On Indexing Error-Tolerant Set Containment

Parag Agrawal
Stanford University
paraga@cs.stanford.edu

Arvind Arasu
Microsoft Research
arvinda@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

ABSTRACT

Prior work has identified set based comparisons as a useful primitive for supporting a wide variety of similarity functions in record matching. Accordingly, various techniques have been proposed to improve the performance of set similarity lookups. However, this body of work focuses almost exclusively on *symmetric* notions of set similarity. In this paper, we study the indexing problem for the asymmetric *Jaccard containment* similarity function that is an error-tolerant variation of set containment. We enhance this similarity function to also account for string transformations that reflect synonyms such as "Bob" and "Robert" referring to the same first name. We propose an index structure that builds inverted lists on carefully chosen token-sets and a lookup algorithm using our index that is sensitive to the output size of the query. Our experiments over real life data sets show the benefits of our techniques. To our knowledge, this is the first paper that studies the indexing problem for Jaccard containment in the presence of string transformations.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Algorithms, Performance

Keywords

Data Cleaning, Indexing, Jaccard Containment, Transformations

1. INTRODUCTION

Data cleaning is an essential ingredient in the use of data warehouses for accurate data analysis. For example, owing to various errors in data, the customer name in a sales record may not match exactly with the name of the same customer as registered in the warehouse, motivating the need for *record matching* [20, 22]. A critical component of record matching involves determining whether two strings are similar or not. String similarity is typically captured via a similarity function that measures textual similarity.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

ID	Organization Name
1	`Madison Garden`
2	`Olive Garden Italian Restaurant, Madison WI 53701`
3	`Pizza Hut, Milwaukee WI`
...	...

Figure 1: Organization Table

Prior work has identified various measures of textual similarity — edit distance, Jaccard similarity, Jaro-Winkler distance, Hamming distance, each of which is applicable for different scenarios [20, 22]. In order to apply a string similarity function f for record matching, we need to perform efficient *similarity lookups* where given an input string s, the goal is to find all strings r in a reference relation R such that $f(r, s) > \theta$ for a given threshold $0 \leq \theta \leq 1$. Accordingly, specific indexing methods have been proposed for the above similarity functions [14, 20, 22]. However, a data cleaning *platform* is faced with the impractical option of implementing and maintaining a large suite of indexing techniques in order to support a variety of similarity functions.

Prior work has identified set similarity lookups as a primitive operation that can be used for supporting lookups based on several similarity functions. The idea is to model a string as a set of tokens. For example, edit distance can be indexed by building an index over the set of *q-grams* of a string [17]. This observation has led to the body of work on efficient techniques to perform lookups and joins based on set similarity [6, 10, 15, 19, 29, 31]. Most of these techniques focus on one specific form of set similarity, namely the *Jaccard coefficient* which measures the ratio of the size of the intersection to the size of the union. For example, the Jaccard coefficient between the sets {`Olive`, `Garden`} and {`Olive`, `Tree`} is 1/3.

However, the Jaccard coefficient is only one in a class of set-based similarity functions. In particular, it is a symmetric function. There are scenarios where an asymmetric notion of set-based similarity is more appropriate. Consider for instance, the reference relation shown in Figure 1 containing restaurant names. Assume that the strings are converted into sets by treating each space-delimited word as a token. Suppose that we wish to lookup the string `Olive Garden` in this relation. The record with id 1, `Madison Garden` has Jaccard coefficient 1/3 whereas the record with id 2, `Olive Garden Italian Restaurant, Madison WI 53701` which intuitively constitutes a better match has Jaccard coefficient 2/7 < 1/3. The main issue in this example is that the Jaccard coefficient penalizes record 2 for its longer length even though it contains all the tokens in the query string. While assigning token weights that vary inversely with frequency (such as idf) mitigates the problem described here with Jaccard coefficient, it does not eliminate it.

Motivated by the above limitation, *asymmetric* notions of set-based similarity have been proposed in prior work [8, 10]. Specifically, the *Jaccard containment* of a query set in a reference set is the ratio of the (weighted) size of the intersection of the two sets to the (weighted) size of the query set. In the above example, the Jaccard containment of `Olive Garden` with respect to record 1 is 1/2 whereas the containment in record 2 is 1. We note that Jaccard containment is a generalization of exact set containment which has been studied extensively in prior work [11, 24, 25, 28].

Besides being a useful similarity function for record matching, Jaccard containment has other applications. For example, error-tolerant lookups are gaining increasing importance online for matching user queries against addresses (map services) and products (products search). Since regular keyword search is based on set containment, Jaccard containment offers a natural error-tolerant alternative. Further, it is also applicable in performing fuzzy autocompletion [21, 23] where we have to identify matches from an underlying reference relation as the query string is being typed. In this paper, we study the problem of indexing lookups based on Jaccard containment.

Recent work [4, 14, 26] has recognized that textual similarity alone is inadequate in matching strings that are syntactically far apart but still represent the same real-world object. For example, the first name `Robert` can be written as `Bob`, or `United States of America` can be abbreviated to `USA`. The notion of *string transformations* has been identified in recent work [4, 14, 26] as a way of overcoming this inadequacy. Combining explicitly provided string transformations of the form `USA` $\rightarrow$ `United States of America` along with a core similarity function like Jaccard containment forms a new *programmable* similarity function that is influenced by the input set of transformations. Briefly, string transformations are used to generate "derived" strings from a single (query or data) string. For example, the string `Olive Garden Madison WI USA` can derive the string `Olive Garden Madison Wisconsin USA` under the transformation `WI` $\rightarrow$ `Wisconsin`. Queries are logically executed as though each of the derived queries is run over all strings derived from the reference relation. We study the indexing problem for Jaccard containment in the presence of an explicitly provided set of string transformations.

While the indexing problem for the special case of exact set containment has been studied extensively [11, 24, 25, 28], there has been little work on its error-tolerant variants. To our knowledge, this is the first paper that addresses indexing for Jaccard containment while also accounting for string transformations.

1.1 Challenges and Contributions

The point of departure for describing the solution proposed in this paper is the standard inverted index which is used to answer exact set containment queries by performing an intersection of the relevant lists [7]. This solution does not naively generalize for the case of Jaccard containment since a record can be in the output even if it does not occur in all lists. The presence of string transformations creates additional challenges since the number of derived strings generated by a given string can be large.

Our challenge is to design an indexing technique that can handle not only the Jaccard containment aspect (in contrast with exact containment) but also this potential explosion in the number of derived strings due to the presence of transformations.

We begin our contributions by identifying the main limitation of a standard token-level inverted index, namely that long lists adversely affect the query performance even in the special case of exact containment lookups. For instance, consider an exact containment query `This is it` posed over a table of movie names. Each of the tokens `This`, `is` and `it` is common so that their lists are expected to be long even though the output of query is likely to be small. This problem with a standard inverted index is exacerbated in the case of Jaccard containment since we cannot simply intersect all the lists.

Accordingly, we propose a parameterized index structure (Section 3) that, in addition to token-level lists, explicitly stores inverted lists for token-sets that are *minimal-infrequent*: the frequency is less than a given parameter (infrequent) whereas the frequency of any subset is larger than this parameter (frequent). (Only token-sets with non-empty intersections are considered.) The idea is that by building these additional lists, long lists are scanned only when the output of a query is large, hence making the performance sensitive to the output size. This index generalizes token-level inverted lists — when the value of the parameter is equal to the cardinality of the indexed relation, then the only minimal-infrequent sets are the singleton tokens.

We formally show that for the special case of exact containment queries, our index yields an output sensitive guarantee. We then discuss how the case of Jaccard containment with transformations can be addressed by running a collection of exact set containment queries (Section 4). However, the number of such "variant" queries can be large in the presence of transformations, and cause us to do redundant work. We then present algorithms that significantly reduce the amount of redundant work. In the presence of transformations, even computing the containment score between two strings becomes challenging. We propose an efficient algorithm for the same by reducing the problem to weighted bipartite matching.

Theoretically, the index size is in the worst case exponential in the record size. We note that the notion of minimal-infrequent sets is closely related to the classic data mining notion of maximal frequent item sets [2]. Extensive prior work [18, 27] has shown both analytically and empirically that the number of maximal frequent item sets in a database is unlikely to be exponential in the data string size in practice. Indeed, several algorithms to compute maximal frequent item sets are widely accepted as practical [16]. We argue formally that a similar intuition also applies to the size of the index we propose (Section 5).

