CRSI: A Compact Randomized Similarity Index for Set-Valued Features

Petros Venetis
Stanford University
venetis@cs.stanford.edu

Yannis Sismanis
IBM Research – Almaden
syannis@us.ibm.com

Berthold Reinwald
IBM Research – Almaden
reinwald@us.ibm.com

ABSTRACT

We propose a similarity index for set-valued features and study algorithms for executing various set similarity queries on it. Such queries are fundamental for many application areas, including data integration and cleaning, data profiling as well as near duplicate document detection. In this paper, we focus on Jaccard similarity and present estimators that work for arbitrary similarity thresholds based on a single similarity index. We show how to build this similarity index a-priori, without knowledge about query similarity thresholds, based on recently proposed synopses for multiset operations. The index is deployed using existing disk-based inverted indexing implementations and our algorithms exploit available techniques, like skip-lists, to further optimize the query performance. The index has provably small space footprints, is orders of magnitude smaller and faster to create/incrementally maintain than exact solutions, and the algorithms provide approximate answers, with an error that is controlled by a user-specified parameter. We prove the error bounds of our algorithms analytically, and, finally, we demonstrate the performance of the algorithms and verify their accuracy experimentally.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Performance

Keywords

Set Similarity, KMV Synopsis

1. INTRODUCTION

The need to manage large repositories of data, where inconsistencies arise due to a plethora of reasons, is constantly increasing. The introduction of unstructured and semi-structured data in repositories has multiplied the effects of typographic mistakes, different formatting conventions, and transformation errors. However, consistent data is of high monetary significance for business practices, such as data profiling, data integration, and near duplicate document detection.

As a motivating example, consider the problem of profiling existing business datasources (for integration, data governance or in general data quality reasons). Such datasources typically consist of hundreds of tables, with many thousands of attributes in total. Various datasources also include XML attributes for storing sparse semi-structured attributes (like product features and categorizations) as well as document repositories for storing unstructured information (like product descriptions, manuals and customer reviews). A typical query in this setting asks for a given set-valued feature (like a relational column or an XML path) to determine all other similar (with respect to some metric [11, 20, 22]) features. The similarity is discovered through an operator that attempts to alleviate the effect of inconsistencies in the data. Depending on the exact setting and inconsistencies, many different similarity operators have been proposed in the literature [10, 23]. The main concept behind such operators is to view the operands as sets of items, and evaluate the similarity of the sets. Whenever, the similarity is high enough, the corresponding feature is flagged as interesting (or potential duplicate).

More formally, given a set-valued feature q and a similarity threshold t, the result to a similarity selection query is all other features a_i in the collection A of features with a similarity no smaller than t, i.e., the result is the set $\{a_i \in A \mid Sim(a_i, q) \geq t\}$. Two interesting variations of the above query are the join and self-join versions [1, 4, 14, 17, 21, 25]. In the join version (which is typical in data integration scenarios) given two collections A and B of features, we are interested in finding all pairs across the collections that have a similarity no smaller than a threshold. In this case the query input is just the similarity threshold t and the result to the query is the set $\{(a_i \in A, b_j \in B) \mid Sim(a_i, b_j) \geq t\}$. In the self-join version (which is typical in duplicate document discovery and data profiling) we are interested in all pairs of features with similarity no smaller than t in the same collection A.

In the above traditional application scenarios, the similarity threshold t must be known in advance (most existing methods require a full scan of the data every time t changes). However, in practice this threshold is not known beforehand and it varies according to the particular dataset properties

EDBT 2012, March 26–30, 2012, Berlin, Germany.

as well as the exact application requirements. Users have to experiment with various threshold values, which either return very few results (if the threshold is too high) or too many (if the threshold is too low). The *selectivity* version of the queries above, instead of returning the actual similar features (or pairs of features) returns the corresponding counts [16]. Such counts can be used by the user in order to determine the particular threshold that makes sense for a given application and dataset properties.

Similarity queries pose many interesting research challenges. The sheer volume of modern datasets makes it prohibitive to scan the data over and over any time the user poses a query. In the case of joins, the expected quadratic complexity — of comparing every feature conceptually with every other feature — is another challenge that must be addressed. However, in the above applications, it is not usually necessary to insist on the exact answers. Instead, determining an approximate answer suffices. This motivates the design for summary structures with small space and time requirements that can be maintained easily and support approximate results. Such structures enable a two-step approach in similarity queries. First, a fast retrieval of approximate answers, followed by a slower but accurate computation in the few cases where the user is not satisfied with the approximate answer.

In the literature, many techniques for performing exact or approximate versions of the similarity queries have been proposed. However, there are many shortcomings. Most importantly, there is no unified technique and data structure for performing all the similarity queries described above, for a wide range of similarity functions. Different structures and algorithms are required for performing, for example, self-joins and for estimating the corresponding selectivity. Other major obstacles are that most algorithms require (1) a-priori knowledge of the similarity threshold (t) and (2) a full scan of the data any time the parameters of the query (e.g., the parameter/threshold t) change.

Motivated by the above challenges, we design an approximate similarity index that can be built without knowledge of any similarity threshold and can be used for a wide family of similarity functions. The same index can be used to perform *all versions of the similarity queries* discussed above, providing robust results at minimal computational cost, storage overhead and incremental maintenance. In this paper, we focus on Jaccard similarity since it has been shown [25] that Jaccard can be used as the basic block for other more elaborate similarity functions (like cosine, overlap, edit, and dice).

The main idea is to summarize the features in the collection using a recently proposed synopsis [5] for multiset operations. Every item in the feature is hashed by one hash function (without the need of a min-wise independent family of hash functions), and the synopsis consists of the k-minimum hashed values (KMV). An inverted index on the KMV is then created using an external-memory (persistent) implementation. This index can be created *without any knowledge* of the query similarity thresholds t that the user might use later on. We show that any similarity selection, join and selectivity query with threshold t can be mapped to a variation of an at least $t \times k$ out of k query on the inverted index. This type of inverted index query is a hybrid between the typical AND/OR queries, which are supported by available inverted indexes. We provide unbiased estimators of the Jaccard distance as well as analytical error bounds that can be used to appropriately pick the size k of the synopses. We also provide algorithms that exploit techniques, like skip-lists, that most inverted indexes support to further optimize the query time performance.

Contributions Our main contributions are:

- We provide a formal framework for thinking about set similarity queries and approximations (Section 2).
- We provide estimators for the Jaccard similarity metric using the KMV synopses, error bounds on the accuracy of the estimations, and analytical formulae for appropriately selecting the synopsis size (Section 3).
- We propose a data profiling tool that uses the proposed index (Section 4).
- We propose novel algorithms for constructing appropriate similarity indexes, without knowledge of the similarity thresholds (t) that the user is interested in (Section 5).
- We develop optimization techniques that exploit modern inverted indexes characteristics to perform set similarity queries on our proposed index (Section 5).
- We demonstrate that our techniques can be directly deployed on top of freely available inverted index libraries (Section 5).
- The experimental results show that the similarity index is orders of magnitudes smaller than the input data, it can be incrementally maintained, it provides very accurate results, and it is applicable to various application scenarios (Section 6).

2. PRELIMINARIES

In this section we provide necessary background on similarity and specialized indexes for answering similarity queries.

2.1 Notation and Concepts

2.1.1 Notation

We consider a collection $\mathcal{C}$ of n set-valued features: a_1, a_2, ..., a_n, where each set a_i draws items (w_p) from a finite universe. In the following we will call each of these sets (a_i) a *feature*. A *similarity function* returns a value in $[0, 1]$ when applied on two features. We denote this similarity function as $Sim(\cdot, \cdot)$.

