Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment

Raul Castro Fernandez, Jisoo Min, Demitri Nava, Samuel Madden
CSAIL, MIT
<raulcf, jisoomin, demitri, madden>@csail.mit.edu

***Abstract*—Data analysts often need to find datasets that are similar (i.e., have high overlap) or that are subsets of one another (i.e., one contains the other). Exactly computing such relationships is expensive because it entails an all-pairs comparison between all values in all datasets, an $O(n^2)$ operation. Fortunately, it is possible to obtain approximate solutions much faster, using locality sensitive hashing (LSH). Unfortunately, LSH does not lend itself naturally to compute containment, and only returns results with a similarity beyond a pre-defined threshold; we want to know the specific similarity and containment score.**

The main contribution of this paper is LAZO, a method to simultaneously estimate both the similarity and containment of datasets, based on a redefinition of Jaccard similarity which takes into account the cardinality of each set. In addition, we show how to use the method to improve the quality of the original JS and JC estimates. Last, we implement LAZO as a new indexing structure that has these additional properties: i) it returns numerical scores to indicate the degree of similarity and containment between each candidate and the query—instead of only returning the candidate set; ii) it permits to query for a specific threshold on-the-fly, as opposed to LSH indexes that need to be configured with a pre-defined threshold a priori; iii) it works in a data-oblivious way, so it can be incrementally maintained. We evaluate LAZO on real-world datasets and show its ability to estimate containment and similarity better and faster than existing methods.

I. INTRODUCTION

Analysts in modern organizations are faced with an ever increasing number of heterogeneous and disconnected datasets. One way to help analysts manage this complexity is to provide them with tools that discover data related to each other in particular ways. For example, finding pairs of datasets that are *similar* to each other helps analysts identify duplicate tables or complement a table of interest with additional, similar information. Similarly, finding pairs of datasets with a *containment* relationship—e.g., the values of a column are fully contained in another column—helps in identifying primary-key/foreign-key (PKFK) relationships as well as derived datasets. These can be used to help analysts join datasets together. Similarity and containment are the measures used by Aurum [7] as well as [13], [21], [22] to relate datasets together.

Two commonly used metrics to measure *similarity* and *containment* relationships between columns are Jaccard similarity (JS) and Jaccard containment (JC). Jaccard similarity measures the size of the intersection of the columns over the size of their union i.e., $JS(X, Y) = |X \cap Y|/|X \cup Y|$ for two columns X and Y. Jaccard containment measures the ratio of the size of the intersection of the two columns to the size of one of the sets (it is asymmetric). For example, for X it is defined as $JC = |X \cap Y|/|X|$.

With repositories containing a large number of columns, computing these metrics exactly requires an all-pairs comparison of the columns ($O(n^2)$), which is prohibitively expensive, as we demonstrate in Section II. As a result, common practice is to rely on *approximate* methods, which scale to larger datasets, and are acceptable for data discovery tasks, which are inherently exploratory and imprecise.

One widely used technique for approximating Jaccard similarity is locality sensitive hashing (LSH), which is used in many applications [8], [14]. With LSH, it is possible to hash the columns in the databases, in a way that *similar* columns hash to the same bucket. This reduces the complexity of finding candidates to $O(n)$.

Despite the promising properties of LSH, there are several important limitations that make it difficult to use directly for the data discovery problem we are interested in:

LSH Cannot Estimate Containment. Estimating containment requires estimating the size of the intersection, which is difficult to do with LSH [2]. A few methods have attempted to adapt LSH schemes for containment but either i) they do not support incremental indexing, which is necessary in the context of modern organizations, where existing data is continuously changing and new data arriving; or; ii) they do not work well in the context of databases [18]. Although non-LSH methods exist for computing containment, such as [19], they almost all involve some form of all-pairs comparison, as we describe in Section VI.

Building MinHash sketches is computationally expensive. Building a MinHash sketch [5] of a set involves hashing every value K times with different independent hash functions, with K usually in the hundreds. This quickly becomes a performance bottleneck, as we show in the next section. For this reason, many methods have recently been proposed that compute sketches by hashing the data only once, i.e., one-permutation hashing methods such as OOPH [16] and many others [11], [12], [17]. Unfortunately, the reduction in building cost comes with significant reduction in estimation quality.

