETI. In practice, we can batch such insertions.

Table 3: An Example ETI Relation

Q-gram	Coordinate	Column	Frequency	Tid-list
oei	1	1	1	{R1}
ing	2	1	1	{R1}
com	1	1	2	{R1,R3}
pan	2	1	2	{R1,R3}
bon	1	1	1	{R2}
orp	1	1	1	{R2}
ati	2	1	1	{R2}
sea	1	2	3	{R1,R2,R3}
ttl	2	2	3	{R1,R2,R3}
wa	1	3	3	{R1,R2,R3}
980	1	4	3	{R1,R2,R3}
004	2	4	1	{R1}
014	2	4	1	{R2}
024	2	4	1	{R3}

All tuples required to compute any one ETI tuple occur together in the result of the *ETI-query*: "select QGram, Coordinate, Column, Tid from *pre-ETI* order by QGram, Coordinate, Column, Tid." We scan the result of the ETI-query, and for a group of tuples corresponding to the q-gram s which occurs as the jth min-hash coordinate of (multiple) tokens in the ith column, we insert the tuple [s, j, i, *freq(s, j, i)*, *tid-list*] in ETI, where freq(s, j, i) is the size of the group, and tid-list the list of all tids in the group. q-grams whose frequencies are above a large threshold, called the stop q-gram threshold (set to 10000 in our implementation), are considered stop tokens. For such q-grams, we insert a NULL value in the tid-list column. Finally, we build a clustered index on the [QGram, Coordinate, Column] attribute combination of the ETI relation so that queries looking up ETI on [QGram, Coordinate, Column] combinations are answered efficiently.

An example ETI relation for the reference relation in Table 1 with q=3 and H=2 is shown in Table 3. If the length of a token is less than q, then we assume that its min-hash signature consists of the token itself. The tuple [R1, Boeing Company, Seattle, WA, 98004] in Table 1 with min-hash signatures {[oei, ing], [com, pan], [sea, ttl], [wa], [980, 004]} for its tokens, respectively, has the tid R1 in the tid-lists of each of these q-grams.

4.3 Query Processing

In this section, we describe the algorithm for processing fuzzy match queries—queries asking for K fuzzy matches of an input tuple u whose similarities (as per fms) with u are above a minimum similarity threshold c. The goal is to reduce the number of lookups against the reference relation by effectively using the ETI. We first describe the *basic algorithm*, which fetches tid-lists by looking up ETI of all q-grams in min-hash signatures of all tokens in u. We then introduce an optimization called *optimistic short circuiting*, which exploits differences in token weights and the requirement to fetch only the K closest tuples to significantly reduce the number of ETI lookups. For efficient lookups, we assume that the reference relation R is indexed on the Tid attribute, and the ETI relation is indexed on the [QGram, Coordinate, Column] attribute combination.

4.3.1 Basic Algorithm

The basic algorithm for processing the fuzzy match query given an input tuple u is as follows. For each token t in tok(u), we compute its IDF weight w(t), which requires the frequency of t. We can store these frequencies in the ETI and fetch them by issuing a SQL query per token. However, we assume for now that frequencies of tokens can be quickly looked up from a main memory cache called the *token-frequency cache*. (See Section 4.4.1 for a discussion on this issue.) We then determine the min-hash signature mh(t) of each token t. (If $|t| \le q$, we define mh(t)=[t].) We assign the weight w(t)/|mh(t)| to each q-gram in mh(t). Using the ETI, we then determine a *candidate set* S of reference tuple tids whose similarity (as per fms^{apx} and hence fms) with the input tuple u is greater than c. We fetch from the reference relation all tuples in S to verify whether or not their similarities with u (as per fms) are truly above c. Among those tuples which passed the verification test, we return the K tuples with the K highest similarity scores.

Candidate Set Determination: We compute the candidate set S as the union of sets S_k, one for each q-gram q_k in the min-hash signatures of tokens in tok(u). For a q-gram q_k which is the ith coordinate in the min-hash signature mh(t) of a token t in the jth column, S_k is the tid-list from the record [q_k, i, j, freq(q_k, i, j), S_k] in ETI. Observe that the lookup for [q_k, i, j, freq(q_k, i, j), S_k] is efficient because of the index on the required attribute combination of ETI. Each tid in S_k is assigned a score that is proportional to the weight w(t) of the parent token t. If a tuple with tid r is very close to the input tuple u, then r is a member of several sets S_k and hence gets a high overall score. Otherwise, r has a low overall score. Tids that have an overall score greater than w(u)·c minus an *adjustment term*—a correction to approximate the edit distance between tokens with the similarity between their q-gram sets—constitute the candidate set.