EXAMPLE 1. *We present one example for relational data and one for semi-structured.*

Relational *Assume we have the relational table shown below:*

ID	*Name*	*ZIP_Code*
1	Alice	12345
2	Bob	54321

Then, we have three features (one for each relational attribute):

$a_1 = \{1, 2\}$ *for attribute ID,*
$a_2 = \{\text{Alice}, \text{Bob}\}$ *for attribute Name, and*
$a_3 = \{12345, 54321\}$ *for attribute ZIP_Code.*

Semi-structured *Now, assume that we have the following two XML documents:*

<person>
<ID>1</ID>
<Name>Alice</Name>
<**homeZIP**>12345</**homeZIP**>
</person>

<person>
<ID>2</ID>
<Name>Bob</Name>
<**workZIP**>54321</**workZIP**>
<person>

In this case, we have the following features:
$a_1 = \{1, 2\}$ *for the XML path* person/ID*,*
$a_2 = \{\text{Alice}, \text{Bob}\}$ *for the XML path* person/Name*,*
$a_3 = \{12345\}$ *for the XML path* person/homeZIP*, and*
$a_4 = \{54321\}$ *for the XML path* person/workZIP*.*

2.1.2 Inverted indexes (posting lists, skip-lists)

An inverted index is a data structure that efficiently maps an item w_p to a sorted list of feature identifiers such that the corresponding features contain the item w_p [2]. The sorted list of feature identifiers is called a *posting list*. Typical posting lists implementations also contain *skip-lists* that allow their efficient traversal. More specifically, the skip-lists allow the efficient discovery of any feature identifier greater than or equal to any given identifier. Such skip-lists are used to optimize AND queries on inverted indexes, where *all* the input items in an input query must exist in the output. In contrast, OR queries return all features that contain *at least* one item from the input.

2.1.3 Inverted indexes (stored fields)

We continue and conclude our discussion on inverted indexes, by denoting as a *stored field* any stored metadata associated with a particular feature identifier. The stored fields are typically used to store useful metadata information about the indexed features. They can be thought of as a mapping from feature identifiers to any other useful metadata information. Standard inverted indexes support stored fields.

2.1.4 Similarity index

A similarity index is an index (based on a given collection $\mathcal{C}$) that can execute at least the following class of queries for any given similarity threshold t:

Similarity Selection Given a feature q return all features with similarities no less than t, i.e., $\{a_i \in \mathcal{C} \mid Sim(a_i, q) \geq t\}$.

Similarity Join Given two similarity indexes for collections $\mathcal{C}_1$ and $\mathcal{C}_2$, return all pairs of features with similarities no less than t, i.e., $\{(a_i \in \mathcal{C}_1, b_j \in \mathcal{C}_2) \mid Sim(a_i, b_j) \geq t\}$. The self-join variation returns the corresponding pairs when both features are drawn from the same collection/index $\mathcal{C}$.

Similarity Selectivity For similarity selections, joins and self-joins, instead of returning the actual result set, return the corresponding size (count) of the result set.

2.1.5 Approximations

A function $\widehat{Sim}(\cdot, \cdot)$ is called an (ε, δ)-approximation of $Sim(\cdot, \cdot)$, when the Absolute Relative Error (ARE) of the approximation is greater than ε with probability at most δ. The absolute relative error of $\widehat{Sim}(\cdot, \cdot)$ is defined as:

$$\text{ARE} = \frac{|\widehat{Sim}(\cdot, \cdot) - Sim(\cdot, \cdot)|}{Sim(\cdot, \cdot)}.$$

The values ε and δ must be small for practical applications.

An (ε, δ)-approximate similarity index can efficiently execute the above class of queries, when the $Sim(\cdot, \cdot)$ function is replaced with an (ε, δ)-approximation $\widehat{Sim}(\cdot, \cdot)$. Keep in mind that most applications try to find the most similar pairs of features; the thresholds t selected are found empirically and it is not fundamental that the similarity of two features is exactly more than t. For example, in near duplicate document detection, it is important that the similarity of a pair of documents is high, *but* it is not too important to note that it *must* be higher than t, as long as it is close or more than t. Thus, solving the (ε, δ)-approximation problem is usually sufficient for practical purposes.

2.1.6 Similarity Functions

In the rest of the paper, $Sim(\cdot, \cdot)$ denotes the Jaccard similarity $J(\cdot, \cdot)$ by default, unless otherwise stated. Thus, for two sets a_i and a_j we have $Sim(a_i, a_j) = \frac{|a_i \cap a_j|}{|a_i \cup a_j|}$. We point out that we focus on Jaccard similarity, because it has been shown [25] that $J(\cdot, \cdot)$ can be used as the basic block for other similarity functions (like cosine, overlap, edit, and dice).

2.2 Straight-Forward Approaches

In this section, we describe some straight-forward ideas for using inverted indexes in order to perform exact similarity queries and identify the pitfalls.

Let's assume we have the inverted index of a collection of features. The inverted index maps any given distinct item w_p that appears inside a feature of the collection to the corresponding posting list.

For simplicity, let's start with the case of a similarity selection query. More precisely, let's assume we are given a feature $q = \{w_1, w_2, w_3\}$, and we are interested in finding all other similar features in the collection with a threshold $t = 2/3$ using the Jaccard similarity function. It is easy to see that the only possible features (of size at most 3) that could match our query (other than q itself) are $\{w_1, w_2\}$, $\{w_1, w_3\}$ and $\{w_2, w_3\}$. In other words we are looking for features that contain at least $t \times |q| = 2$ items from q.

However, typical inverted indexes only support AND/OR posting list processing. In other words, inverted indexes can either find features that contain *at least* one item in q or *all* the items in q. We must point out that especially for the AND posting list processing, typical inverted indexes provide skip-lists that can speed up the processing tremendously [2].

We denote the posting list processing where we need at least n items out of m as n/m-posting list processing. In our example above we require $n = 2$ out of $m = 3$ items. This kind of processing is a hybrid between AND and OR processing. In the literature it is an instance of Weighted-AND (WAND) queries [6].

The previous example suggests that (at least for Jaccard similarity) inverted indexes are a very natural starting point for similarity queries. Another appealing point is that various similarity queries (other than selections) can be handled naturally as well. For example, a similarity join can be executed in an "indexed nested-loop join" fashion: For each feature, we probe the index to efficiently find other similar features. The same approach can be used of course to handle self-joins. Selectivity queries, that only return the number of results, are executed as their counterpart full queries, that

return the actual results, with a small tweak that only counts the results instead of producing them as output.

However, current solutions that use inverted indexes have various undesirable properties:

1. The full inverted index has a size that is in the same order of magnitude as the original data (or in the best case around $(1-t)$ times the data size [25]), even with the current optimal techniques. This means that it may be impractical to construct such an index for some business applications.
2. Existing inverted index implementations can only handle a small number of input items, since they can only open a relatively small number of posting lists efficiently. This means that only features with a very small cardinality can be handled efficiently (even if we had the full inverted index).
3. An optimized n/m-posting list processing is not part of available inverted index implementations.

In the following, we describe how we address the above pitfalls. The main idea is to summarize the features such that the size of the inverted index is orders of magnitude smaller than the actual data. The summarization however, allows for provably efficient approximation of the actual similarity between features for arbitrary query-time similarity thresholds and many different similarity functions. It also reduces the amount of posting lists that must be processed, during query time, so that existing implementations can handle it (more) efficiently. Finally, we propose various techniques for executing n/m-posting list processing that take the skip-lists into account; these techniques are an extension of the WAND queries [6], specifically customized for our index structure. In particular, we have to deal with n/m-posting list processing in the scenario where n and m depend on the feature we are currently examining and the features that we find in the posting lists. This means that n and m are not constant through the processing, even if all other parameters (like t) are constant (in contrast to [6]).