We then conduct an empirical evaluation over various real-life data sets (Section 6) that show that our algorithms yield significant performance benefits without incurring an excessive space overhead. We also show that the index size in practice is far from the exponential worst case which is consistent with our theoretical analysis of the index size. We discuss related work in Section 7 and conclude in Section 8.

2. PRELIMINARIES

In this section, we formally define the Jaccard containment similarity function, the notion of string transformations and how they can be incorporated into Jaccard containment to define a new programmable similarity function. We also outline the challenges involved in indexing this function.

2.1 Strings as Sets

We model strings as sets. In general, our techniques are applicable irrespective of how strings are converted to sets. But in this paper, we model strings as a set of delimited tokens. The delimiters include white spaces and punctuation symbols. For example, the string `Olive Garden` represents the set {`Olive`, `Garden`}. We use strings to denote the sets and refer to the elements of the sets as *tokens*. Previous work has identified the need to associate weights with tokens [22]. For instance, the *inverse document fre-*

quency (idf) [22] may be used to associate a higher weight with rarer tokens and a lower weight with frequent tokens such as stop words. We thus assume that the sets are weighted, where every token e in the universe has a positive integral weight $wt(e)$. The weight of a set s, denoted $wt(s)$ is the sum of the weights of its tokens.

2.2 Jaccard Containment

Given two sets s_1 and s_2, the *Jaccard containment* of s_1 in s_2, denoted $JaccCont(s_1, s_2)$ is defined as $\frac{wt(s_1 \cap s_2)}{wt(s_1)}$. Consider for example the sets s_1 = `Olive Garden` and s_2 = `Madison Olive`. Suppose that all tokens have unit weight. Then $JaccCont(s_1, s_2) = 1/2$. We note that Jaccard containment is an asymmetric similarity function that generalizes exact set containment — if $s_1 \subseteq s_2$, then $JaccCont(s_1, s_2) = 1$.

2.3 String Transformations

As noted in Section 1, string transformations such as `Bob` $\rightarrow$ `Robert` are used to boost the similarity between strings that are textually far apart [4, 14, 26].

In this paper, we focus on transformations that are of the form `lhs` $\rightarrow$ `rhs` where each of `lhs` and `rhs` is a single token. Examples of such transformations are: `IL` $\rightarrow$ `Illinois`, `Grdn` $\rightarrow$ `Garden`, `St` $\rightarrow$ `Street` and `J` $\rightarrow$ `Jeffrey`. We note here that this restriction is merely to simplify presentation. The techniques presented in this paper can be extended to work for more general transformations.

We assume that the set of transformations is provided as part of the input either explicitly in a table (recent work [5, 11] has studied the problem of discovering transformations), or generated by a program. For instance, while a transformation like `Bob` $\rightarrow$ `Robert` may be materialized, transformations that expand abbreviations such as `J` $\rightarrow$ `Jeffrey` and transformations that account for token-level edit errors such as `Masachusets` $\rightarrow$ `Massachusetts` may be generated programmatically.

When we have a transformation $e \rightarrow e'$ we say that token e *derives* e'. The set of all tokens derived by e is called the set of *derivations* of e, denoted $\bar{e}$. A set s *derives* the set s', denoted $s \Rightarrow s'$ when s' can be obtained by starting with s and replacing some subset of its tokens $e_1, \ldots, e_k$ respectively with tokens $e'_1, \ldots, e'_k$ such that $e_i \rightarrow e'_i$. The set of *derivations* of a set s, denoted $\bar{s}$ is the collection of all sets derived by s.

Example 1. Consider the set of transformations `Drive` $\rightarrow$ `Dr` and `IL` $\rightarrow$ `Illinois`. The collection of sets derived by `Main Drive Chicago IL` is {`Main Drive Chicago IL`, `Main Dr Chicago IL`, `Main Drive Chicago Illinois`, `Main Dr Chicago Illinois`}. □

Notice that the number of derivations of s, $|\bar{s}|$ can be large: if every token of s derives two other tokens, then $|\bar{s}| = 3^{|s|}$.

2.4 Jaccard Containment With Transformations

A set of string transformations together with Jaccard containment can be used to define a new programmable similarity function as follows. Given a set of transformations $\mathcal{T}$, the Jaccard containment of set s_1 in set s_2 *under* T, denoted $JaccCont_{\mathcal{T}}(s_1, s_2)$ is defined to be the maximum containment $JaccCont(s'_1, s_2)$ among all s'_1 derived by s_1 using $\mathcal{T}$.

Example 2. Consider the sets s_1 = `Olive Grdn` and s_2 = `Olive Garden`. As noted above, the containment of s_1 in s_2 is 1/2. However, in the presence of the transformation `Grdn` $\rightarrow$ `Garden`, the containment of s_1 in s_2 becomes 1. □

We can trivially modify the above definition to allow transformations to be applied to *both* s_1 and s_2 and in general, different sets of transformations to be applied to s_1 and s_2. For instance, when we have a query set being looked up against a reference relation, we could apply programmatically generated transformations such as token-edits and expansions of abbreviations only on the query set while applying transformations that capture synonyms such as `Bob`$\rightarrow$`Robert` on both the query and the reference sets. For ease of exposition, unless otherwise stated, we will assume that the transformations $\mathcal{T}$ are applied *only* on the query and not on the reference relation. (Our techniques work with the most general case.)

Finally, we observe that one could consider other ways in which transformations may be used to program a similarity function. For instance, we could associate weights with transformations and have the overall similarity function reflect these weights. We believe the techniques presented in this paper can be extended to handle some of these alternate semantics. A detailed discussion of these alternatives is beyond the scope of this paper.

2.5 Indexing Problem

We are now ready to formally state the problem addressed in this paper. We are given a reference relation R consisting of sets and a set of transformations $\mathcal{T}$. Given an input query set q, the Jaccard containment lookup problem seeks all reference sets $r \in R$ such that $JaccCont_{\mathcal{T}}(q, r) \geq \theta$ for a specified similarity threshold θ. Our goal is to design an index structure and a query-time algorithm using said index structure to efficiently solve the lookup problem.

2.6 Inverted Lists, Signatures, Covering

The main data structure we use for indexing in this paper is the inverted list. However, we build inverted lists not only for single tokens but token sets. We now introduce some terminology that we use in the rest of the paper.

If we build an inverted list on a token-set, we say that we *index* the token set. A token-set that is indexed is called a *signature*. The list corresponding to a signature sig, denoted $List(sig)$ contains the record identifiers of all records in the reference relation R that contain the signature. We refer to standard token-level inverted lists as *token lists*. Inverted lists store only the record identifiers but we talk about them as though they contain the records themselves. Given a query set q, a signature sig is said to *cover* q if $sig \subseteq q$. If we pose q as an exact containment query, then $List(sig)$ is a superset of the result of q.

Finally, we talk about various kinds of sets in this paper. We refer to token sets in general as *tokensets*. Records in the reference relation which are a special case of tokensets are referred to as *records*. The input query and signatures are also tokensets and we refer to them respectively as *query* and *signature*. We also talk about collections of tokensets, such as the reference relation itself and sets of signatures. We use the term *set* when it is clear from the context which of the above sets we are referring to. We denote the cardinality of any set s as $|s|$.

2.7 Review of State of the Art

As noted in Section 1, most of the prior work has focused either on notions of set similarity that are symmetric (specifically, Jaccard similarity) or on exact subset containment which corresponds to the case where the lookup threshold is 1. The state of the art method for addressing Jaccard containment (without transformations) is a technique known as *prefix filtering* [10, 29] which we now review.

Consider a total ordering over all tokens in the universe. Given a weighted set r and $0 \leq \beta \leq 1$, define $prefix_\beta(r)$ as follows: sort the elements in r by the global token ordering and find the shortest

Algorithm 2.1 Prefix Filtering (No Transformations)

1: Create Index
Build token lists over reference relation R.
2: Lookup Algorithm for threshold θ
(1) Given query set q, compute $prefix_{1-\theta}(q)$.
(2) For each set $r \in \cup_{sig \in prefix_{1-\theta}(q)} List(sig)$
(3) If $(JaccCont(q, r) \geq \theta)$ then output r.