During the process of looking up tid-lists corresponding to q-grams, we maintain the *scores of tids* in these tid-lists in a hash table. At any point, the score of a tid equals the sum of weights of all q-grams whose tid-lists it belongs to. The weight $w(q_k)$ assigned to a q-gram q_k in the min-hash signature mh(t_i) of a token t_i is w(t_i)/|mh(t_i)|. If a tid in S_k is already present in the hash table, then its score is incremented by w(q_k). Otherwise, we add the tid to the hash table with an initial score of w(q_k). After all q-grams in the signatures of input tokens are processed, we select a tid r and add it to the candidate set S only if its score is above w(u)·c (minus the adjustment term).

An optimization to the after-the-fact filtering of tids with low scores described above is to add a tid to the hash table only if the score it can potentially get after all min-hash q-grams are processed is greater than the threshold. We add a new tid to the hash table only if the total weight, which is an upper bound on the score a new rid can get, of all min-hash q-grams yet to be looked up in the ETI is greater than or equal to w(u)·c. This optimization significantly reduces the number of tids added to the hash table. We summarize the basic algorithm in Figure 3.

```
FuzzyMatch(input tuple u, H, ETI, R, c)
1. Initialize hash table TidScores; AdjustmentTerm = 0
2. Tokenize u and compute min-hash signatures Q of all tokens
3. Assign token weights; RemWt = sum of all token weights
4. threshold = c·RemWt
5. For each q-gram s in Q s.t. s = $mh_i(t)$ of token t in column col
6.    if ($mh_i(t)$ is the first q-gram of mh(t) to be looked up)
7.       AdjustmentTerm += w(t)·(1-1/q)
8.    Fetch tid-list(s) by looking up (s, i, col) against ETI
9.    Update TidScores
         a. Increment scores of existing tids by w(t)/|mh(t)|
         b. If RemWt ≥ threshold, insert new tids with score w(t)/|mh(t)|.
10.   RemWt −= w(s)
11. Fetch tuples from R for TIDs with score ≥ c–AdjustmentTerm
12. Compare, using f, each of these tuples with u
13. Return K (or less) most similar tuples with similarity above w(u)·c
```

Figure 3: Basic Query Processing Algorithm

We illustrate the above procedure with the example input tuple I1 in Table 2 and the ETI in Table 3. Suppose q=3 and H=2. We use the notation [q1, q2]:w to denote the min-hash signature [q1, q2] with each q-gram assigned a weight of w. The tokens and their weights in I1 are beoing:0.5, company:0.25, seattle:1.0, wa:0.75, 98004:2.0; their total weight is 4.5. Suppose their min-hash signatures are [eoi, ing]:0.25, [com, pan]:0.125, [sea, ttl]:0.5, [wa]:0.75, [980, 004]:1.0. We lookup ETI to fetch the following tid-lists: [{}, {R1}], [{R1, R3}, {R1, R3}], [{R1, R2, R3}, {R1, R2, R3}], [{R1, R2, R3}], [{R1, R2, R3}, {R1}]. For the purpose of this example, we ignore the adjustment term. R1 gets an overall score of 4.25, R2 a score of 2.75, and R3 3.0. Depending on the threshold, the candidate set is a subset of {R1, R2, R3}. For the example in Figure 2, suppose we looked up min-hash q-grams 'eoi', 'ing', 'com', 'pan', 'sea', 'ttl'. While processing the q-gram 'wa', we add new tids to the hash table only if 0.75 + 2.0 (the total weight of the remaining q-grams) is greater than w(u)·c. We now formally state that the basic algorithm retrieves the correct fuzzy matches with a high probability. For the purpose of the formal guarantee in Theorem 1, we assume that no q-gram is classified as a stop token. Alternatively, the stop q-gram threshold is set to at least |R|. We omit the proof due to space constraints.

Theorem 1: Let $0 < \delta < 1, \varepsilon > 0, H \geq 2\delta^{-2} \log \varepsilon^{-1}$. The basic query processing algorithm returns the K reference tuples closest, as per fms, to the input tuple with a probability of at least $1-\varepsilon$.

4.3.2 *Optimistic Short Circuiting (OSC)*

In the basic algorithm, we fetch tid-lists by looking up ETI of all q-grams in min-hash signatures of all tokens. We now discuss the short circuiting optimization to significantly reduce the number of ETI lookups. The intuition is as follows. Weights of input tokens (and hence weights of min-hash q-grams) often vary significantly. Therefore, we may look up the ETI on just a few important q-grams and—if a *fetching test* succeeds—*optimistically short circuit* the algorithm by fetching the current closest K reference tuples. If we are able to efficiently verify—via a *stopping test*—whether these tuples are actually the closest K tuples then, we can save a significant amount of work: (i) avoid ETI lookups on a number of unimportant q-grams, and (ii) avoid initializing and incrementing similarity scores in the hash table for large numbers of tids associated with unimportant high-frequency q-grams.