3. ESTIMATES FOR SET OPERATIONS

In this section, we describe how to obtain robust estimates for distinct counts and various set operations (union, intersection and Jaccard distances), based (only) on a constant size synopsis.

In [5] a simple and yet powerful summarization technique (KMV), for multiset operations was studied. KMV stands for **k-M**inimum hash **V**alues:

DEFINITION 1. *Assume that we have a finite universe of items $\mathcal{U}$, a feature $a = \{w_1, w_2, \ldots, w_m\} \subseteq \mathcal{U}$ and a hash function $h : \mathcal{U} \to \{0, \ldots, M\}$. The* KMV synopsis *of a under hash function h is the* set *(not multiset) of the k minimum hash values to which the items of a map to. Equivalently, it is the set H consisting of the k minimum hash values of the* set $\{h(w_i) : i \in \{1, 2, \ldots, m\}\}$.

The following example makes the KMV synopses clear.

EXAMPLE 2. *Assume that we have the feature:*

$$a = \{\text{'aa'}, \text{'bb'}, \text{'cc'}, \text{'dd'}, \text{'ee'}, \text{'aa'}\}.$$

Now, assume that our hash function gives the following values: $h(\text{'aa'}) = 1$, $h(\text{'bb'}) = 3$, $h(\text{'cc'}) = 7$, $h(\text{'dd'}) = 5$, $h(\text{'ee'}) = 1$. If we want the KMV *synopsis of this multiset (features are allowed to be multisets[1]) with parameter $k = 2$, i.e., by keeping the two minimum hash values, we have the following* KMV *synopsis for a: $\{1, 3\}$. Observe that the* KMV *synopsis is a* set *even though the value* 1 *appears three times in the hash values (two for the items 'aa' and one for item 'ee').*

We make use of two variations of the KMV synopsis, depending on the multiset we are going to use it on.

DEFINITION 2. *We call the* KMV *synopsis* incomplete*, when the corresponding feature a has more than k distinct items. Equivalently, since we assume that there are no collisions[2] from our hash function, we call the* KMV *synopsis incomplete, when the set $\{h(w_i) : i \in \{1, 2, \ldots, m\}\}$ has more than k items.*

DEFINITION 3. *We call the* KMV *synopsis* complete*, when the corresponding feature a has at most k distinct items. Equivalently, we call the* KMV *synopsis complete, when the set $\{h(w_i) : i \in \{1, 2, \ldots, m\}\}$ has at most k items.*

The intuition behind the names incomplete and complete is the following. The KMV synopsis is called incomplete when it does not capture all the information of feature a. On the contrary, it is called complete when it captures all the information of feature a. We need the separation between complete and incomplete KMV synopses of features in order to correctly estimate distinct counts and set operations between them.

Strict and probabilistic bounds for distinct counts, unions, and intersections for incomplete synopsis were derived in [5]. We complement the analysis for Jaccard similarities, with the following:

LEMMA 1. *An unbiased estimator of the Jaccard similarity of features a_i and a_j with* KMV *synopses* $\text{KMV}(a_i)$ *and* $\text{KMV}(a_j)$ *is:*

- *If and only if at least one of a_i and a_j has an incomplete* KMV *synopsis:*

$$\hat{J}(a_i, a_j) = \frac{|\text{KMV}(a_i) \oplus \text{KMV}(a_j)|}{k}$$

  $\text{KMV}(a_i) \oplus \text{KMV}(a_j)$ *is the set of hash values that belong in* $\text{KMV}(a_i) \cap \text{KMV}(a_j)$ and *in the k smallest hash values of the set* $\text{KMV}(a_i) \cup \text{KMV}(a_j)$.
- *If and only if both a_i and a_j have complete* KMV *synopses:*

$$\hat{J}(a_i, a_j) = \frac{|\text{KMV}(a_i) \cap \text{KMV}(a_j)|}{|\text{KMV}(a_i) \cup \text{KMV}(a_j)|}.$$

The case where both synopses are complete, happens only when size k is large enough to accommodate all items in the

[1]The Jaccard similarity though is defined in terms of distinct items that belong to the intersection over the distinct number of items that belong to the union.

[2]Assume that the number of distinct values that a feature a contains is $|a|$. If $M = \Theta\left(|a|^2\right)$ then the probability of a collision occurring is exponentially small [19]. This means that in current implementations the probability of two items colliding when one is using a uniform hash function is 2^{-128}, thus even for very large features one can safely ignore collisions.

feature. In that case only, we use directly the sizes of the synopses (that are the same as the sizes of the features if we ignore collisions) to get a tight estimator of the Jaccard similarity: If we ignore the hash function collisions in this case, the estimator returns the exact value of the Jaccard similarity.

In the most common case however, where the synopsis size k is much smaller than the cardinality of the corresponding feature, the following lemma connects the size k of the KMV synopses with the error bounds of the Jaccard estimation. Most accurately, it connects the probabilistic error bound to the actual Jaccard similarity we are estimating. Since in practice we do not know the actual Jaccard similarity, we also provide bounds with respect to any arbitrary threshold t of the Jaccard similarity we are interested in. We will only examine error bounds when both of the features are incomplete, because if (at least) one of the features is complete, we get much smaller errors ε for the same δ.

LEMMA 2. *The probability δ that the absolute relative error (ARE) of $\hat{J}$ is greater than ε is bounded by:*

$$\delta \leq \frac{(1-J)}{kJ\varepsilon^2} \leq \frac{(1-t)}{kt\varepsilon^2}.$$

PROOF. If the number of common hash values between the k minimum hash values in $\mathrm{KMV}(a_i) \oplus \mathrm{KMV}(a_j)$ for the incomplete synopses for features a_i and a_j is $K_\cap$, then we can show [5] that

$$\hat{J} = \frac{K_\cap}{k}$$

is an unbiased estimator for the Jaccard distance J. This follows directly from Equation (13) in [5] where $\mathcal{D}_\cup$ and $\mathcal{D}_\cap$ is the size of $a_i \cup a_j$ and $a_i \cap a_j$ respectively:

$$E[\hat{J}] = E[\frac{K_\cap}{k}] = \frac{\mathcal{D}_\cap}{\mathcal{D}_\cup} = J$$

which demonstrates that the expected value of $\hat{J}$ is indeed J and therefore $\hat{J}$ is unbiased. From Equation (14) in [5] we can compute the variance of $\hat{J}$ as follows:

$$\sigma^2 = \mathrm{Var}[\hat{J}] = \mathrm{Var}[\frac{K_\cap}{k}] = \frac{J(1-J)(\mathcal{D}_\cup - k)}{k(\mathcal{D}_\cup - 1)} \leq \frac{J(1-J)}{k}.$$

By using Chebyshev's bounds we can show that:

$$\Pr[|\frac{\hat{J}-J}{J}| \geq \varepsilon] \leq \frac{\sigma^2}{J^2\varepsilon^2} \leq \frac{(1-J)}{kJ\varepsilon^2}.$$

In other words the probability δ that the absolute relative error (ARE) of $\hat{J}$ is greater than ε is given by:

$$\delta \leq \frac{(1-J)}{kJ\varepsilon^2} \leq \frac{(1-t)}{kt\varepsilon^2}.$$

□

Figure 1 depicts the error bounds for $\hat{J}$ for various values of k. For example, for synopsis size $k = 128$ and $t = 0.8$, the ARE of the Jaccard estimator $\hat{J}$ is at most $\varepsilon \approx 12\%$ with probability *at least* 90%. When the synopsis size scales up to $k = 1{,}024$ then the ARE scales down $\varepsilon \approx 5\%$ with the same probability. Our analysis above provides the practitioner with enough information to pick the synopsis size, given the particular constraints of the application.