Algorithm 2.2 Basic Prefix Filtering With Transformations

1: Create Index
Build token lists over the sets in R
2: Lookup Algorithm for threshold θ
Given query set q, run Algorithm 2.1 over each of the sets in $\bar{q}$ and union the results.

prefix such that the sum of the weights of the tokens in the prefix exceeds $\beta \cdot wt(r)$. We have the following result [10, 29].

LEMMA 1. *If $JaccCont(s_1, s_2) \geq \theta$ then $prefix_{1-\theta}(s_1) \cap s_2 \neq \phi$.*

For example, suppose all elements have unit weights. If the Jaccard containment of the set `Olive Garden Restaurant` in set s is at least 0.6, then s must contain either `Olive` or `Garden` — here we are using the order in which we listed the set.

This insight can be used to develop an indexing scheme and lookup algorithm that are explained in Algorithm 2.1. The index used consists of standard token lists. Given a query set q and lookup threshold θ, the union of the lists corresponding to tokens in $prefix_{1-\theta}(q)$ is computed. For each record contained in the union, we check whether $JaccCont(q, r) \geq \theta$. If the token weights are obtained from their idf, then ordering them in descending order by weight to compute the prefix filter tends to access smaller lists.

The main limitation of prefix filtering is that the presence of long lists can substantially worsen the lookup time. Long lists are known to adversely affect the performance of even exact containment lookups where efficient algorithms for performing list intersection can be used to potentially skip over long lists. When performing the union of lists, skipping is no longer an option. Thus, long lists pose a serious challenge in indexing Jaccard containment.

The second issue is that it is not clear how to efficiently adapt prefix filtering in the presence of string transformations. The semantics of transformations suggests the algorithm described in Algorithm 2.2. The index structure is the same since we are discussing the case when transformations are applied only on the query. Given a query set q, we run Algorithm 2.1 for each set derived by q. Since the number of sets derived by q can be large as seen in Section 2.3, this algorithm can be highly inefficient. We address the above challenges in the following sections.

3. INDEX

In this section, we describe our index structure. In order to motivate our design, we interleave a discussion of how we use the index to answer the special case of exact containment queries. We also describe the index construction algorithm. We defer a discussion of the index size and its incremental maintenance to Section 5.

3.1 Minimal Infrequent Tokensets

We focus on creating signatures for queries whose output size is small, while the inverted lists for individual terms are all large. The rationale is that these are the queries for which the benefit of materialization is high. We formalize this intuition by using the notion of *minimal infrequent* tokensets. The index requires a frequency threshold a as an input parameter. Let R be the reference relation being indexed.

DEFINITION 1. *The frequency of a tokenset s, $freq(s)$ is the number of records in R that contain s. A tokenset s is said to be a-*frequent *if $freq(s) > a$. If $freq(s) \leq a$, then s is called a-*infrequent*. Tokenset s is said to be a-*minimal infrequent *if it is a-infrequent whereas every proper subset $s' \subset s$ is a-frequent.*

We note that when we talk about the frequency of a query set q, we mean $freq(q)$; not to be confused with the popularity of q. When the frequency threshold a is clear from the context, we drop the reference to a. We refer to the collection of all a-minimal infrequent tokensets as the *a-infrequent border* of R.

Example 3. Consider the collection R shown on the left in Figure 2. Suppose that $a = 1$. Then the tokensets `Olive` and `Garden` are frequent. The tokenset `Olive Garden` is minimal infrequent whereas `Olive Garden Restaurant` is infrequent but not minimal infrequent. □

3.2 Exact Lookup For a-Infrequent Queries

Any a-infrequent tokenset must contain an a-minimal infrequent subset (possibly itself). Suppose that we index all non-empty a-minimal infrequent tokensets. Any exact containment query q which is a-infrequent is covered by some a-minimal infrequent signature sig. In order to answer q, it suffices to verify for each record r in $List(sig)$ whether $q \subseteq r$. Since $|List(sig)| \leq a$, we obtain the following result.

PROPERTY 1. *If we index all non-empty a-minimal infrequent tokensets, then any a-infrequent query can be answered by fetching at most a records.*

Property 1 quantifies our goal of having an output-sensitive lookup performance and motivates our index design.

The question arises how we find the a-minimal infrequent signature that covers the given a-infrequent query q. We can perform this in time linear in $|q|$ as follows. We process the tokens of q in a fixed order. We keep a token e if dropping it makes the resulting tokenset a-frequent. Otherwise, we drop e. This process is sketched in Algorithm 3.1. In order to implement Algorithm 3.1, we maintain a *frequency oracle* that stores the set of all a-frequent tokensets with their frequencies. The frequency oracle can be used to check whether a given tokenset is a-frequent. We note that the tokenset returned by Algorithm 3.1 is a-infrequent and dropping any single element from the tokenset makes it a-frequent. Thus, we obtain the following result.

LEMMA 2. *Given an a-infrequent query q, Algorithm 3.1 finds an a-minimal infrequent covering signature in time linear in $|q|$.*

Example 4. Consider the collection R on the left in Figure 2. Suppose that $a = 1$ and we are given the query `Olive Garden`

Algorithm 3.1 Linear Time Algorithm to Find a Covering Signature

Input: An a-infrequent set q;
Output: An a-minimal infrequent (signature) set contained in q
1: Fix an order of the tokens in q
2: Let $sig \leftarrow q$
3: For each token $e \in q$ in the fixed order
4: If $sig - e$ is a-infrequent
5: $sig \leftarrow sig - e$
6: Return sig

Restaurant. Suppose that we run Algorithm 3.1 processing the tokens in reverse lexicographical order. We first consider dropping Restaurant. The remaining set Olive Garden is a-infrequent so we drop Restaurant. Dropping Garden would result in Olive which is a-frequent so we keep Garden. Similarly, we also keep Olive. We thus end up with the minimal-infrequent tokenset Olive Garden. □

3.3 Series of Parameters

In general, we need to handle not only a-infrequent queries but also a-frequent queries. The main observation we use to handle frequent queries is that any a-frequent tokenset is minimal-infrequent for *some* frequency threshold a'. We therefore partition the frequency space via a geometrically increasing series of parameters $a, 2a, 4a, \ldots, |R| = N$.[1] In our final index, we index minimal infrequent sets with respect to each of these parameters. In this index, any query set q is infrequent with respect to at least some $2^i \cdot a$. We can then use a $2^i \cdot a$-minimal-infrequent set contained in q to answer the exact containment query q.

ID	Org. Name
1	Madison Olive Oil
2	Olive Garden Italian Restaurant
3	Pizza Hut
4	Bamboo Garden

a	a-Minimal Infrequent Set (Signature)	Inverted List
1	Madison	{1}
1	Oil	{1}
1	Italian	{2}
1	Restaurant	{2}
1	Pizza	{3}
1	Hut	{3}
1	Bamboo	{4}
1	Olive Garden	{2}
2	Olive	{1, 2}
2	Garden	{2, 4}

Figure 2: Organization Table and Signatures

Figure 2 shows an example relation and its corresponding index for the series with $a = 1$. In general, a signature can be minimal-infrequent for more than one value in the parameter series. In this case, we store its list just once. Thus, in Figure 2, there are no signatures corresponding to parameter value 4 since they are subsumed by the smaller parameter values.

Observe that when we set $a = N = |R|$, the minimal infrequent sets are exactly the single tokens. Thus, the index described above generalizes a standard token-level inverted index.

3.4 Exact Lookup For a-Frequent Queries

Given the index defined in Section 3.3, we can apply Algorithm 3.1 also to answer exact containment queries q that are a-frequent. The only additional step is to find the smallest i such that q is $(2^i \cdot a)$-infrequent. We use the frequency oracle to check for a given tokenset and a given value of j whether it is $(2^j \cdot a)$-frequent. We then run Algorithm 3.1 to find a $(2^i \cdot a)$-minimal infrequent signature sig. We note that time taken to compute sig is linear in $|q|$. We then verify for each record in $List(sig)$ whether it contains q. We thus obtain the following result:

THEOREM 1. *A exact containment query q with output size o can be answered by processing at most $\max(a, 2 \cdot o)$ reference records.*

Thus, our index yields an output-sensitive guarantee for exact containment queries. This guarantee illustrates the benefits of indexing minimal-infrequent tokensets. Of course, our goal is to handle the problem of Jaccard containment lookups in the presence of transformations. We will discuss this problem in Section 4.