We illustrate the intuition using the input tuple I1, the reference relation in Table 1, and the ETI relation in Table 3. Suppose K=1, q=3, and H=2. The tokens along with weights in I1 are beoing:0.5, company:0.25, seattle:1.0, wa:0.75, 98004:2.0; their total weight is 4.5. Suppose their min-hash signatures are [eoi, ing]:0.25, [com, pan]:0.125, [sea, ttl]:0.5, [wa]:0.75, [980, 004]:1.0. For the purpose of this example, we ignore the adjustment terms. We order q-grams in the decreasing order of their weights, and fetch their tid-lists in this order. We first fetch the tid-list {R1, R2, R3} of q-gram '980.' We cannot yet distinguish between the K and (K+1)th (here, 1st and 2nd) best scores. So, we fetch the list {R1} of the next most important q-gram '004'. At this point, R1 has the best score of 2.0, and R2 and R3 have scores of 1.0. We now *estimate* (by extrapolating its current score) the score for R1 over all q-grams to be, say, 4.5. The best possible score s^2_{next} that R2 (the current K+1th highest score tid) can get equals its current score plus the sum of weights of all remaining q-grams: 1.0+ (4.5-2.0) = 3.5. Observe that s^2_{next} is also greater than the best possible (K+1)th similarity—as per fms^{apx} and hence fms—among all reference tuples in R. Because 4.5 > 3.5, we anticipate the reference tuple R1 to be the closest fuzzy match, fetch it from R, and compute fms(u, R1). If fms(u, R1) ≥ 3.5/4.5, we stop and return R1 as the closest fuzzy match thus avoiding looking up and processing tid-lists of q-grams: eoi, ing, com, pan, sea, ttl, wa. However, if fms(u, R1) ≤ 3.5, we continue fetching the next most important q-gram (here 'wa').

The robustness of the stopping test ensures that inaccuracy in estimating the score of R1 over all q-grams does not affect the correctness of the final result. However, it impacts performance. If we over-estimate we may fetch more reference tuples and realize they are not good matches, and if we under-estimate then we may perform a higher number of ETI lookups.

The query processing algorithm enhanced with optimistic short circuiting (*OSC*) differs from the basic algorithm in two aspects: (i) the order in which we look up q-grams against ETI, and (ii) the additional short-circuiting procedure we potentially invoke after looking up each q-gram. Pseudo code is almost the same as that in Figure 3 except for two additional steps: 3.1 (the ordering of tokens) and 9.1 (short circuiting procedure). We order Q the set of all q-grams in the min-hash signatures of an input tuple in the decreasing order of their weights, where each q-gram s in the signature mh(t) is assigned a weight w(t)/|mh(t)|. After fetching tid-list(s) (Step 8 in Figure 3) and processing tids in the tid-list (Step 9 in Figure 3), we additionally perform the *short circuiting procedure* (new Step 8.1 whose pseudo code is shown in Figure 4). If the short circuiting procedure returns successfully, we skip steps 10, 11, and 12.

The short circuiting procedure consists of a *fetching test* and a *stopping test*. The fetching test (Step 3 in Figure 4) evaluates whether or not the current K tids could be the closest matches. On failure, we return and continue processing more q-grams. On success, we fetch the current best K candidates from the reference relation R (Step 4), and compare (using fms) each of them with the input tuple u (Step 5). The stopping test (Step 6) confirms whether or not u is more similar to the retrieved tuples than to any other reference tuple. On success, we stop and return the current K candidate tuples as the best K fuzzy matches. On failure, we continue processing more q-grams.

We now describe the fetching and stopping tests. Let *w(Q)* denote the sum of weights of all q-grams in a set of q-grams Q. Let $Q_p=[q_1,\ldots,q_p]$ denote the ordered list of q-grams in min-hash signatures of all tokens in the input tuple u such that $w(q_i) \geq w(q_{i+1})$. Let Q_i denote the set of q-grams $[q_1,\ldots, q_i]$. Let $ss_i(r)$ denote the similarity score of the tid r plus the adjustment term after processing tid-lists of $q_1,\ldots,q_i$. Suppose $r^i_1,\ldots,r^i_K, r^i_{K+1}$ are the tids with the highest K+1 similarity scores after looking up q-grams $q_1,\ldots,q_i$. Informally, the fetching test returns true if and only if the "estimated overall score" of r^i_K is greater than the "best possible overall score" of r^i_{K+1}. We compute the estimated overall score of r^i_K by linearly extrapolating its current similarity score $ss_i(r^i_K)$ to $ss_i(r^i_K)\cdot w(Q_p)/w(Q_i)$, and the best possible overall score of r^i_{K+1} by adding the weight $(w(Q_p)-w(Q_i))$ of all q-grams yet to be fetched to $ss_i(r^i_{K+1})$.