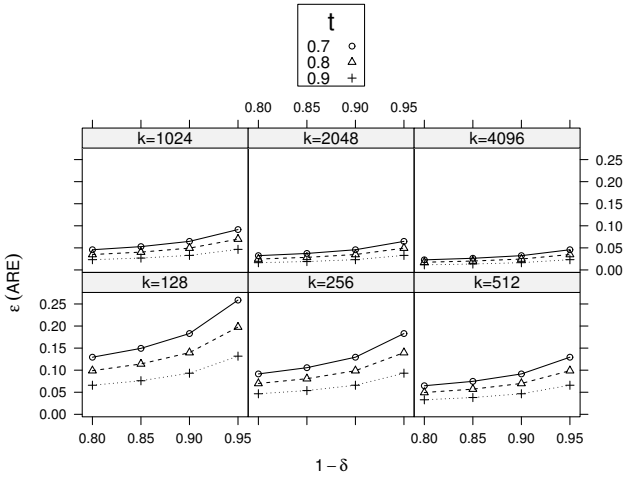


Figure 1: Analytical Error Bounds of $\hat{J}$.

We emphasize that in practice even very small synopsis sizes $k \approx 100$ are sufficient to get very accurate Jaccard estimates, since Chebyshev's bounds can be very loose (only the mean and the variance of the distribution is taken into account). This is demonstrated in our experiments, where even for small k the error was much smaller than the error bounds depicted in Figure 1. If one wants to be strict in the selection of k, then after deciding the appropriate ε, δ and the minimum threshold $t_{\min}$ for which similarity queries will be asked, he should choose $k \geq (1 - t_{\min})/(t_{\min} \cdot \varepsilon^2 \cdot \delta)$.

4. HIGH LEVEL OVERVIEW

We implemented a data profiling tool, that allows for many different similarity functions to be estimated using one index without a-priori deciding the similarity threshold. We first ingest the data, either from relational datasets or from XML/document repositories. For each feature of our dataset we compute its KMV synopsis and invert it using a freely-available inverted indexer. The output of this procedure is an inverted index, which we call Compact Randomized Similarity Index *CRSI*. Our system queries the similarity index and finds the similar features of our original dataset (self-join), with accuracy that we can control.

In the following, we examine algorithms and techniques in order to create the similarity index and also to query the index to get the similar features. For clarity and without loss of generality, we focus on Jaccard similarities; we can also handle other similarity functions (e.g., overlap similarity, Hamming distance) and mine foreign-key relations or set containments using our framework.

5. ALGORITHMS

Assume that we have a collection $\mathcal{C}$ of features: $\mathcal{C} = \{a_1, a_2, \ldots, a_n\}$. Each feature is a multiset of items, i.e., $a_j = \{w_{j1}, w_{j2}, \ldots, w_{jl_j}\}$. Features in our collection could be structured attributes (e.g., relational columns), semi-structured attributes (e.g., XML paths), or even completely unstructured attributes (e.g., text documents).

When the features are relational columns, we typically use the contents of a column directly as items of the corresponding feature. When the features are text we typically treat

each document as a set of keywords (maybe after stemming and removal of the stop-words). Other popular approaches include using q-grams for doing approximate string matching.

For semi-structured XML documents, we assign a feature for every distinct path in the documents. The features in this case correspond to the distinct XML paths. Then, for any XML path (feature), we gather all items that appear in it (possibly in different XML documents) and we treat it as a set. If we are given some XML schema, then we can further decide if we want to perform stemming/removal of stop-words for a text XML-path, or we leave it as is (for example for numerical values).

As we will see later on, our techniques can easily handle large amounts of XML data efficiently, even though the data can be sparse. Other methods that construct inverted indexes with size comparable to that of the initial size of the dataset (or comparable to that) cannot handle XML data with just one pass through the data (see Section 6.2).

As we described in Section 4, our algorithms consist of two parts: (1.) building the *CRSI* for the collection, and (2.) using the *CRSI* to answer similarity based queries. In the following we examine with detail each of the two steps.

5.1 Building the *CRSI*

Our algorithm's first step is building an index which we call Compact Randomized Similarity Index (*CRSI*). This inverted index is created as a normal inverted index, containing the posting lists from the KMV hash values to the appropriate features. More specifically, the input of the algorithm consists of: (1) a collection of features a_1, a_2, ..., a_n, (2) integer k that determines the (maximum) size of the KMV synopsis of a feature, and (3) hash function $h(\cdot)$ that takes a number of bytes as input and returns a value in $\{0, \ldots, M\}$. The output of the algorithm is an inverted index from hash values of items of the features to the KMV synopses/features; also, the KMV synopsis of each feature and a flag indicating whether it is complete or incomplete is kept as a stored field in the inverted index.

The pseudocode for the detailed algorithm is Algorithm 1. In a nutshell the algorithm starts with an empty similarity index, it computes the KMV synopsis for every feature by hashing and keeping its k minimum values, creates posting lists for the KMV hash values that contain feature-ids, and finally stores the KMV for every feature and whether it is complete or incomplete. The fact that we store a flag that tells us if the KMV synopsis is complete or incomplete allows us to use the best similarity estimator during query time and to avoid going back to the base data anytime we need the KMV synopsis of any feature. In that sense, *CRSI* we build is self-contained for a size penalty. Essentially, we are not only indexing the KMV synopses but also storing them, roughly doubling the storage requirements. However, as we show in the experiments, the similarity index is quite small, even after doubling the storage requirements.

Finally, we must emphasize that the only parameter that affects the creation of the *CRSI* is the parameter k, i.e., the synopsis size. We do *not* need to know beforehand the exact threshold t that we will use.

The algorithm runs in linear time on the number of items that all the sets contain. More precisely, the complexity of the algorithm is $O\left(k \log(k) \sum_{i=1}^{n} |a_i|\right)$. Since typical values for k are no more than 1,024, i.e., bounded by a constant, the complexity is $O\left(\sum_{i=1}^{n} |a_i|\right)$. In any case, only one pass through the data is required.

Algorithm 1: Building the *CRSI*

```
for i = 1 to n do
    // find KMV for feature a_i
    L = ∅;
    foreach v ∈ a_i do
        if v ∉ L then
            if |L| < k then
                L ← L ∪ {v};
            else
                if v < max_{x∈L} {x} then
                    L ← L ∪ {v};
                    currentMax ← max_{x∈L} {x};
                    L ← L \ {currentMax};
    // hash values to features
    foreach v ∈ L do
        posting_list(v).add(a_i);
    // stored information
    store(a_i, L);
    store(a_i, complete or incomplete);
```

In this algorithm, we have assumed that we have direct access to the items of each feature. If this is not true (like for XML documents when features are all contents for particular XPaths), then there are at least two ways our algorithm can work: (1) first do a pass over the data and gather all the items for each feature, or (2) keep for each feature (XPath) its KMV synopsis in memory and update it as you see more items for that feature. Solution 1 is better when we have a lot of features or limited memory for the amount of data we have to process. In our experiments we used Solution 2.

5.2 Similarity Queries using the *CRSI*

Let us now see how we can use the *CRSI* we have built in Section 5.1, in order to answer various similarity queries. A similarity selection query using the *CRSI* is the most basic operation. Similarity joins and self-joins are executed in an indexed nested loop fashion, i.e., by posing one similarity selection query for every feature in the outer loop of the join. As we discussed in Section 2.2, one can categorize the corresponding techniques, we present here, as n/m-posting list processing.

We study three main techniques: (1) posing OR queries to the *CRSI*, (2) looking at the frontier of the posting lists, and (3) doing pivot queries (inspired by [6]) and exploiting the skipping lists. Finally, we explore a small optimization for the last method (pivot queries).