Indexes do not return a similarity score. In discovery scenarios we need to know the similarity *score* between a query and the candidates e.g., finding whether two columns are 30% or 70% similar to each other. LSH indexes are configured to return the set of columns with a similarity to the query higher than a pre-specified threshold, and do not contain the score. A baseline solution to obtain the similarity score of candidates is to index the data into indexes configured with different thresholds and then query them all, determining at which threshold a result becomes a candidate. This solution, however, increases the storage needs as well as the indexing and querying time.

This paper introduces the LAZO method, the first approach that simultaneously estimates containment and similarity with LSH. The gist of the LAZO method consists of a redefinition of JS based on the cardinality of the sets that enables estimation of intersection and union independently. This, in turn, allows estimation of both JS and JC. The JC estimate depends on that of JS; i.e., they are coupled. We use this coupling to our advantage. In particular, we know that the JC between two columns cannot be larger than 1, and yet, it often happens that an initial estimate is greater than this because the JC estimate depends on the JS estimate, which is not perfect. We use this invariant to detect and correct our estimate of JS. Practically, this means that LAZO makes faster-to-build sketches, such as OOPH [16], perform as well as the more accurate but slower to build MinHash sketches.

Along with the LAZO method, we describe an implementation of a LAZO index, which returns the similarity and containment scores along with each candidate. In addition, the index can be queried on-the-fly for arbitrary thresholds, instead of requiring pre-configuring the index with a specific threshold a priori. This addresses the third challenge above. To achieve all these properties LAZO only requires column cardinalities, which are simple to obtain. In many cases the cardinality is readily available—such as in databases, where it is used for query optimization. When it is not available, we estimate it using HyperLogLog [9].

In practice, we have used Lazo to compute JS and JC relationships at any threshold in the context of Aurum [7], our data discovery system. Lazo's ability to index columns incrementally is crucial to maintain Aurum up to date.

In summary, the contributions of this paper are:

• We develop the first method to estimate JS and JC at the same time in $O(n)$ time that does not depend on data distributions and performs well in practice. This addresses the first challenge above.

• By using LAZO we derive an *error correcting heuristic* (ECH), which, by detecting invariants in the JC estimate, allows us to improve the estimation quality of sketching methods. This means that we can use one-hash sketches like OOPH, which are much faster to compute than traditional MinHash, without sacrificing any estimation accuracy. This addresses the second challenge above.

• The LAZO method is compatible with existing sketches, such as MinHash and OOPH.

• We describe a new index implementation which retrieves candidates with different similarities and containment scores, supports on-the-fly querying and can be maintained incrementally, as data arrives.

	All Pairs Disk	**All Pairs RAM**	**MinHash LSH**
DWH (n=1690)	165s	9.5s	7.4s
MassData (n=5514)	261s	201s	82s
canadagov (n=97621)	-	>24h*	17min

TABLE I
RUNTIME OF ALL-PAIRS AND MINHASH/LSH FOR 3 DIFFERENT DATASETS. (* DID NOT FINISH AFTER 24 HOURS)

In addition to estimating JS and JC simultaneously and in a data-oblivious way, we show in the evaluation section that: i) the containment estimation quality of LAZO is higher than that of LSHEnsemble, especially for the case when we are using high containment thresholds (where one column is very nearly contained in another) to discover inclusion dependencies, which is the most important application of containment in data discovery and integration applications in the database literature; ii) we show that we improve OOPH's accuracy, making it the preferred choice when compared with the slower-to-build MinHash; iii) we demonstrate our LAZO index implementation performance and describe its interface, which is better suited for data discovery tasks. Finally, we demonstrate that LAZO's overhead is negligible when running on several large datasets.

The rest of the paper is organized as follows. In the next section, we provide background and explain the limitations of existing sketching techniques. We introduce our main method in Section III and its implementation in Section IV. We then evaluate our techniques in Section V, before presenting related work (Section VI) and conclusions (Section VII).

II. PRELIMINARIES

In this section we introduce MinHash and LSH (section II-A): techniques to solve an approximate nearest neighbor problem, and show the practical performance difference with all-pairs methods in section II-B. Because MinHash suffers a performance bottleneck, we walk through the family of one-permutation hashing methods, OPH, in section II-C and demonstrate the performance and quality gap between MinHash and OPH methods in II-D. Finally, we discuss the state of containment estimation in II-E.