$$\text{Fetching Test} = \begin{cases} \text{True}, & \dfrac{ss_i(r^i_K)}{w(Q_i)}\cdot w(Q_p) > ss_i(r^i_{K+1}) + (w(Q_p) - w(Q_i)) \\ \text{False, Otherwise} & \end{cases}$$

The stopping test returns successfully if $fms(u, r^i_j) \geq ss_i(r^i_{K+1}) + w(Q_p)-w(Q_i)$, for all $1 \leq j \leq K$. Since $ss_i(r^i_{K+1}) + w(Q_p)-w(Q_i)$ is the maximum possible overall score any candidate outside the current top K candidates can get, if the similarities (as per fms) are greater than this upper bound we can safely stop because we are sure that no other reference tuple will get a higher score. The following theorem (whose proof we omit) formalizes the guarantees of the algorithm. Again, for the purpose of obtaining the formal guarantee, we assume that no q-gram is classified as a stop token.

Theorem 2: Let $0 < \delta < 1, \varepsilon > 0, H \geq 2\delta^{-2} \log \varepsilon^{-1}$. The query processing algorithm enhanced with optimistic short circuiting returns the K reference tuples closest according to fms to the input tuple with probability at least $1-\varepsilon$.

```
BOOLEAN ShortCircuit_ETILookups(TidScores, TupleList)
//FetchingTest($s_K$, $s_{K+1}$)
1   Identify K+1 tids $r^i_1,\ldots,r^i_{K+1}$ with the highest similarity scores
2   Estimate the score $s^K_{opt}$ over $Q_p$ of $r^i_K$ and determine the best
    possible score $s^{K+1}_{best}$ over $Q_p$ of $r^i_{K+1}$
3   If $s^K_{opt} > s^{K+1}_{best}$
4       Fetch R tuples $r^i_1,\ldots,r^i_K$
5       Compare them with u to determine fms(u, $r^i_1$), …, fms(u, $r^i_K$)
//Stopping Test
6       If fms(u, $r^i_j$) $\geq s^{K+1}_{best}$ for all j, then assign TupleList =
        <$r^i_1,\ldots,r^i_k$> and return True; else, return false
```

Figure 4: Short-Circuiting Decision Procedure

4.4 Resource Requirements

We now discuss the resource requirements of the two phases of our algorithm: the ETI building and the query processing phases.

The expensive steps of the ETI building phase are: (1) scan of the reference relation R, (2) writing the pre-ETI, (3) sorting the pre-ETI, and (4) writing the ETI. The total I/O cost during these phases is $O(m_{avg}\cdot q\cdot H\cdot|R| + |ETI|\cdot(12+q))$ where m_{avg} is the average number of tokens in each tuple, and |ETI| is the number of tuples in ETI which is less than $H\cdot n\cdot|\sum|^q$—the maximum number of q-grams times H times the number of columns in R—given that $\sum$ is the alphabet over which tokens in R are formed from.

The expensive steps for processing an input tuple are: (1) looking up ETI for tid-lists of q-grams, (2) processing tid-lists, and (3) fetching tuples in the candidate set. The number of ETI lookups is less than or equal to the total number of q-grams in signatures of all tokens of a tuple. On average, this number is $m_{avg}\cdot H$. The number of tids processed per tuple and the size of the candidate set is bounded by the sum of frequencies of all q-grams in the signatures of tokens in a tuple. In practice, the candidate set sizes are several orders of magnitude less than the above loose upper bound. Due to its dependence on the variance of token weights of input tuples, the reduction in the number of ETI lookups due to OSC is hard to quantify.

4.4.1 Token-Frequency Cache

Thus far, we assumed that frequencies of tokens are maintained in a main memory token-frequency cache enabling quick computation of IDF weights. Given current main memory sizes on desktop machines, this assumption is valid even for very large reference relations. For example, a relation Customer[Name, city, state, zip code] with 1.7 million tuples has approximately 367,500 distinct tokens (even after treating identical token strings in distinct columns as distinct tokens). Assuming that each token and its auxiliary information (4 bytes each for column and frequency) together require on average 50 bytes, we only require 18.375 MB for maintaining frequencies of all these tokens in main memory. In those rare cases when the token-frequency cache does not fit in main memory, we can adopt one of following approaches.