[1]The multiplicative factor 2 could in general be another input parameter c. We use $c = 2$ in the rest of the paper.

3.5 Index Construction

The notion of minimal infrequent sets is closely related to the notion of maximal frequent item sets [2]. Efficient algorithms to compute frequent item sets have been extensively studied [18]. We can adapt any of these algorithms to efficiently compute both the frequency oracle as well as our index. We can piggy-back the computation of the minimal infrequent sets with respect to the parameter settings $2a, 4a, \ldots$ over the computation of the a-minimal-infrequent sets. We omit these details from this paper since they are straightforward adaptations of prior work. We discuss the index size and its incremental maintenance in Section 5.

4. JACCARD CONTAINMENT LOOKUP

In this section, we discuss the lookup algorithm for Jaccard containment with transformations. We formulate an optimization problem for which we present two algorithms.

4.1 Query Variants

An error-tolerant set containment query can be thought of as representing a collection of exact containment queries. For example, suppose that we are performing a Jaccard containment lookup without transformations where the query set is q and the lookup threshold is θ. Define the collection of minimal subsets of q whose weight is greater than or equal to $\theta.wt(q)$ as the *Jaccard variants* of q, denoted $JaccardVariants_\theta(q)$. The lookup can be executed by finding all Jaccard variants of q, issuing an exact containment query corresponding to each of these variants and taking the union of the results.

Example 5. Consider the query q = Olive Garden Restaurant. Suppose that the weights of the elements are 7, 2 and 1 respectively. If the query threshold is $\theta = 0.8$, then answering q is equivalent to answering the exact containment queries Olive Garden and Olive Restaurant and taking the union of their results. □

This observation generalizes even in the presence of transformations.

Example 6. Consider the query q = Olive Grdn Restaurant where the weights of the elements are 7, 2 and 1 respectively. Suppose that we have a transformation Grdn → Garden and the weight of Garden is also 2. If the query threshold is $\theta = 0.8$, then q is equivalent to the union of the exact containment queries Olive Grdn, Olive Garden and Olive Restaurant. □

We formalize the above intuition by introducing the notion of the *variants* of a query.

DEFINITION 2. *Given a query set q and a lookup threshold θ, we define $Variants_\theta(q) = \bigcup_{q \Rightarrow q'} JaccardVariants_\theta(q')$. (Recall that we use $q \Rightarrow q'$ to denote that q' can be derived from q by applying transformations.)*

We observe that when the threshold $\theta = 1$, then the variants of q are exactly the sets derived from q via transformations.

PROPERTY 2. *$JaccCont_\mathcal{T}(q, r) \geq \theta$ if and only if there is some $q_{var} \in Variants_\theta(q)$ such that $q_{var} \subseteq r$. Therefore, q can be answered by issuing an exact containment query for each $q_{var} \in Variants_\theta(q)$ and taking a union of the results.*

Algorithm 4.1 Covering Based Lookup Framework

Input: Query set q and lookup threshold θ.
Output: All records $r \in R$ such that $JaccCont_{\mathcal{T}}(q, r) \geq \theta$
1: Find the matching transformations for q
2: Find $C \subseteq Signatures(\bar{q})$ that is a variant-covering of q
3: For each record $r \in \cup_{sig \in C} List(sig)$
4: If $(JaccCont_{\mathcal{T}}(q, r) \geq \theta)$
5: Output r

The main issue with issuing a separate (exact containment) query per $q_{var} \in Variants_{\theta}(q)$ as suggested by Property 2 is that it fails to account for the fact that a single signature can cover multiple variants, and hence potentially accesses many lists redundantly.

Example 7. Consider the query q = `Olive Grdn Restaurant` of Example 6. The signature `Olive` covers all the variants `Olive Grdn`, `Olive Garden` and `Olive Restaurant`. □

4.2 Covering Framework

In general, what we seek is a collection of signatures that cover *all* variants. This observation suggests an optimization problem which we introduce next.

DEFINITION 3. *Given a set q, define $Signatures(q)$ to be the collection of subsets of q that are indexed. (Exactly those subsets of q that are minimal infrequent for some index parameter $a.2^i$ are included.) We also define $Signatures(\bar{q}) = \bigcup_{q \Rightarrow q'} Signatures(q')$.*

We note that $Signatures(\bar{q})$ can be computed using the frequency oracle by using algorithms to compute minimal infrequent item sets, similar to the index build algorithm described in Section 3.5.

DEFINITION 4. *A subset of $C \subseteq Signatures(\bar{q})$ is a* variant-covering *if each variant in $Variants_{\theta}(q)$ is covered by some signature in C.*

For instance, in Example 7, the signature `Olive` is a variant-covering.

We outline a lookup framework based on the notion of variant-covering in Algorithm 4.1. The idea is to first find a variant-covering of the query, access the corresponding lists and verify for each of the retrieved records whether the similarity threshold is satisfied. In general, there are multiple variant coverings. For a given covering C, the number of record identifiers retrieved in Step 3 is the size of the union of the inverted lists corresponding to the signatures in C. Our goal is to find a covering such that this number is minimized. We use the sum of the list sizes as a proxy for the size of the union.

DEFINITION 5. *The* cost *of a covering C is the sum of the sizes of the lists corresponding to signatures in C.*

Covering Optimization: Among all variant-coverings of q, find the one with least cost.

We model this optimization problem as a bipartite graph where we have the signatures of the query on the one side and the query variants on the other. An edge connects a signature to a variant if the signature covers the variant. A variant-covering is a subset of signatures such that all variants are incident to them. Our goal is to find a minimum cost variant-covering. Figure 3 illustrates the bipartite graph for Example 5. The numbers in the parentheses of the signature sets indicate the lengths of their lists. We now describe two solutions to this optimization problem.

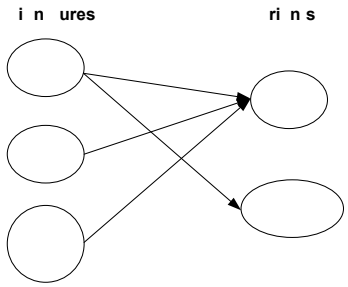


Figure 3: Illustrating Covering Optimization Problem.

Algorithm 4.2 Greedy Algorithm to Solve Covering Optimization

Input: Query set q and lookup threshold θ.
Output: $C \subseteq Signatures(\bar{q})$ that is a variant-covering of q
1: $C \leftarrow \phi$
2: While (there is an variant uncovered by C)
3: Pick the signature with maximum benefit-cost ratio
(Benefit of signature sig = number of variants uncovered by C but covered by sig) (Cost of $sig = |List(sig)|$)
4: $C \leftarrow C \cup sig$

4.3 Set Cover Approach

It is easy to see that the covering optimization problem is closely related to the set cover problem: we need to cover variants using signatures with the minimum cost. We can thus invoke the well-known greedy approximation algorithm to set cover to solve our covering problem. We sketch this algorithm in Algorithm 4.2. Here, we compute all variants $Variants_{\theta}(q)$ and all signatures $Signatures(\bar{q})$ *explicitly* and then add signatures in the order of their benefit-to-cost ratio. The benefit of a signature is the number of variants covered by it while the cost is the length of its corresponding list.

4.3.1 Analysis

Algorithm 4.1 coupled with the method sketched in Algorithm 4.2 to find the covering has formal guarantees for the index we proposed in Section 3 that we now discuss. Consider the "ideal" index that indexes all tokensets. Let us denote the cost of the optimal covering over this hypothetical index as $Cost_{TrueMin}$. This cost represents the minimum cost required to answer the query under our covering framework. Suppose that the cost of the solution returned by the greedy algorithm is $Cost_{Greedy}$. We have the following result:

LEMMA 3. *For a query q and lookup threshold θ:*

$$Cost_{Greedy} \leq Cost_{TrueMin} \times \max(a, 2) \times O(\lg |Variants_{\theta}(q)|$$

The factor $\max(a, 2)$ arises since we don't materialize all tokensets, while $O(\lg |Variants_{\theta}(q)|$ arises due to the greedy approximation to set cover that is used. We note that the upper bound yielded by the above result is a function of a. Even though this guarantee is weak for typical values of a like $a = 100$, it is a worst-case guarantee. The actual gap from $Cost_{TrueMin}$ is likely to be smaller in practice.