Problem Statement. Consider the problem of finding pairs of columns that satisfy a relationship r, among all the columns of a large dataset. We say two columns c_1 and c_2 are related if $r(c_1, c_2) > \delta$, for a threshold δ. An exact algorithm to find all related columns exhaustively computes the relationship strength between all pairs of columns and returns the pairs with strength above the threshold. This *all-pairs* comparison is an $O(n^2)$ operation, which becomes prohibitively expensive when n (the number of columns in the dataset) is large.

A. MinHash and LSH

It is possible to find all pairs of similar columns in $O(n)$ by solving an approximate nearest neighbor problem with LSH instead of an all-pairs one. For that, it is first necessary to obtain a representation of each column, i.e., a sketch. Such sketches must have two properties: i) they must be fast to compute; ii) they can be used along with LSH; i.e., indexed in a hash table to find collisions. According to the second property, there are two well-known sketch methods. One is based on random projections (SimHash [3]), and it is good for approximating cosine similarity. The other method, which is the focus of this paper, is based on random permutations (MinHash [5]), and is good for approximating Jaccard similarity, which is defined as $JS(X,Y) = |X \cap Y|/|X \cup Y|$

Minwise hashing. Consider two sets, or columns, $c_1, c_2 \in C$ with each set consisting of elements from a universe of elements $\forall e \in c_i, e \in E$. Consider a random hash function, h, that maps an element from E to S, the universe of hash values, and obtains the minimum value $s \in S$ produced by applying h on c_i, that is $\min(h_j(c_i))$. For a given h, the probability of the minimum value between two sets being the same is equivalent to the JS between the sets [5], that is:

$$P(\min(h_i(c_1)) = \min(h_i(c_2))) = \frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} \tag{1}$$

Then, given K independent random hash functions, h_1, h_2,...,h_K, the JS is estimated with MinHash as:

$$\hat{JS}(X,Y) = \frac{1}{K}\sum_{i=1}^{K}[\min(h_i(x)) = \min(h_i(y))] \tag{2}$$

with $[x]$ being the Iverson bracket, that returns a 1 when x is true and 0 otherwise. In practice, this means that each value of each column is hashed K times. The resulting set of K values is the MinHash sketch.

Once available, the MinHash sketch is used along with LSH to retrieve other sketches with a Jaccard similarity greater than a specified threshold in $O(n)$ as explained next.

Locality Sensitive Hashing Index. LSH indexes the sketches in a way that those that are similar are more likely to hash into the same hash table entry. The LSH index can be queried with a sketch, and will return a set of sketches, called *candidates*, that are likely to have a Jaccard similarity of more than a pre-configured similarity threshold. It works as follows:

LSH divides each sketch into a set of bands b, each with $r = K/b$ hash values. Then, each of the band values are indexed (after concatenation) into a hash table entry. The idea is that MinHash sketches that collide in the same entry are likely to be candidates. The parameters b and r are chosen based on δ (according to the method in [10]), the similarity threshold, and with the aim of minimizing false positives and negatives.

The advantage of using LSH is that it becomes possible to retrieve, for a given column, other columns with a Jaccard similarity above δ in $O(n)$. This makes it a good choice when the number of columns is large and approximate results are sufficient; it fits well with our discovery scenario.

B. All-Pairs Vs LSH

To illustrate the performance difference between All-Pairs and LSH, we measured the time they take to obtain the Jaccard similarity of more than 0.7 for two small datasets DWH and MassData ($n = 1690$ and $n = 5514$ columns, respectively), and a larger one, canadagov from [22] ($n = 97621$ columns). We executed All-Pairs in two modes. First, assuming data does not fit in memory and reading it from disk (All-Pairs Disk). Second, although generally unrealistic, with the assumption that data fits in memory to focus on the CPU cost differences (All-Pairs RAM). The results for both modes as well as for MinHash/LSH are shown in table I. We highlight two points: i) the increased IO cost of reading a subset of the data for each comparison means that All-Pairs Disk is much slower than a MinHash/LSH solution, where data is read only once; ii) even for the unrealistic All-Pairs RAM, runtime grows much more quickly with n than MinHash/LSH. In the case of the larger dataset, All-Pairs RAM does not finish after 24 hours while MinHash/LSH takes only 17 minutes.