Cache without Collisions: We can reduce the size of the token-frequency cache by mapping each token to an integer using a 1-1 hash function (e.g., MD5 [21]). We now only require 24 bytes of space (as opposed to a higher number earlier) for each token: the hash value (16 bytes), the column to which it belongs (4 bytes), and the frequency (4 bytes). Now, the token-frequency cache for the 1.7 million tuple customer relation requires only around 10MB.

Cache with Collisions: A less preferred option is to restrict the size of the hash table to at most M entries allowing multiple tokens to be collapsed into one bucket. The impact on the accuracy and correctness of our fuzzy matching algorithm depends on the collision probability. More the collisions, the more likely we will compute incorrect token weights.

In our experiments, we assume that the token-frequency cache fits entirely in main memory and hence do not measure the impact of collisions in a size-restricted token frequency cache on accuracy.

5. EXTENSIONS

We now discuss several extensions to the query processing algorithm and the fuzzy match similarity function.

5.1 Indexing Using Tokens

We now extend the ETI and the fuzzy match query processing algorithm to effectively use tokens for further improving efficiency. Consider the input tuple I1 [I1, Beoing Company, Seattle, WA, 98004] in Table 2. All tokens except 'beoing' are correct, and this characteristic of most tokens in an input tuple being correct holds for a significant percentage of input tokens. Tokens are higher level encapsulations of (several) q-grams. Therefore, if we also index reference tuples on tokens, we can directly look up ETI against these tokens instead of several min-hash signatures thus potentially improving efficiency of the candidate set retrieval. However, the challenge is to ensure that the candidate set we fetch contains all K fuzzy matching reference tuples. If we do not look up ETI on the q-gram signature of a token, say 'beoing', we may not consider reference tuples containing a token 'boeing' close to 'beoing'. And, it is possible that the closest fuzzy match happens to be the reference tuple containing 'boeing'. So, the challenge is to gain efficiency without losing accuracy.

Our approach is to split importance of a token equally among itself and its min-hash signature by extending the q-gram signature of a token to include the token itself, say, as the 0th coordinate in the signature. The extension modifies the similarity function fms^{apx} resulting in fms^{t_apx}. Under the broad assumption that all tokens in an input tuple are equally likely to be erroneous, the new approximation fms^{t_apx} resulting from the modification of the token signature is *expected to be a rank-preserving* transformation of fms^{apx}. That is, if v_1 and v_2 are two reference tuples, and u an input tuple then $E[fms^{apx}(u, v_1)] > E[fms^{apx}(u, v_2)]$ implies $E[fms^{t\_apx}(u, v_1)] > E[fms^{t\_apx}(u, v_2)]$. Consequently, the fuzzy matches identified by using fms^{t_apx} are the same as that identified by using fms^{apx}. Hence, we gain efficiency without losing accuracy. Lemma 5.1 formally states this result. We omit the proof due to space constraints.

Definition of fms^{t_apx}: Let u be an input tuple, v be a reference tuple, t and r be tokens, q and H be positive integers. Define

$$sim'_{mh}(t,r) = \frac{1}{2}\left(I[t=r] + \frac{1}{H}\sum_i I[mh_i(t) = mh_i(r)]\right)$$

$$fms^{t\_apx}(u,v) = \sum_{col} \sum_{t \in tok(u[col])} w(t) * \underset{r \in tok(v[col])}{Max} \left(\frac{2}{q} sim'_{mh}(t,r) + d\right)$$

Lemma 5.1: If the probability of error in an input token is a constant p ($0 < p < 1$), then fms^{t_apx} is a rank-preserving transformation of fms^{apx}.

The construction of the ETI index relation has to be modified to write additional tuples of the form [token, 0, column, tid-list]. We omit details of the ETI building and query processing, which are straight-forward extensions of the earlier discussion.

5.2 Column Weights

Our infrastructure can be extended to assign varying importance to columns while matching tuples. Let $W_1,\ldots,W_n$ be the weights assigned respectively to columns $A_1, \ldots, A_n$ such that $W_1+\ldots+W_n = 1$. A higher W_i value exaggerates the contribution due to matches and differences between attribute values in the ith column to the overall similarity score. The only aspect that changes is that of weights assigned to tokens during the query processing algorithm. Now, a token t in the ith column gets a weight $w(t)\cdot W_i$ where w(t) is the IDF weight and W_i is the column weight. The fuzzy match similarity function, the ETI building algorithm, and the rest of the query processing algorithm remain unchanged.

5.3 Token Transposition Operation

The fuzzy match similarity function may also consider additional transformation operations while transforming an input tuple to a reference tuple. We now consider one such operation: the *token transposition* operation which re-orders adjacent tokens.