The input for the algorithms described in this section is: (1) the *CRSI*, which contains the KMV synopses of all features and information regarding to whether they are complete or incomplete, (2) an input feature q, and (3) the similarity threshold t. The output is the set of features, along with the corresponding $\hat{J}$, that have $\hat{J}$ higher than the threshold t: $\{(a_j, \hat{J}(q, a_j)) \mid \hat{J}(q, a_j) \geq t\}$.

In the rare case where one needs exact results, there are two things that can be done. (1) After we get the results of our algorithms back, we can evaluate the Jaccard similarity

of the similar features, i.e., treat the returned set as a *candidate* set. (2) Based on the error bounds we have over the estimations for the Jaccard similarity, we make sure that we use a different value $t' < t$ small enough so that we do not miss any pairs from the candidate set (with a probability that we can determine). Usually, this is not required, since the threshold t that somebody sets is intuitive. Real applications just need to have an idea of which features are similar with each other and give an arbitrary (and always high) value of t. There are very few applications where somebody is interested in finding pairs of features that have a similarity that are similar *exactly* 0.823 or more for example.

5.2.1 Posing OR Queries to the *CRSI*

From the estimators we saw in Section 3 we can conclude that if two of our features are similar, then they must share at least one hash value in their KMV synopses. Otherwise the (unbiased) estimator returns Jaccard similarity 0.

For the input feature q this algorithm finds all other features that share at least one hash value with q in their KMV, by issuing OR query on *CRSI*. Then, for each of those candidates $a_j \in L$ it estimates the Jaccard similarity $\hat{J}(q, a_j)$. If this estimated Jaccard similarity is greater or equal to the given threshold the algorithm outputs the pair of features and their estimated similarity.

5.2.2 Frontier of Posting Lists

Inverted indexes are not constructed to accept OR queries of many keywords. To avoid posing such queries on *CRSI* we develop the algorithm described in this section.

Algorithm 2: Similarity Selection using the frontier of the posting list

```
foreach v ∈ KMV(q) do
    // posting list for v points to its first
    // feature
    posting_list(v).start();
// F is a min-heap priority queue (according to
// IDs) of features that posting lists currently
// point to (frontier)
F ← ⋃_{v∈KMV(q)} (posting_list(v).getFeature());
foreach a_j ∈ F do
    PLs_j ← posting lists that point to a_j;
while F ≠ ∅ do
    a_l ← F.getMinFeature();
    // PLs_l is used for the following estimation
    if Ĵ(q, a_l) ≥ t then
        output (a_l, Ĵ(q, a_l));
    F.remove(a_l);
    foreach PL ∈ PLs_l do
        if not PL.empty() then
            PL.moveToNext();
            a_g ← PL.getNext();
            if a_g ∉ F then
                F.insert(a_g);
            PLs_g.insert(PL);
    delete PLs_l;
```

This algorithm handles the posting lists more effectively than the OR queries algorithm (Section 5.2.1). The key idea of this algorithm is the efficient usage of the *frontier of the posting lists*. The detailed pseudocode for this algorithm can be found in Algorithm 2. The frontier of the posting lists at each point is the set of all features that are currently pointed by at least one posting list. For example, if $\text{KMV}(q) = \{1, 2\}$ and we have the following posting lists: $1 \rightarrow \langle f_1, f_3 \rangle$ and $2 \rightarrow \langle f_2, f_3 \rangle$, then starting at the **foreach** loop we start processing the posting lists, the frontier would be $\mathcal{F} = \{f_1, f_2\}$, since these are the features that are pointed by our two posting lists. If we advance the posting list of hash value 1 to the next element (i.e., f_3) then the updated $\mathcal{F}$ will be $\mathcal{F} = \{f_2, f_3\}$.

Here is how the algorithm works: It finds the hash values in $\text{KMV}(q)$ (from the stored field of *CRSI*). Assume that a_l is the feature in $\mathcal{F}$ with the lowest index (l is as small as possible). Then, we are guaranteed that it will not appear in any other position of the posting lists we examine as we proceed. This is true because the posting lists keep the elements sorted (in our case by feature ID). Thus, we know exactly how many overlapping items q and a_l have in their KMV synopses (it is precisely $|PLs_l|$, where PLs_l are all the posting lists that point to the feature a_l currently).

Knowing the size of $\text{KMV}(q) \cap \text{KMV}(a_l) = PLs_l$, we have enough information about q and a_l and we can estimate their similarity and output a_l if it has a similarity higher than the threshold t. After this step, we can advance each posting list PL that was pointing to a_l to its next feature (noted as a_g in the pseudocode), update $\mathcal{F}$ and PLs_g and repeat the procedure until $\mathcal{F}$ is empty (all posting lists have been exhausted).

5.2.3 Pivot Queries to *CRSI*

One can observe that we do not always need to check each feature in the frontier $\mathcal{F}$ (that was described in Section 5.2.2) to see how similar it is to a. Sometimes, a subset of $\mathcal{F}$ contains features that are guaranteed to have a similarity with a that is less than t.

For example, suppose that $k = 128$, $t = 0.8$ and that we are examining an incomplete KMV synopsis for a; then for any feature f we have $\hat{J}(q, f) = \frac{|\text{KMV}(q) \oplus \text{KMV}(f)|}{k}$ (Section 3). Thus, we need a feature f to have $|\text{KMV}(f) \cap \text{KMV}(q)| \geq |\text{KMV}(f) \oplus \text{KMV}(q)| \geq 103$ in order to have $\hat{J}(f, q) \geq 0.8$. If at some point for example we have $\mathcal{F} = \{f_4, f_7, f_8, f_{10}, f_{11}\}$ with $|PLs_4| = 10$, $|PLs_7| = 10$, $|PLs_8| = 10$, $|PLs_{10}| = 75$ and $|PLs_{11}| = 23$, then we are guaranteed that none of the features with IDs less than the ID of f_{10} can have a $\hat{J} \geq 0.8$. This is true because there are $10 + 10 + 10 = 30$ posting lists pointing to feature IDs less than the ID of f_{10} (remember that posting lists are sorted on feature ID). Even if they all pointed to the same feature, then $\hat{J}$ would be less than 0.8. Thus, we can skip all the posting lists in $PLs_4 \cup PLs_7 \cup PLs_8$ to feature ID 10.

Taking this observation into account, we developed Algorithm 3. With this technique, we use skip lists very frequently (see .skipTo() method in our pseudocode). With the way skipping lists are typically constructed, this leads to great efficiency differences with the previous approaches.

The algorithm to a large extent works as the frontier algorithm. The main difference is observed when we know that the first feature in $\mathcal{F}$ (a_l) has $\hat{J}(a_l, a) < t$. In this case, a set $\mathcal{F}' \subseteq \mathcal{F}$ of all features that have no chance in getting a

Algorithm 3: Pivot queries to the *CRSI*

```
foreach v ∈ KMV(q) do
    posting_list(v).start();
F ← ⋃_{v∈KMV(a_i)} (posting_list(v).getFeature());
foreach a_j ∈ F do
    PLs_j ← posting lists that point to a_j;
while F ≠ ∅ do
    a_l ← F.getMinFeature();
    if Ĵ(a_l, q) ≥ t then
        output (a_l, Ĵ(a_l, q));
        F.remove(a_l);
        foreach PL ∈ PLs_l do
            if not PL.empty() then
                PL.moveToNext();
                a_g ← PL.getNext();
                if a_g ∉ F then
                    F.insert(a_g);
                PLs_g.insert(PL);
        delete PLs_l;
    else
        assume F' ⊆ F is a subset from the beginning of
        F such that if all posting lists pointing to F'
        were pointing to one feature f, then Ĵ(f, q) < t;
        a_l' ← (F \ F').getMinFeature();
        foreach a_j ∈ F' do
            F.remove(a_j);
            foreach PL ∈ PLs_j do
                if not PL.empty() then
                    PL.skipTo(l');
                    a_g ← PL.getNext();
                    if a_g ∉ F then
                        F.insert(a_g);
                    PLs_g.insert(PL);
            delete PLs_j;
```

similarity greater or equal to t is determined. All the posting lists that point to features in $\mathcal{F}'$ are skipped to the first potentially similar feature in $\mathcal{F} \setminus \mathcal{F}'$.