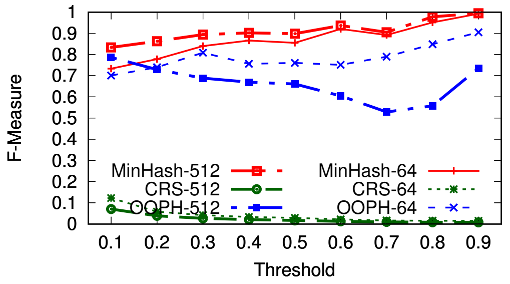

Fig. 1. F-measure for the MassData dataset using the MinHash, CRS and OOPH sketch along with an LSH index with K=64 and 512.

Regardless of using All-Pairs or MinHash/LSH, it is possible to reduce n by filtering columns by type, size, etc. However, after filtering, the All-Pairs method still requires $O(n^2)$ operations. More importantly, these filters are brittle to data quality problems; e.g., if we only match the same types, an ID that appears in a table as an integer but as a string on a derived table that was dumped into a lake would be missed.

In summary: when approximate solutions suffice, such as in the data discovery applications that motivate this work, MinHash/LSH are a much more performant solution. These techniques are the focus of this paper.

Although MinHash has much better runtime than an all pairs solution, the MinHash sketch still requires hashing data K times, which (as we will show) becomes a bottleneck in many applications. This bottleneck motivates one-permutation hashing methods, which we explain next.

C. One-Permutation Hashing Methods

One-permutation hashing methods compute sketches using only one hash function, unlike MinHash which needs K. We describe next the different types of one-permutation hashing methods and discuss their benefits and disadvantages.

1) Conditional Random Sampling: Conditional Random Sampling (CRS) [11], also referred to as K-MinHash or bottom-K sketch consists of hashing data only once, and selecting the top-K minimum hash values—instead of hashing K times and selecting the minimum per hash function.

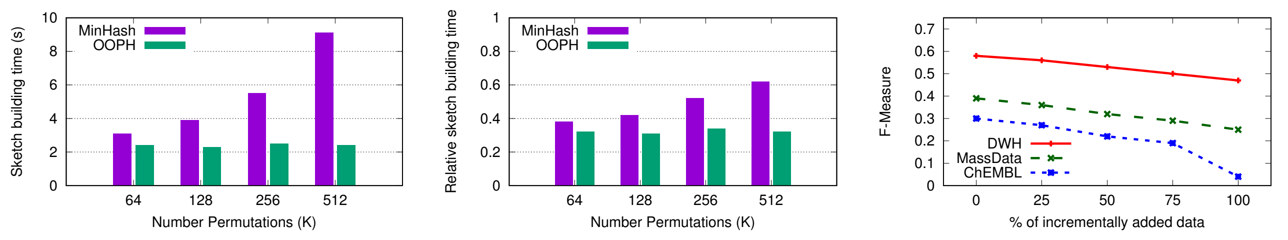

Fig. 2. Left: MinHash and OOPH building times for a real dataset as K changes. Center: Percentage of MinHash and OOPH building time in an end-to-end workflow when K changes. Right: F-Measure decreases with LSHEnsemble as a higher percentage of data is incrementally added.

Unfortunately, this simple method does not work well with LSH because the sketches are not "aligned", which is a well known problem [17]. When CRS sketches are indexed in LSH—by splitting them into bands—the values in the bands are not aligned to each other, as they would be when computed with independent hash functions. This means that collision-free CRS sketches do not imply dissimilarity: similar values may not be part of the top-K minimum values. These sketches perform poorly when used with LSH (see Fig. 1), and they are not an option for our setting.

2) One Permutation Hashing: With OPH [12], instead of hashing K times with independent hash functions, the values of the set are hashed once and partitioned into K different buckets, and the sketch is comprised of the minimum hash value per bucket. The major drawback of this method is the existence of empty buckets, which appear when data is skewed. If untreated, empty buckets would collide with each other, artificially producing false similar sets. This renders OPH unusable for our data discovery setting, because real data is skewed. Fortunately, recent methods have proposed to handle empty buckets through different *densification* strategies, which address the problem.

Densifying One Hash Permutation. To deal with empty buckets, One-Hash Permutation with Rotation (OPHR) [17] fills empty buckets based on the closest non-empty bucket to the right and some constant, which indicates the number of "hops" between the empty and non-empty bucket [17].