Token transposition: Let $u[r, a_1,\ldots,a_n]$ be an input tuple. The token transposition operation transforms a token pair (t_1, t_2) consisting of two adjacent tokens in $tok(a_i)$ where t_2 follows t_1 into the pair (t_2, t_1). The cost is a *function* (e.g., average, min, max, or constant) $g(w(t_1), w(t_2))$ of the weights of t_1 and t_2. Because the token transposition operation only transforms the ordering among tokens the resulting similarity is still less (probabilistically) than fms^{apx}. Therefore, all the analytical guarantees of our fuzzy matching algorithm are still valid when we include the token transposition operation.

6. EXPERIMENTS

Using real datasets, we now empirically demonstrate (i) the quality of our new similarity function under a variety of commonly encountered errors, and (ii) the efficiency of our fuzzy matching operation.

6.1 Datasets and Setup

We start with a clean *Customer*[*name, city, state, zip code]* relation consisting of about 1.7 million tuples from an internal operational data warehouse as the reference relation. We create input datasets by introducing errors in randomly selected subsets of Customer tuples. Therefore, all characteristics of real data—e.g., variations in token lengths, frequencies of tokens—are preserved in the erroneous input tuples. We consider two types of error injection methods. The type I method introduces errors in tokens with equal probability, i.e., all tokens in a column are equally likely to become erroneous. The type II method introduces errors in tokens with a probability that is directly proportional to their frequency, i.e., tokens with higher frequency are more likely to become erroneous. This is a common phenomenon because the more frequently a token occurs the more likely it is to have erroneous versions, e.g., several different versions of the token 'corporation' are 'corp, co., corpn, inc.' Observe that the type II error injection method is biased towards fms because errors in low weight high frequency tokens do not significantly reduce fms similarity.

Table 4: Types and descriptions of errors

e_j	Description of Error	$P(e_j \mid u[i]\ error)$	
		i = 1	**$i \neq 1$**
1	*Spelling error*: modify token	0.5	0.4
2	*Token replacement*: replace commonly abbreviated tokens with abbreviations	0.25	0.25
3	*Missing values*: u[i] = null	0.0	0.1
4	*Truncation*: truncate u[i] by 5 or less characters	0.1	0.1
5	*Token merge*: remove delimiters in u[i]	0.1	0.1
6	*Token transposition*: reorder adacent tokens	0.1	0.05

As shown in Table 4, we associate with column i a probability p_i ($0 < p_i < 1$) with which we introduce errors into the value u[i] of tuple u. Error introduction across columns is independent. If we decide (with probability p_i) to introduce an error into u[i], we select from among several types of errors with conditional probabilities $P(e_j \mid u[i]\ error)$ shown in the table above. We do not introduce missing values in the name column as input tuples with a missing name cannot possibly be matched with their target. Hence, we have two conditional probability columns: one each for i=1 and $i\neq1$.

Metrics

We use the following evaluation metrics.

(1)*Normalized Elapsed Time*: the elapsed time to process the set of input tuples using the fuzzy match algorithm divided by the elapsed time to process one input tuple using the naïve algorithm (which compares an input tuple with each reference tuple). If the normalized time for a fuzzy match algorithm is less than the number of input tuples, then it outperforms the naive algorithm.

(2)*Accuracy*: The percentage of input tuples for which a fuzzy match algorithm identifies the seed tuple, from which the erroneous input tuple was generated, as the closest reference tuple is its *accuracy.*

Parameter Settings: In all our experiments, we set K=1 (i.e., we only retrieve the closest fuzzy match), the q-gram size q=4, the minimum similarity threshold c=0.0, and the token insertion factor (required for measuring fms) c_{ins}=0.5.

Machine Specifications: We ran experiments on a 930MHz Pentium machine with 256MB RAM running Microsoft Windows XP. We implemented our algorithm on the Microsoft SQLServer 2000 database system using OLEDB for database access.