There is a number of variations of this algorithm based on the selection we will do on $\mathcal{F}'$. Experimentally, we found that when we skip the posting lists that currently point to exactly the feature with the smallest ID in $\mathcal{F}$ is the most efficient.

5.2.4 Optimistic Pruning

Instead of considering all features, we can introduce an "optimistic" preprocessing step. For the feature a we only consider a few posting lists for the first $k' = 8$ hash values (instead of all k) in KMV(q). We find the features that have a similarity with the current feature q more than $t' = \frac{t}{3}$ based on the estimates for these few posting lists. Then, we open all k posting lists in KMV(q), but only take into account our candidates from the preprocessing step.

This pruning *rarely* misses any similar pairs of features (hence the name optimistic). Actually, the probability of missing a pair that has a similarity that is greater than t is ∼3% (for $k' = 8$ and $t' = \frac{t}{3}$, based on the Chebychev inequality we have used for the proof of Lemma 2). We emphasize that during our experiments, we did not miss *any* pair of similar features. This is mainly because of the loose bounds that we have used for the proof of the Lemma 2.

6. EXPERIMENTAL EVALUATION

We implemented the algorithms described in Section 5 and we applied our prototype to various real-world datasets. We empirically show the time and space requirements for computing and storing the *CRSI*. We compare against the time and space requirements of another algorithm (`ppjoin++`). Then we demonstrate the incremental maintenance behavior of our approach. We evaluate its query performance (1) for similarity selection queries and (2) for self-joins. The latter is implemented as an indexed nested loop join over the index and stresses *CRSI* performance. Finally we analyze the accuracy of the returned results and compare with the probabilistic error bounds of our results we derived in Figure 1.

6.1 Experimental Setup

Implementation We implemented the algorithms described in Section 5 as Java UDFs and we used IBM DB2 v9.5 to store our datasets. For an inverted index, we used the Lucene[3] text indexer, which is available under the Apache License. All experiments were performed on a PC with an Intel CPU running at 2.66GHz with 2GB of RAM and 120GB of disk space. The operating system was Windows XP.

Algorithms We will refer to the algorithm that uses plain OR queries as `OR`, to the algorithm where we handle the frontier of posting lists as `PL` and to the algorithm where we do the pivot queries as `PQ`. For the `PQ` algorithm, we observed that the version that does the skips in the first group of posting lists ($|\mathcal{F}'| = 1$) gives the best response times and this is the we used this variation for our experiments. Also, we will refer to the pivot queries with optimistic pruning algorithm as `MA`; we used $k' = 8$ and $t' = \frac{t}{3}$ for this algorithm, as described in Section 5.2.4.

Datasets We evaluated our methods using three different real-world datasets. The `OPIC` dataset contains product information for a large computer company. The `RDW` dataset was obtained from the data warehouse of a large financial company. Both `OPIC` and `RDW` are relational datasets. These datasets were actually used for a data profiling/integration application and were very good candidates for evaluating all aspect of the data profiling tool we built around *CRSI*. Finally, `INFOBOXES` is a real-world, freely-available dataset consisting of Wikipedia XML documents. Wikipedia[4] contains semi-structured information for particular types of information (like actors, sport teams), which are called infoboxes. Many different XML paths in the `INFOBOXES` dataset contain very similar information (like addresses, times, locations, etc.) under very different XML paths. The main reason (and motivation for the *CRSI*) is that different users and groups of users are responsible for maintaining one infobox, without however strict coordination across infoboxes. Our *CRSI* applied in this scenario, for example can help further understand and organize the content of Wikipedia (for example, in our experiments we identified that the XML paths movie_stars/names and actors/names are "duplicates"/very

[3] `http://lucene.apache.org/`
[4] `http://www.wikipedia.org/`

similar and thus should be maintained appropriately). We extracted all Wikipedia information and created the corresponding XML repository, on which we run our algorithms to find similar, in terms of content, XML paths (see Example 1). More detailed information about our datasets can be found in Tables 1 and 2.

Dataset	# of Columns	# of Tuples
OPIC	1,802	27,757,807
RDW	504	2,661,506

Table 1: Relational Datasets Characteristics.

Dataset	# of XML paths	# of XML documents
INFOBOXES	174,527	532,338

Table 2: XML Dataset Characteristics.

In order to test the incremental behavior of the *CRSI* in certain experiments we used an increasing percentage of the features. We also experimented with an exact similarity self-join algorithm (the ppjoin++ proposed in [25], which is a state-of-the-art algorithm for similarity self-joins) algorithm to see its storage and time requirements on our datasets. This algorithm requires knowledge a-priori of the similarity threshold t. We must emphasize, that ppjoin++ works in an interleaved-way, i.e., performing the similarity-join as it creates posting-lists in memory. This is contrast to *CRSI* which is targeted towards persistence and repeated usage of the index. Finally, we make the observation that ppjoin++ requires the items in each feature to be sorted, which resulted in considerable time especially for features with very large cardinalities (like primary/foreign keys in our datasets).

6.2 *CRSI* Creation Time

We now focus on the time required to create a similarity index. We experiment with various values of k and all our datasets and examine the required time for the index to be created. The results we got can be seen in Figure 2.

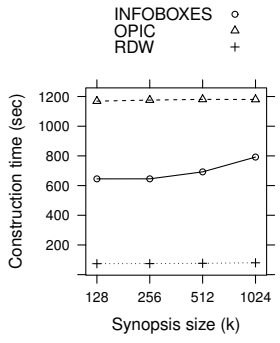


Figure 2: *CRSI* creation time.

The time required to create the similarity index is independent of the value of k for the relational datasets. For the INFOBOXES, the value k plays a key role in the time required, since it makes the preprocessing of the XML data (collecting KMV's for all features simultaneously) slightly worse as it grows.

For ppjoin++ the results are depicted in Table 3. Although, the *CRSI* does not need a-priori knowledge of any similarity thresholds, for ppjoin++ it is required. We used $t = 0.8$ for these experiments. For the XML data we expected the memory requirements to be about as large as the input data in order to find all the data for all the XML paths initially and indeed we run out of memory for the INFOBOXES dataset. Also, the size of the index is a large percentage (approximately $1 - t$) of the input data size.

Dataset	Time (sec)	% of dataset processed
OPIC	∼120,000	∼25%
RDW	∼10,000	∼20%
INFOBOXES	∼10,800	run out of memory

Table 3: ppjoin++ processing.

6.3 *CRSI* Space Requirements

We first examine the size of our similarity index for our datasets. The results can be seen in Table 4.

Dataset	Size of Dataset (MB)	k	Size of index (MB)
OPIC	5,409	128	1.27
		256	2.05
		512	3.3
		1,024	5.2
RDW	665	128	0.774
		256	1.36
		512	2.53
		1,024	4.53
INFOBOXES	668	128	48.3
		256	58.3
		512	69.2
		1,024	79.9

Table 4: The size of the similarity index for all our datasets and various values of k.