4.3.2 Limitations

The main limitation of Algorithm 4.2 is that it computes the covering by explicitly enumerating *all* signatures and *all* variants. As observed in Section 2.3, a query may have a large number of variants and signatures if a lot of transformations match the query. In

Algorithm 4.3 Hitting Set Algorithm for Covering Optimization

Input: Query set q and lookup threshold θ.
Output: $C \subseteq Signatures(\bar{q})$ that is a variant-covering of q
1: Let C denote the current covering. $C \leftarrow \phi$.
2: Let $Hit(C)$ denote the minimal hitting sets of C.
$Hit(C) \leftarrow \{\phi\}$
3: $\widetilde{Hit}(C) \leftarrow$ complements of sets in $Hit(C)$
4: If no $\widetilde{h} \in \widetilde{Hit}(C)$ contains a variant, stop and return C
5: Else find an uncovered variant and a signature sig that covers it
6: Add sig to C
7: Compute $Hit(C)$ and go to step 3

such cases, computing the covering using Algorithm 4.2 becomes prohibitively expensive. Thus, even though the cover returned by the set cover approach may contain few signatures and have a low cost, the cost paid in computing it may be significant. We encounter this situation in our experimental study in Section 6. We thus propose an alternate algorithm that generates a covering without requiring an enumeration of all variants of q.

4.4 Hitting Set Approach

Before we describe the alternate lookup algorithm, we discuss some preliminaries. In the context of a query q, we have a universe of tokens namely the set of all tokens in $\bar{q}$ (that is, all tokens in q and all tokens on the right hand side of transformations that match q). We can use this universe to define the *complement* of a variant which includes all tokens from the universe that are not in the variant. We refer to the complement of variant q_{var} as $\widetilde{q}_{var}$.

It follows that if a signature does *not* cover a variant, then the complement of the variant intersects (*hits*) the signature; in other words it is a *hitting set* of the signature. We can generalize this argument to show the following property:

PROPERTY 3. *A variant $q_{var} \in Variants_\theta(q)$ is not covered by a collection of signatures if and only if its complement $\widetilde{q}_{var}$ hits each signature in the collection.*

Example 8. Consider the query `Olive Garden Italian Restaurant 53701`. Suppose that (1) all tokens have weight 1, (2) we have no transformations and (3) the containment threshold is 0.5. The variant `Italian Restaurant 53701` is not covered by the collection of signatures {`Olive Italian`, `Garden Restaurant`}. The complement of the above variant is `Olive Garden` which hits both the signatures in the collection. □

Suppose that we have a collection of signatures. Based on Property 3, we can check if all variants of q are covered by (1) enumerating all minimal hitting sets of the signature collection, (2) for each minimal hitting set, checking if its complement contains a variant.

Example 9. Continuing Example 8, we note that `Olive Garden` is a hitting set for the signature collection {`Olive Italian`, `Garden Restaurant`}. Its complement contains `Italian Restaurant 53701`, a variant of the query. □

If the complement of some hitting set contains a variant, we add a signature that covers this variant to our collection (using Algorithm 3.1) and proceed.

Example 10. In Example 9, we may add a signature, say `Italian 53701` that covers the variant `Italian Restaurant 53701`. □

The lookup algorithm based on this intuition is outlined in Algorithm 4.3. Step 4 is straightforward and we omit its details owing to lack of space. Step 5 can be implemented using the linear time algorithm presented in Algorithm 3.1.

4.4.1 Analysis

We note that Algorithm 4.3 does not need to compute all variants and signatures for the query like the set cover algorithm. Instead, the time required in computing the variant-covering is dependent on the number of signatures in the covering. Hence, in cases where the number of signatures in the variant-covering is small relative to the total number of signatures and variants for the query, the hitting set algorithm is likely to outperform the set cover algorithm in terms of the time required to compute the covering. Our experiments in Section 6 demonstrate this phenomenon. We do note however that hitting set is known to be a hard problem, and hence the worst case bound for this algorithm is not polynomial (a superpolynomial bound may be shown in a manner similar to [18]).

Although Algorithm 4.3 is heuristic in terms of the cost of the covering that it computes, it it can be adapted to reflect the benefit-cost ratio order in which the greedy set-cover based algorithm proceeds. In order to provide a higher preference to signatures that cover more variants, we order the tokens in increasing order of their weight in Step 5 before running the linear algorithm. In Section 6, we will study how the hitting set covering algorithm compares with the set cover algorithm both in terms of signature generation time as well as in terms of the cost of the output covering.

4.4.2 Implication for Prefix Filtering

We can show that in the absence of transformations and when we only have token-inverted lists, Algorithm 4.3 reduces to prefix filtering. Thus, Algorithm 4.3 yields a generalization of prefix filtering to handle transformations. This is how we implement prefix filtering in our experiments. We also observe that the problem of finding the hitting set of a collection of singleton signatures is trivial — there is only one hitting set, namely the union of the singletons. Thus, the complexity of Algorithm 4.3 is polynomial in the number of elements in all sets in $\bar{q}$. In contrast, Algorithm 4.2 requires enumeration of all sets in $Variants_\theta(q)$.

4.5 Computing Similarity Score

We now discuss Step 4 in Algorithm 4.1 that checks whether $JaccCont_\mathcal{T}(q, r) \geq \theta$. So far in this section, we have considered applying transformations only on the query. Under this assumption, checking whether $JaccCont_\mathcal{T}(q, r) \geq \theta$ is trivial. All techniques presented so far can be extended in a straightforward manner to also handle transformations applied on the reference relation. However, checking whether $JaccCont_\mathcal{T}(q, r) \geq \theta$ becomes challenging — the definition of $JaccCont_\mathcal{T}$ requires us to consider all sets in $\bar{q}$ and $\bar{r}$. Just as in solving the covering optimization problem above, our goal is to avoid enumerating the variants in $\bar{q}$ and $\bar{r}$.

We show here how we can reduce our problem of checking if $JaccCont_\mathcal{T}(q, r) \geq \theta$ by reducing it to the problem of finding the maximum matching in a *weighted* bipartite graph. We know that for two tokensets s_1 and s_2, $JaccCont(s_1, s_2) \geq \theta \Leftrightarrow wt(s_1 \cap s_2)(1-\theta) + wt(s_1 - s_2)(-\theta) \geq 0$. Based on this observation, we construct a bipartite graph with nodes corresponding to tokens of q on one side, nodes corresponding to the tokens of r on the other and add an edge between a node corresponding to tokens $e_1 \in q$ and $e_2 \in r$ if either (1) $e_1 = e_2$, with corresponding weight $wt(e_1)$ or (2) $e_1 \rightarrow e_2$ with weight $wt(e_2)$ or vice versa, or (3) there is some e_3 with $e_1 \rightarrow e_3$ and $e_2 \rightarrow e_3$; the edge weight here is $wt(e_3)$. In general we could add an edge between two nodes through more than one of the above ways. In this case, we assign the maximum weight to the edge.

We add a new set of "mirror" nodes to the side corresponding to r. This is to account for nodes in q that are unmatched. Hence, there is one mirror node for each node in q. For token $e_1 \in q$ let

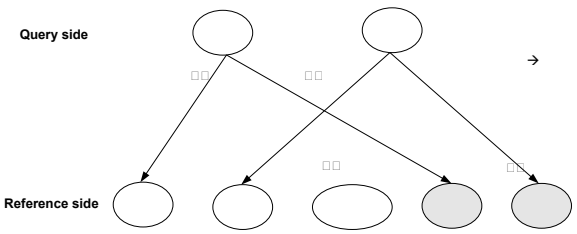


Figure 4: Reduction to Weighted Bipartite Matching.