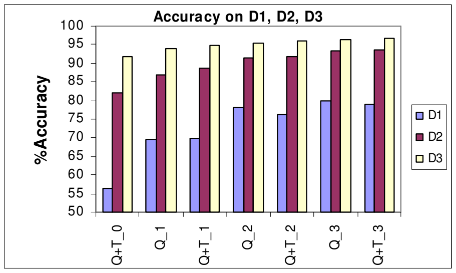


Figure 5: Accuracy

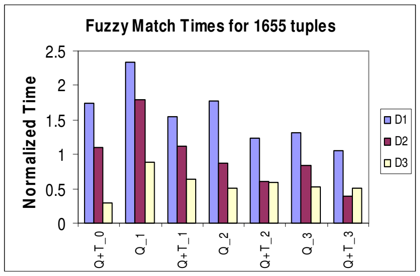


Figure 6: **Normalized Elapsed Times**

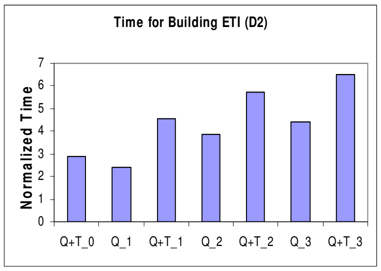


Figure 7: ETI Building Time

6.2 Experimental Results

In this section, we use the following notation to denote the approaches and parameters we evaluate. To denote the signature computation strategy, we use A_H, A∈ {Q, Q+T} and H ≥ 0. Q denotes q-grams only, and Q+T denotes q-grams plus token signatures as discussed in Section 5.1. H is the number of q-grams in the signature. For example, Q+T_2 is a signature with 2 q-grams and the token; Q+T_0 denotes a token only (no q-grams at all) signature.

6.2.1 Accuracy

We first compare the quality of ed and fms, and then evaluate accuracy of our fuzzy match algorithms.

6.2.1.1 Comparison between ed and fms

We show that the quality of fms is better than ed using two datasets: one created using Type I and the other using Type II error injection methods. Each one of these datasets has around 100 tuples. The probabilities of error in columns are 0.90, 0.5, 0.5, 0.6, respectively. Because we want to compare the quality of similarity functions and not the efficiency of algorithms for identifying the fuzzy matches, we use the naïve algorithm to identify the best fuzzy match for each input tuple.

	fms	**ed**
Accuracy on Type I	69%	63%
Accuracy on Type II	95%	71%

The adjacent table shows the accuracies of fms and ed on each dataset. We observe that fms is better than ed. As expected, it is significantly better for the dataset created with Type II errors than it is for the dataset with Type I errors. To study the cases that are not biased towards fms, we henceforth consider only datasets created with Type I error injection method.

Table 5: Error probabilities for creating datasets

Dataset	**Error Probabilities: [Name, City, State, Zip code]**
D1	[0.90, 0.90, 0.90, 0.90]
D2	[0.80, 0.5, 0.5, 0.6]
D3	[0.70, 0.5, 0.5, 0.25]

6.2.1.2 Accuracy of Algorithms

We evaluate the accuracy of various strategies on datasets D1, D2, and D3 generated using the type I error injection method. The error probabilities on each column for these datasets are shown in Table 5. Note that D3 is relatively cleaner than D2, which in turn is cleaner than D1. Each of D1, D2, and D3 has *1655 tuples*. The Customer relation which is the reference relation in all our experiments has approximately 2 million tuples. Figure 5 shows the results from which we observe the following.

(i) Min-hash signatures significantly improve accuracy: Q_H (for H>0) is more accurate (5% to 25%) than Q+T_0 (the tokens only approach).

(ii) Adding tokens to the signature does not negatively impact accuracy, because when H > 0, Q+T_H is as accurate as Q_H.

(iii) Even small signature sizes yield higher gains in accuracy: Q_2 is more accurate than Q_1, but the difference in accuracy between Q_2 and Q_3 is not significant.

6.2.2 Efficiency

To demonstrate the overall efficiency of our algorithms, we measure the *normalized elapsed time* for processing fuzzy match queries, the *number of candidate reference tuples* fetched per input tuple, and the *number of tids* processed per input tuple. To demonstrate the effectiveness of the optimistic short circuiting (OSC) optimization, we observe the numbers of reference tuples fetched per input tuple when OSC succeeded versus when it failed. Figure 6 shows the normalized elapsed times, from which we observe the following.

(i) Our algorithms are 2 to 3 orders of magnitude faster than the naïve algorithm: the normalized elapsed time of any of our strategies for processing all *1655* input tuples is less than 2.5. That is, they process all 1655 tuples before the naïve algorithm processes 2 or 3 tuples.

(ii) The query processing time decreases with the signature size. Even though we may have to look up ETI for more q-grams, the presence of more q-grams helps better distinguish differences between similarity scores of tids. Consequently, the average number of reference tuples fetched per input tuple decreases with signature size, also confirmed by Figure 8.

(iii) For all 1≤H≤3, Q+T_H is significantly faster than Q_H thus confirming our intuition (discussed in Section 5.1) that the use of tokens significantly improves efficiency of candidate set retrieval without compromising on accuracy.