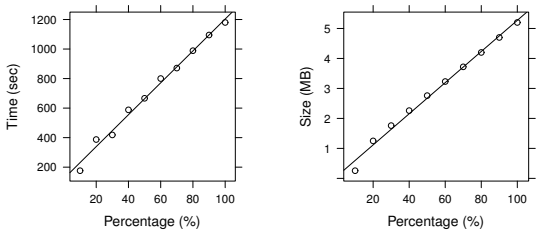
We see that the larger k is the more space we need, as expected. Generally, the size depends on the number of features we are indexing. Thus, since RDW has the fewer features (relational columns) it is the smallest index; INFOBOXES has the largest number of features (XML paths) and thus has the largest index.

As we mentioned in Section 6.2 the ppjoin++ method requires that all features are sorted under some universal ordering, which was expensive for features like primary/foreign keys. Just for comparison reasons, we note that ppjoin++ required 530MB and 47MB for the OPIC and RDW datasets respectively at the time we stopped the experiments. We have excluded the INFOBOXES dataset, since the ppjoin++ algorithm run out of memory during the preprocessing of this dataset.

Even though, we did not run the ppjoin++ experiments till the end, it is obvious that in terms of size (and time as we observed earlier) the similarity index we create is much smaller (and faster). Our index is as small as 0.1% of our original dataset in some cases, while the ppjoin++ algorithm requires 10% of the size of the original dataset just for one fourth of the dataset processed. This of course is expected since our approach advocates approximation in the results, while ppjoin++ produces exact results. It demonstrates, however that it is quite impractical to require exact results, especially when the features have very large cardinalities.

6.4 Incremental Maintenance of *CRSI*

The next thing we examined is the scaling of the *CRSI* we construct. We wanted to see two things, compared to the size of the input dataset: First, how fast is the similarity index constructed and second, how large is the similarity index. For this experiment, we selected the largest dataset in our disposal: `OPIC`.



(a) Time scaling (b) Space scaling

Figure 3: Incremental maintenance with respect to # of features (`OPIC` datasets).

The scaling in terms of time can be seen in Figure 3(a) and, the scaling in terms of size in Figure 3(b), where the x-axis is the percentage of features included in the *CRSI*. There is a very clear (almost linear) relationship between both the time and the size vs. the size of our input dataset. That is, the larger the dataset we have, the more time and space we need to construct the index.

However, we must emphasize that the inverted index creation is *incremental*: We are adding more and more features (incrementally) over the same inverted index. This demonstrates, that *CRSI* is applicable to scenarios with continuously changing/incoming data.

6.5 *CRSI* Query Performance Evaluation

In Figure 4 we depict the query performance of *CRSI* for selection queries for the `INFOBOXES` dataset for various values of k and t. More specifically, in the percentile boxplot the thick black dot corresponds to the mean query response time observed, and the width of the box around the mean is proportional to the density of the corresponding query response time values. We observe that the mean response time is less than 50 ms and that it consistently decreases as the threshold t increases or when the synopsis size k decreases. We also observe that the majority of observed AREs are less than the mean ARE and that there are very few outliers (about 100 ms). In our experiment we report probe times for the `PQ` algorithm, but other algorithms and datasets exhibit very similar distributions.

We now evaluate *CRSI*'s query performance for self-joins and test how each parameter affects the response time. The self-joins were executed as an indexed nested loop join. This stresses the performance of the index, since conceptually all features have to be compared to each other. As we saw earlier, `ppjoin++`'s inverted index could not be created in reasonable time for large features and thus we were not able to compare our query time results with it. Such a comparison would not be fair anyway, since `ppjoin++` works on an interleaved-way creating posting-lists and performing the self-join at the same time, while our approach is targeted towards persistence and repeated usage of the index.

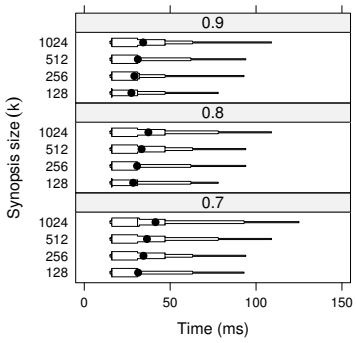


Figure 4: Similarity selection response time distribution of the `PQ` algorithm for the `INFOBOXES` dataset and various values of k and t.

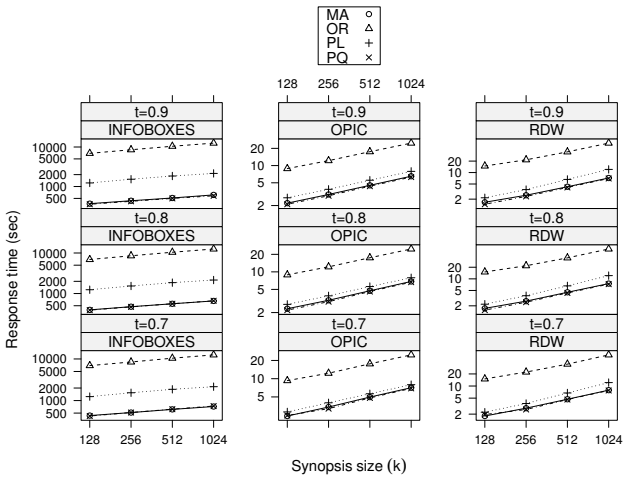


Figure 5: *CRSI* Self-Join performance for several datasets and values of k and t.

Figure 5 presents the results. Both axes are depicted in logarithmic scales. The pivot join approach outperforms consistently (by many orders of magnitudes) the simple `OR` approach. It also outperforms by a factor of two, the approach where we take the frontier of the posting lists into account. On the other hand, the `MA` algorithm, which handles the pivot join using an optimistic pruning, did not offer any significant benefits comparing to `PQ`. It only seems to have a slight positive effect for synopsis size k greater than 1,024.

6.6 *CRSI* Accuracy Evaluation

In the analysis we provided strong probabilistic guarantees about the errors of $\hat{J}$ using the proposed index. In this section we demonstrate that the actual errors of the estimations are much smaller than described.

More specifically, we examined the `OPIC` dataset for $t = 0.8$ and various values of k to see if the average relative error was in the predicted bounds that can be seen in Figure 1. The absolute relative error that we observed can be seen in the percentile boxplot in Figure 6. The thick black dot corresponds to the mean ARE observed, and the width of

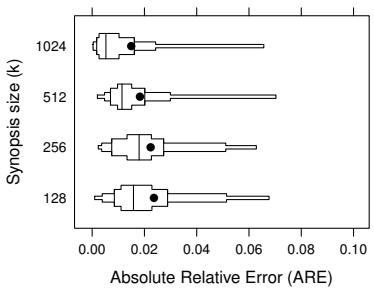


Figure 6: The observed ARE distribution using the `OPIC` dataset for $t = 0.8$ and various values of k.

the box around the mean is proportional to the density of the corresponding ARE values. We observe that the mean ARE is around 2% for $k = 128$ and that it consistently decreases as k increases. We also observe that the distribution is quite skewed. The majority of observed AREs are concentrated below the mean ARE. However, there are very few outliers (about 7% ARE) that increase the mean ARE. This effect is much less pronounced for large synopsis size k.

In summary, we observe that absolute relative error is less than the theoretical one in real datasets. This is mainly due to the fact that we used the Chebychev inequality, which can be very loose. Thus, the actual errors that our predictions make are even smaller than the theoretical one. We must note that, the higher the value for k is, the less we err in our predictions. We can observe a trade-off between large and small k. If it is large, then the errors we make are very small, but the similarity index is larger, its construction takes longer, and the query time is longer. If it is small on the other hand, then we have larger errors, i.e., lower quality, but the construction of the similarity index takes less time, the similarity index is actually smaller and the query time is smaller.