$minwt$ denote the minimum weight among all tokens derived by e_1. We add an edge that connects $e_1 \in q$ to its mirror node with weight $minwt \times (-\theta)$. Since the algorithm for weighted bipartite matching assumes that weights are positive, we make all edge weights positive by adding a suitably large number M. Figure 4 shows an instance of the bipartite graph as constructed for the pair q = `Olive Grdn` and r = `Olive Garden`. The token weights and relevant transformations are shown in the figure. The shaded nodes indicate the mirror nodes. We show the following result:

LEMMA 4. *The weight of the maximum matching in the (weighted) bipartite graph constructed above is at least $M|q|$ if and only if $JaccCont_{\mathcal{T}}(q, r) \geq \theta$.*

5. OTHER INDEX PROPERTIES

We now discuss the properties of the index we proposed such as its size and incremental maintenance.

5.1 Index Size

One of the main questions around the index we described above relates to its size. In the worst case, the number of minimal-infrequent sets is exponential in the record size. We now discuss why we expect the index size to be much smaller in practice.

First, we note that the notion of minimal infrequent sets is closely related to the data mining notion of maximal frequent item sets [2]. Extensive prior work [18, 27] has shown both analytically and empirically that the number of maximal frequent item sets in a database is unlikely to be exponential in the record size. Indeed, based on this intuition, there are previously proposed algorithms to compute the set of maximal frequent item sets that are widely accepted as practical [16]. We argue that a similar intuition applies to minimal infrequent sets. Along the lines of prior work on frequent item set mining [27], we characterize the index size by using the concept of the *Vapnik Chervonenkis (VC) dimension*.

DEFINITION 6. *Fix a collection of sets R drawn from domain $\mathcal{D}$. A set $s \subseteq \mathcal{D}$ is said to be* shattered *by R if the collection $\{s \cap r : r \in R\}$ is the power-set of s. The* VC-Dimension *of R is the size of the largest set that can be shattered.*

Example 11. Consider the collection R = {r_1 =`Pizza Hut`, r_2 =`Madison Olive`, r_3 =`Bamboo Garden`, r_4 = `Olive Garden Italian American Restaurant`}. The set s = `Olive Garden` is shattered by R since it is contained in r_4, and we can obtain `Olive` by intersecting s with r_2, `Garden` by intersection s with r_3 and ϕ by intersecting s with r_1. No set of size 3 can be shattered by R. Thus, its VC-dimension is 2. □

Intuitively, a larger VC-dimension indicates a larger degree of correlation among the tokens. We obtain the following result:

LEMMA 5. *For reference relation R, consider the collection $\mathcal{C}_a(R)$ consisting of all a-minimal infrequent sets and all a-frequent sets. Both the index construction time as well as the index size are worst-case exponential in the VC-Dimension of $\mathcal{C}_a(R)$.*

This result means that the index size is correlated with the degree of inter-token correlation. Prior empirical work [27] has shown that the VC-dimension of various real life data sets is small.

While Lemma 5 provides an upper bound on the worst case of the index size, the index size could potentially be impractical even when the VC-dimension is small. For instance, suppose that for a database with average set size 10, the VC-dimension is 7. It is possible that a large fraction of size 7 sets get indexed. The number of such sets could be as large as $\binom{10}{7} = 120$ times the cardinality of the relation.

We thus need to understand how many minimal infrequent sets there could be among all token sets of size d. We now provide analytical insight into how this number behaves. We show that for data that is generated independently and uniformly at random, the expected number of minimal-infrequent sets that are added to the index decreases sharply as the set size increases.

LEMMA 6. *Consider a relation R consisting of N sets each of size l, generated independently and uniformly at random from a domain of size D. The expected number of a-minimal infrequent sets of size $d+1$ is at most $D \times \frac{(Nl^d)^a}{D^{(a-1)d}}(\frac{e}{a})^a$. (Here, e refers to the base of the natural logarithm.)*

Suppose that $N = 10^7$ and $l = 10$ with $D = 10^6$. These numbers are drawn from the real-life data set `Places` consisting of addresses and landmarks which we empirically study (see Section 6). Then, by Lemma 6, the expected number of sets of size 3 that are minimal infrequent for $a = 100$ is at most $10^6 \times \frac{10^{1100}}{10^{99*12}}(\frac{e}{100})^{100} << 1$.

Of course, the result in Lemma 6 assumes uniformity and independence both of which are unlikely to hold in real-life data. However, the above analysis does hold out the hope that the worst case behavior of the index size is unlikely to arise in practice. Finally, we note that the index is parameterized and the parameter can be used to control its size. As a special case, when $a = N$, we reduce to token-lists.

5.2 Generalization

Despite the analysis presented above, we need additional methods to control the index size should any of the above assumptions get violated. For example, even though our index is not designed for large document corpora, it would be useful to have methods that can help us realize our index for such a scenario. We now discuss some such methods.

The idea of minimal infrequent sets can be generalized as follows. Define a property P over sets to be *monotonic* if: whenever set s satisfies P, every subset of s also satisfies P. The predicate $freq(s) > a$ is an example of a monotonic predicate. For a relation R, a tokenset s belongs to the *negative border* with respect to a monotonic predicate P if s fails to satisfy P but every proper subset of s does. We note that the negative border with respect to the predicate $freq(s) > a$ is exactly the collection of a-minimal infrequent sets.

We could therefore build inverted lists for tokensets that lie on the negative border of alternate monotonic predicates. The index construction algorithm as well as the lookup algorithm are applicable for the negative border of any monotonic predicate. While the predicate $freq(s) > a$ is useful for indexing as we have discussed earlier in the paper, we can control the index size using alternate monotonic predicates. We outline two such possibilities.

First, we can restrict the cardinality of sets being indexed via the monotonic predicate $freq(s) > a \wedge |s| \leq l$. This ensures that sets above size l will not even be considered for indexing.

Second, the frequency $freq(s)$ does not have to be measured over the reference relation. We can for instance measure the frequency of a set in a relation obtained by considering the q-grams of the strings instead of the entire strings. This does not affect correctness since the predicate $freq(s) > a$ is monotonic no matter which relation the frequency is measured over. However, it significantly impacts the index size since the record sizes in the q-gram relation are no larger than q. At the same time, we get the effect of targeting performance benefits for queries that are contained within a window of size q of the reference records.

The above methods are by no means exhaustive. As noted above, so long as the predicate is monotonic, our lookup algorithms are guaranteed to be correct. We use a combination of the above methods to control the index size in our implementation. Our experiments over various real-life data sets show the success of using these methods (Section 6). In the rest of the paper, we refer to our index as the *negative border* index.

5.3 Index Updates

Now we discuss the problem of incrementally maintaining the index we have proposed in this paper. When a new record r is added to (or deleted from) the reference relation, we can simply add (respectively delete) its record identifier to all sets in $Signatures(\bar{r})$ (like a standard inverted index). This method of maintaining the index does not modify the negative border but still guarantees correctness of the maintained index structure: one can think of the maintained index as being based on a monotonic predicate $freq'(s) > a$ where the incrementally added records are not considered in computing $freq'$. Of course, this incrementally maintained index may degrade in performance as lots of additional records are added. To regain performance guarantees, we can periodically refresh the frequency negative border by splitting sets that go from infrequent to frequent due to the added records.

6. EXPERIMENTS

In this section, we describe our empirical evaluation of the techniques presented in this paper. The goals of our experiments are to study the following questions:

- To study the performance benefits of materializing lists in addition to the basic token-level lists.
- To compare the set-cover based covering algorithm with the hitting-set based algorithm.
- To study the lookup performance as a function of the query threshold and the number of transformations matched per query set.
- To study the index size as a function of parameter settings.

In order to undertake the above study, we compare the following lookup algorithms — prefix-filtering (PF) which is the state of the art method based on token-level inverted lists, and the negative-border (NB) index based algorithms proposed in this paper, both based on set cover (NB_SC) and hitting sets (NB_HS). We begin our discussion by describing the implementation details.

6.1 Implementation

We have prototyped the algorithms described in this paper as a part of our record matching system. This system is a stand-alone library that performs record matching in-memory. We argue that it is reasonable to assume that the data and indexes fit in memory given the increasing sizes of main memory and given that even search engine indexes built over the entire web corpus are cached in their entirety in memory [13].