We now discuss results on the average number of reference tuples fetched per input tuple (Figure 8), the average number of tids processed per input tuple (Figure 9) for D2. The results for D1 and D3 are similar. Again, Figure 8 shows that more q-grams help decrease candidate set sizes by better distinguishing similarity scores of tids. Even though, as shown in Figure 9, the average number of tids processed per input tuple increases, it is more than compensated by the average reduction in candidate set sizes.

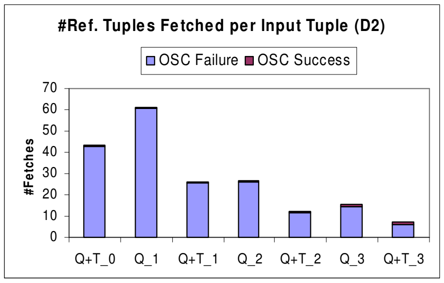


Figure 8: **Average Candidate Set Size**

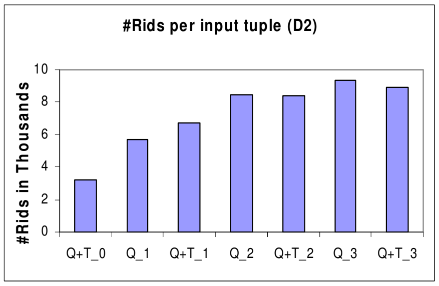


Figure 9: #Tids processed per Input Tuple

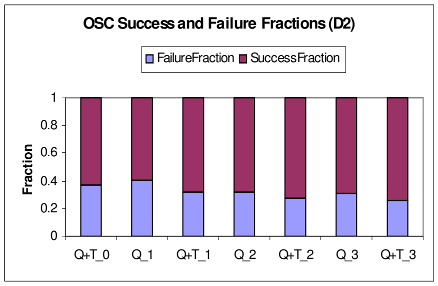


Figure 10: OSC Success Fractions

Figure 10 shows that the optimistic short circuiting (OSC) optimization is successful for 50%—75% of the input tuples, and the success fraction increases with signature size. Once again, we believe that this behavior is due to the higher distinguishing ability between similarities by using more q-grams. Figure 8 also splits the average number of reference tuples fetched into two parts: the number when OSC succeeds and the number when OSC fails. We observe that when OSC succeeds, we retrieve very few (around 1 per input tuple) candidate tuples. For those remaining tuples where OSC fails, we fetch a much larger number.

6.2.2.1 ETI Building Time

Figure 7 shows the normalized ETI building times for various settings. As expected the time for Q+T_H is greater than Q_H. Observe that the normalized time for any setting is less than 7. Thus, if we have more than 10 input tuples to fuzzy match, then it seems advantageous to build the ETI, and use our fuzzy match algorithm. Because we persist the ETI as a standard indexed relation, we can use it for subsequent batches of input tuples if the reference table does not change. Due to space constraints, we do not discuss ETI maintenance when the reference table changes.

7. CONCLUSIONS

In this paper, we generalized edit distance similarity by incorporating the notion of tokens and their importance to develop an accurate fuzzy match similarity function for matching erroneous input tuples with clean tuples from a reference relation. We then developed the error tolerant index and an efficient algorithm for identifying with high probability the closest fuzzy matching reference tuples. Using real datasets, we demonstrated the high quality of our similarity function and the efficiency of our algorithms.

REFERENCES

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of VLDB,* Hong Kong, 2002.

[2] R. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors. In R. Monge, editor, *Proceedings of the XXIII Latin American Conference on Informatics (CLEI'97)*, Valparaiso, Chile, 1997.

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

[4] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES '97)*, 1998.

[5] P. Ciaccia, M. Patella, P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. VLDB 1997.

[6] E. Cohen. Size estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 1997.

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

[8] 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.

[9] E. Cohen and D. Lewis. Approximating matrix multiplication for pattern recognition tasks. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[10] W. Cohen and J. Richman. Learning to match and cluster entity names. In *proceedings of SIGKDD*, Edmonton, July 2002.

[11] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170--231, 1998.

[12] 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 VLDB*, Roma, Italy, September 11-14 2001.

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

[14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing (STOC)*, 1998.

[15] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science*, 1991.

[16] R. Motwani and P. Raghavan. *Randomized Algorithms* Cambridge University Press, 1995.

[17] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19--27, 2001.

[18] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, 2000.

[19] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28--46, 2002.

[20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. *In Proceedings of ACM SIGKDD*, Edmonton, Canada, 2002.

[21] B. Schneier. *Applied Cryptography* John Wiley, 1996.

[22] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195--197, 1981.

[23] Trillium Software. http://www.trilliumsoft.com

[24] W. Winkler. The state of record linkage and current research problems. http://www.census.gov/srd/papers/pdf/rr99-04.pdf