Depending on our application and our tolerance to accuracy errors, one can select the appropriate value of k, using either our pessimistic analysis in Section 3 or the more pragmatic Figure 6.

7. RELATED WORK

This section describes work related to ours.

7.1 KMV Synopses

Bar-Yossef et al initially introduced the main idea behind the KMV synopsis in [3]. Weighted versions of similar synopses are discussed in [12, 13]. Unbiased estimators for distinct counts, intersections, and unions are analyzed in [5]. In our work, we are extending the definition of KMV synopses and we use two versions of it: The complete and the incomplete. Based on that, we create unbiased estimators for the Jaccard similarity and use them to find similar features. KMV synopses have been successfully applied for the estimation of the size of the result of a similarity query in [16].

7.2 Near Duplicate Object Detection

There is a lot of work on the problem of near duplicate object detection under many different contexts. Some researchers find the nearest neighbor of a given query point under different metric functions, others find near duplicate documents for eliminating the indexing of similar web pages. Other variations of the same problem are set similarity, all pairs similarity, and the similarity join.

There are two main different approaches that are used in order to solve this problem. The first one is the exact solution and the second one is the approximate solution.

Exact solution To the best of our knowledge, all previous exact solutions [1, 4, 25], make use of an inverted index to speed up the process. The first algorithm that was developed is the `Probe_Cluster` algorithm [21]. The first algorithm though that has precise performance guarantees is described in [1]. Bayardo et al. [4] proposed the `all-pairs` algorithm, which is shown to perform better than the algorithm in [1]. The latest and best exact algorithm in terms of efficiency until now is described in [25]. The algorithm presented in [25] is called `ppjoin` and this is the algorithm we are comparing our results against in Section 6. In [24], the typical similarity threshold t is replaced with a top-k threshold, that returns only the k most similar features (regardless how high or low the threshold is).

Approximate solution The "classic" work in approximate near duplicate object detection is [18], where the Locality Sensitive Hashing (LSH) is described for the first time. The idea of LSH is to treat each record as a vector and create a signature for each one of the vectors. Then, for each record, based on its signature (k minimum hash values from k different hash functions — we use the k minimum hash values of one hash function), it is "easy" to determine which other records are similar to it, under some probabilistic bounds that are provided. There has been work that extends LSH, with more sophisticated algorithms [15], but the basic ideas are the same. Also, Charikar in [9] uses rounding algorithms for LPs and SDPs in the context of approximation algorithms to solve this problem. This usage can be viewed as LSH schemes for several collections of objects.

In the context of duplicate document detection, many randomization techniques have been proposed in [8]. However, it is not clear how to index and incrementally maintain them.

Even though there is a lot of work in set similarity in general, there is no work that we know of in the context of data profiling. In this context, the databases of interest can be extremely large, with lots of updates, insertions and deletions every day. All the ideas mentioned so far are static, i.e., they do not assume that new data can arrive, or old data can be deleted or altered. Thus, there are many characteristics in our setting that make our problem unique and none of the previously given solutions can solve it. Also, they ignore functions like the foreign-key distance, which is important in our setting.

Differences with our method The main differences between our work and the previous related work is that we have all the following characteristics at the same time. The inverted index we construct and use is *extremely small* comparing to our original database. We give clear and very good in practice *error guarantees*. The persistent inverted index that we propose is *highly updateable*; both when new data arrive, are deleted or updated and when new columns are added or deleted. There are no requirements to have the data presorted at all times (as currently most exact similarity algorithms require) under some universal ordering. We

make use of *only one hash function*. Thus, we do not have the requirement to find a family of min-wise independent hash functions [7], that LSH for example imposes (and it makes its practical usage difficult). We only do one pass through our data. The problem we are solving is focused on large databases, with lots of features where doing more than one passes would be extremely impractical.

7.3 WAND Operators

In [6] a way of performing Weighted-AND (WAND) queries is described. Queries of the form n/m can be thought of as a special case of WAND queries. In our setting though, there are differences like (1) the fact that there are two kinds of KMV synopses (complete/incomplete) which complicates the process and (2) we can handle more complex queries than the basic n/m: Return the pairs of features that have a Jaccard similarity of at least 0.8 and overlap similarity at least 20 items. Such queries can be answered using the techniques we have developed.

8. CONCLUSIONS

We propose a novel approximate similarity index for handling a large class of similarity queries. We studied the accuracy of the results both analytically and experimentally. Overall the index provides very strong accuracy guarantees. We demonstrated that the index is orders of magnitudes smaller than the original data, that it can be easily maintained and used to answer a variety of similarity queries. We showed that the index construction and maintenance does not require the a-priori knowledge of any similarity thresholds. In addition, the index has the benefit that it can be deployed using existing inverted index implementations. Finally, we describe algorithms that exploit (1) properties of our estimators and (2) skip-lists of the inverted index to optimize the query performance of the index. We also, experimentally, show that the overall approach is applicable to large range of real datasets.

9. REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient Exact Set-Similarity Joins. In *VLDB*, pages 918–929, 2006.

[2] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[3] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting Distinct Elements in a Data Stream. In *RANDOM*, pages 1–10, 2002.

[4] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all Pairs Similarity Search. In *WWW*, pages 131–140, 2007.

[5] Kevin Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On Synopses for Distinct-Value Estimation under Multiset Operations. In *SIGMOD*, pages 199–210, 2007.

[6] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient Query Evaluation using a Two-Level Retrieval Process. In *CIKM*, pages 426–434, 2003.

[7] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations (extended abstract). In *STOC*, pages 327–336, 1998.

[8] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic Clustering of the Web. *Comput. Netw. ISDN Syst.*, 29:1157–1166, 1997.

[9] Moses S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *STOC*, pages 380–388, 2002.

[10] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*, page 5, 2006.

[11] Steve Chien and Nicole Immorlica. Semantic Similarity between Search Engine Queries using Temporal Correlation. In *WWW*, pages 2–11, 2005.

[12] Edith Cohen and Haim Kaplan. Bottom-k Sketches: Better and More Efficient Estimation of Aggregates. In *SIGMETRICS*, pages 353–354, 2007.

[13] Edith Cohen and Haim Kaplan. Summarizing Data using Bottom-k Sketches. In *PODC*, pages 225–234, 2007.

[14] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient Similarity Search and Classification via Rank Aggregation. In *SIGMOD*, pages 301–312, 2003.

[15] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in high Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[16] Marios Hadjieleftheriou, Xiaohui Yu, Nick Koudas, and Divesh Srivastava. Hashed Samples: Selectivity Estimators for Set Similarity Selection Queries. *PVLDB*, pages 201–212, 2008.

[17] Monika Henzinger. Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms. In *SIGIR*, pages 284–291, 2006.

[18] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards removing the Curse of Dimensionality. In *STOC*, pages 604–613, 1998.

[19] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Comput. Surv.*, pages 43–45, 1996.

[20] Mehran Sahami and Timothy D. Heilman. A Web-Based Kernel Function for Measuring the Similarity of Short Text Snippets. In *WWW*, pages 377–386, 2006.

[21] Sunita Sarawagi and Alok Kirpal. Efficient Set Joins on Similarity Predicates. In *SIGMOD*, pages 743–754, 2004.

[22] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten. Evaluating Similarity Measures: a Large-Scale Study in the Orkut Social Network. In *KDD*, pages 678–684, 2005.

[23] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. SpotSigs: Robust and Efficient Near Duplicate Detection in Large Web Collections. In *SIGIR*, pages 563–570, 2008.

[24] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k Set Similarity Joins. In *ICDE*, pages 916–927, 2009.

[25] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient Similarity Joins for Near Duplicate Detection. In *WWW*, pages 131–140, 2008.