We implement prefix filtering in the presence of transformations by adapting Algorithm 4.3 as described in Section 4.4.2. For each of the algorithms, we use the bipartite matching algorithm described in Section 4.5 to check if the similarity is indeed above the input threshold. In the absence of these improvements, the performance of prefix filtering suffers by orders of magnitude and hence we do not report it. For matching transformations, we use the algorithms proposed in prior work [4]. Even though our techniques can be extended to handle multi-token transformations, our current implementation only supports single-token transformations.

The transformations we use are generated in three ways. First, we use a static table that is explicitly input. This table is manually curated depending on the data set. It is applied only on the reference relation. On the query side, we programmatically generate transformations in two ways using a dictionary consisting of all tokens in the reference relation — (1) for any query token, we find all reference tokens that are within a given edit distance of the query token; we refer to this as the *edit* transformation provider, and (2) for any query token that is a single character, we find all its expansions in the reference relation; we refer to this as the *abbreviation* provider. We have knobs to control the number of transformations generated by either of these providers. For the edit provider, we use the edit distance. For the abbreviation provider, we sort all expansions returned in increasing order of the weight of the respective tokens to return the top-k; this ensures that more popular tokens are given higher preference (our weights are determined by using inverse document frequencies).

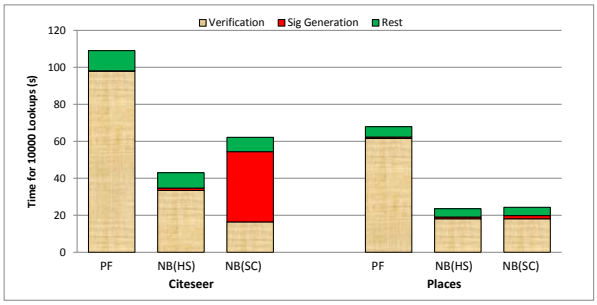
We use the apriori algorithm [3] to compute the signatures at index build time. We adapt it in a straightforward manner to incorporate transformations. As noted in Section 3, the frequency oracle is maintained by storing all a-frequent item sets (we use the apriori algorithm to build the oracle). When we measure the index size, it also includes the size of this frequency oracle. For some of the data sets containing larger sized records, we use the q-gram approach mentioned in Section 5.2 with $q = 10$ to build our index. Finally, all experiments are conducted on a dual-core workstation with an AMD Opteron processor and 8GB of main memory.

6.2 Data

Data Set	Cardinality	Average Record Length (No. of characters)
`Places`	7 million	9
`Citeseer`	0.5 million	104
`Computer`	136,000	669
`News`	50,000	5759

Figure 6: Data Sets

We consider various real-life data sets of increasing record size shown in Table 6. The table `Places` consists of names of various landmarks and organizations used in an actual online matching service. `Citeseer` refers to citations obtained from Citeseer [12] — we concatenate the authors, title, journal and year fields to obtain a single string field. The table `Computer` consists of real-life product names and attribute values, all concatenated into one single field, corresponding to computer products. Finally, even though the focus of this paper is on record matching where record sizes are typically not very large, we test our index structure on a `News` table that consists of news articles. The data characteristics are shown in



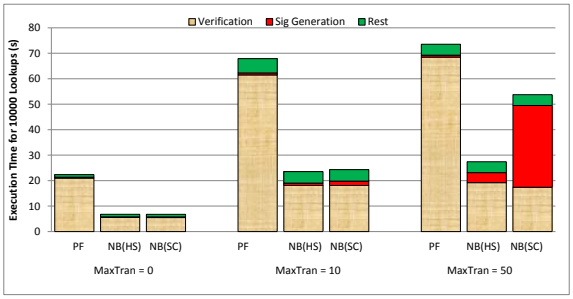


Figure 5: (a)Overall Efficiency (b)Varying No. of Transformations Per Query

Table 6. While we report our experiments on index size over all of the above data sets, we report our performance study over the `Places` and `Citeseer` data sets.

6.3 Performance

We now discuss how the techniques presented in this paper help improve the execution time of lookup queries. Owing to lack of space, we report the results of this study over the `Places` and `Citeseer` data sets. Queries are generated by taking a random subset of the input data itself. Since our focus in this paper is on asymmetric containment, we cut the query length down by choosing tokens at random from within the query. We fix the query size to be 5 tokens long in all of our experiments below. There are several parameters of interest in this set of experiments — the value of the index parameter, the query threshold and the number of transformations matched per query. We study the effect of each of these parameters on the query performance.

6.3.1 *Overall Performance*

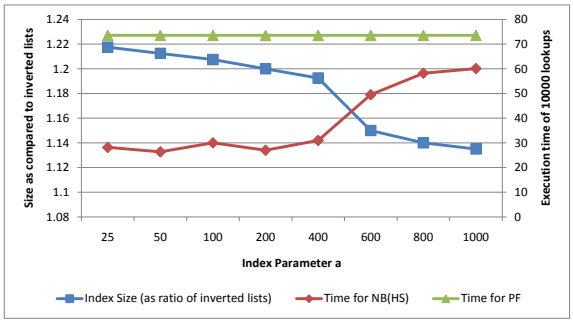
Our first experiment compares the negative border index with basic prefix filtering. As noted in Section 6.1 above, the implementation of prefix filtering in the presence of transformations itself uses our techniques both for similarity computation as well as for signature generation. Figure 5 shows the results of our experiment on the `Citeseer` and `Places` data sets. We control the number of transformations per query record to allow a maximum of 10 transformations per record. The X-axis shows the lookup algorithm clustered by data set and the Y-axis shows the execution time for 10000 queries at a threshold of 0.8 in seconds. As we can see, materializing lists in addition to the basic inverted lists yields a significant benefit in execution time. We get a factor of 3 speed up in the execution times for both the data sets. We note that this is an implementation of prefix filtering that is enhanced with the techniques presented in this paper to use bipartite matching for checking the similarity threshold and the hitting set algorithm for incorporating transformations. We note that the performance of prefix filtering in the absence of these optimizations is orders of magnitude worse and do not report these numbers. For each of the techniques, we divide the execution times into three parts — (1) verification which consists of the time to fetch the rid lists and process each of the rids (either by running the similarity computation algorithm or detecting that the rid had been fetched before), (2) signature generation time, and (3) the rest which consists of tokenization, transformation matching and iterating over the output. We first note that using our index, we can perform lookups at the rate of over 400 per second (yielding an average query time of under 3 ms) even over 7 million rows when there are string transformations based on edit distance and abbreviations in the picture; if we materialize these transformations, their number exceeds 20 million. Further, we can also see that (1) the reference comparison time is the dominant factor in the execution time, (2) the negative border index gains over prefix filtering in reducing the number of rids processed at a relatively small price of increased time in signature generation, (3) the set-cover based covering algorithm takes longer to run than the hitting set based covering algorithm; in the case of `Citeseer` data, it takes significantly longer and so (4) the benefit yielded by set cover in reduced verification time is overshadowed by the signature generation time.

6.3.2 *Space-Time Tradeoff*

The speed up yielded by the negative border index comes at the expense of an increase in index size. In order to study the space time tradeoff, we vary the index parameter a and measure how both the index size as well as the query performance vary. Figure 7(a) shows the result of the plot. The X-axis shows the index parameter and the Y-axes represent the size as a ratio over the size of standard inverted lists as well as the query execution of 10000 lookups in seconds when the number of transformations per query is bounded at 10 and the lookup threshold is 0.8. This is shown for the `Places` data but we note that similar results hold for the `Citeseer` data set as well. We can see the expected tradeoff that as the value of the parameter increases, the space consumed decreases whereas the execution time increases (we have also shown the execution time of prefix filtering for comparison). The results reported in Figure 5 are for the value of the parameter $a = 200$ for which the space consumed by our index is about 15% more than that of standard inverted lists.

6.3.3 *Effect of Transformations*

String transformations are a crucial design point in our techniques. We thus study what effect they have on the query performance. We fix the lookup threshold at 0.8 and vary the number of transformations per query record starting with no transformations at all up to a maximum of 50 transformations per record. As noted



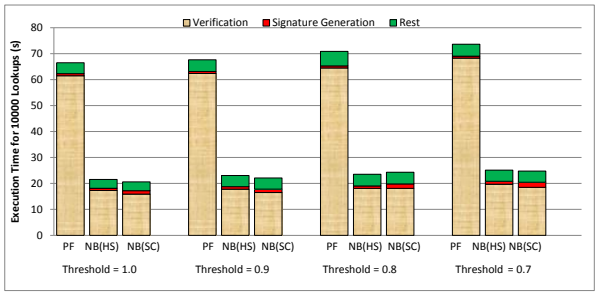


Figure 7: (a)Space-time Tradeoff (b)Varying Query Threshold (Places Data Set)

earlier, we do this by varying the knobs we have on the edit and abbreviation transformation providers. Figure 7(b) shows the result of this study for the `Places` data set. Again, the X-axis shows the lookup algorithm clustered by the number of transformations applied and the Y-axis, the execution time for 10000 queries. We can see that as expected, the execution time increases as the number of transformations increases. However, for the set cover based covering algorithm, the signature generation time increases sharply when the number of transformations per record is bounded at 50. This becomes a significant portion of the overall execution time making the set cover based algorithm substantially slower than the hitting set algorithm. Even though the hitting set algorithm has no approximation guarantees, it does only slightly worse than the set cover algorithm in terms of reference comparisons while at the same time performing signature generation efficiently.

6.3.4 *Effect of Query Threshold*

We also study the effect of the lookup threshold on the various algorithms, fixing the transformations per query at 10. Figure 7(c) reports the results of this study with the signature scheme on the X-axis clustered by the value of the threshold and the execution time on the Y-axis. As expected, the execution time increases as the lookup threshold decreases. In fact, both signature generation and reference comparisons become more expensive as the query threshold decreases as can be seen from the Figure.

6.4 Index Size

We next report the results of our experiment which studies the behavior of the index size as the index parameter a is varied. We consider relatively small values of parameter a in order to understand whether the worst case of the index size is realized in practice. Intuitively, the index size is expected to grow with the record size.

Figure 8 shows the size of the index for each of the data sets in Figure 6 for small values of the index parameter a. We report the size as a ratio of the index size to the size of standard inverted lists. This ratio is reported on the Y-axis and the X-axis shows the value of parameter a. As we can see, even though the size is bigger than that of inverted lists as expected, it is far from the worst case even for data consisting of large bodies of text such as the `News` data. This is consistent with our analysis in Section 5. We note here that for the `Computers` and `News` data, we use the q-gram frequency as discussed in Section 5.2 with $q = 10$ to cut down the index construction time. With this additional optimization, the index construction times for each of the data sets is less than half an hour, including the time spent in building the frequency oracle.

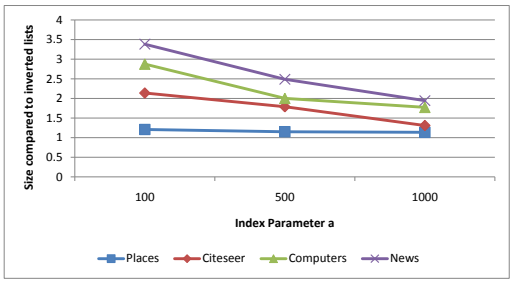


Figure 8: Index size (as a ratio of standard inverted lists).

6.5 Summary

In summary, we make the following observations from our experiments.

1. The benefit of the extra space consumed by the index is in improving the performance of error-tolerant set containment lookups significantly. We can perform over 400 lookups per second over a reference relation of 7 million rows in the presence of transformations programmatically generated from token edits and abbreviations (when materialized, the number of transformations exceeds 20 million). In contrast, if we do not enhance prefix filtering with the algorithms presented in this paper, the performance is orders of magnitude worse.
2. While the set cover lookup algorithm often yields significant improvements in running time, it has the potential to take huge amounts of time in signature generation for reasons discussed in Section 4. This can over-shadow the benefits yielded in reducing the verification time.
3. We find that the worst case of the negative border index proposed by us is not realized over various real life data sets of increasing complexity. This is the case even for small values of the input parameter a.

7. RELATED WORK

Recognizing set based similarity as a useful primitive for error-tolerant string matching, efficient similarity search techniques have

been proposed [1, 6, 10, 15, 29, 31]. All of these methods work with symmetric similarity functions, most notably Jaccard similarity. In this paper, we study Jaccard containment which is an asymmetric similarity function that is an error-tolerant version of set containment. We also handle string transformations. To the best of our knowledge, this specific combination of Jaccard containment coupled with transformations has not been addressed in prior work.

A closely related area is that of exact set containment indexing [11, 9, 24, 25, 28]. The value of constructing inverted lists corresponding to sets of keywords rather than individual keywords has been recognized earlier [9, 11] in this context. Some of this work has exploited the benefits of a power law [9] that real data sets often satisfy. Recent work [11] has used set frequencies as a basis for indexing. The main distinction from our approach is that instead of indexing minimal-infrequent token sets, any token set which can be obtained from a frequent set by adding *one* element is indexed. The number of such token sets is strictly larger than the number of minimal-infrequent sets. While this additional space consumption is beneficial for exact set containment, this is not the case under our covering based lookup framework.

Besides record matching, error-tolerant set containment is of relevance also in keyword search. For instance, it can be used to define an error-tolerant semantics for keyword search over databases. It is also applicable in performing fuzzy autocompletion [21, 23] where we have to identify matches from the underlying reference relation early on as the query string is being typed. For example, the TASTIER system [23] supports type ahead search by exploiting a trie in conjunction with inverted lists. Our work is complimentary to theirs: it will be interesting to investigate ways to use the techniques proposed in our paper as part of autocompletion.

The concept of minimal infrequent sets relates closely to frequent item sets in data mining [2]. There is a large body of work on computing frequent item sets in data mining (See [16] for a survey). Any of these algorithms can be adapted for index construction. Our query time algorithms also draw upon ideas from work done in mining frequent item sets [18, 30].

Finally, previous work [4] has reduced the problem of computing the Jaccard similarity under transformations to the problem of finding the maximum matching in a bipartite graph. But this previous reduction was for the special case of unweighted Jaccard similarity. In contrast, our technique presented in Section 4.5 is for weighted Jaccard containment.

8. CONCLUSIONS

In this paper, we studied the indexing problem for the Jaccard containment similarity function enhanced with string transformations. To the best of our knowledge, indexing this specific form of error-tolerant set containment has not been studied in prior work. We proposed a parameterized index structure that creates inverted lists for token-sets that are minimal-infrequent. We also proposed lookup algorithms that used our index structure to provide an output-sensitive guarantee for error-tolerant set containment. Our experiments over real-life data showed the benefit of our techniques. Finally, our experiments on the index size showed that across a wide variety of data sets, the size of the index remains far from its worst case, which is consistent with our analysis.

9. REFERENCES

[1] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. A new method for similarity indexing of market basket data. *SIGMOD Rec.*, 28(2):407–418, 1999.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, 2008.

[5] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1), 2009.

[6] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set similarity joins. In *VLDB*, 2006.

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

[8] A. Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences*, pages 21–29, 1997.

[9] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, 2007.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[11] S. Chaudhuri, V. Ganti, and D. Xin. Mining document collections to facilitate accurate approximate entity matching. *PVLDB*, 2(1), 2009.

[12] Citeseer. http://citeseer.ist.psu.edu/.

[13] J. Dean. Challenges in Building Large-Scale Information Retrieval Systems. WSDM'09 Keynote.

[14] A. Elmagarmid, P. G. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. 19(1), 2007.

[15] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[16] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Report on FIMI'03. *SIGKDD Explor. Newsl.*, 6(1), 2004.

[17] 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 *VLDB*, 2001.

[18] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharm. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2), 2003.

[19] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[20] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2), 2009.

[21] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, 2009.

[22] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[23] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *SIGMOD*, 2009.

[24] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, 2003.

[25] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28, 2003.

[26] S. Minton, C. Nanjo, C. A. Knoblock, M. Michalowski, and M. Michelson. A heterogeneous field matching method for record linkage. In *ICDM*, 2005.

[27] A. Nakamura and M. Kudo. What Sperner family concept class is easy to be enumerated? In *ICDM*, 2008.

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

[29] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[30] K. Satoh and T. Uno. Enumerating maximal frequent sets using irredundant dualization. In *Discovery Science*, pages 256–268, 2003.

[31] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, 2009.