Although this is an effective way of treating empty buckets, the quality of the estimation deteriorates when there are consecutive empty buckets, which is common in the context of databases. To improve on OPHR, Optimal One-Hash Permutation (OOPH) [16] uses 2-universal hash functions to not only select consistently the bucket from which to draw values when a bucket is empty, but also to do so with sufficient randomness to avoid the consecutive empty bucket problem. OOPH is the state-of-the-art method for one-hash permutation sketches.

D. The Gap between MinHash and OOPH

Given the previous discussion of different sketching methods, we now explore three questions: i) Is there a measurable runtime difference between OOPH and MinHash when building sketches?; ii) Do these differences matter in real applications?; and iii) Is there an estimation quality difference between these sketches?

To answer these questions, we use one of the real datasets from our evaluation and implement an end-to-end workflow that: i) reads the data; ii) creates sketches with MinHash or OOPH; iii) indexes them in a traditional LSH index; and iv) queries the index to retrieve similar candidates. We use a small dataset for this microbenchmark; note time grows linearly with dataset size. We vary K, the number of permutations.

The results in Fig. 2 (left) show that the cost of building MinHash grows with K, unlike OOPH. Note that we assume we hash each value read exactly once. In practice, we may want to first produce n-grams from the read data, and then hash the multiple n-grams, as in [21]; this would further widen the gap observed in these graphs.

Do these differences matter in practice?. In short, yes. These time differences, along with the cost of I/O, represent a large fraction of the total computation time in our end-to-end workflow. Specifically, Fig. 2 (center) shows the relative cost of building the sketches with respect to the total workflow time for all the configurations of K. The results show that building the sketches take a non-negligible portion of the entire time, especially when K grows.

Do OOPH and MinHash estimates differ in quality?. Again, yes. The gain in performance of OOPH comes with the cost of lower estimation accuracy. To demonstrate this, we obtain the truly similar pairs, i.e., ground truth, for the dataset and measure the the F-measure of the results obtained using the workflow described above. Note that in general it is possible to perform an expensive post-processing step on the candidates obtained, in which an exact computation of similarity is performed to remove false positives. This step improves the precision and not the recall. We do not perform that post-processing step in this paper because we are interested in understanding the properties of the different sketches and techniques directly.

The results of the experiment are shown in Fig. 1, showing the F-Measure for 1 dataset (see Section V for details), with different K values. The first obvious result from the figures is that CRS does not perform well when used along with LSH, as we had anticipated. Second, OOPH performs significantly worse than MinHash, with a difference in the F-measure of up to 0.3 for high thresholds.

Thus, we have a tradeoff with existing methods: either we stick to MinHash, which is expensive to compute but achieves good quality, or we use OOPH, which sacrifices quality for faster to compute sketches. As a way around this tradeoff, the method we present in this paper bridges this estimation-accuracy gap: LAZO is compatible with both MinHash and OOPH, when configured to use OOPH it matches the MinHash accuracy while maintaining the building time of OOPH.

E. State of Containment Estimation

The previous discussion is on methods to estimate similarity. We now turn our attention to containment.

The LSHEnsemble technique [22] is the state of the art method to estimate containment. It suffers from two problems we have identified in practice. First, LSHEnsemble needs to know a-priori the entire set size distribution, that is, all the cardinalities of all the sets. If new sets become available later, all datasets need to be re-inserted eventually to avoid a deterioration of the estimation quality. We have measured whether this deterioration is important. Fig. 2 (right) shows the F-measure achieved by LSHEnsemble on the same datasets used above, but this time we measure containment. In the X axis, we indicate the amount of data that is inserted after the index is created. The first point, with label '0' means that all data is inserted at once; this is the setting for which LSHEnsemble is designed, so this is the baseline. As a larger percentage of the data is inserted after the index is created (move in X axis to the right), the estimation quality deteriorates.

In the context of databases in organizations, data is continuously changing, so this deterioration poses an important problem. Although it could be avoided by re-inserting the sets from time to time, this would introduce two additional challenges: i) to decide when to re-insert; ii) additional overhead of reading the data from the source and creating the LSHEnsemble index all over again.

The second drawback we found when using LSHEnsemble is its low recall for high containment thresholds. Good recall at high thresholds is important in databases because candidate PKFK are precisely obtained by identifying pairs of columns with high containment, i.e., inclusion dependencies.[1] A low recall means that many of the candidate pairs remain uncovered. We note that for the application of linking datasets, recall is generally more important than precision. It is always possible to remove false positives by performing a post-processing step after using LSH to measure the real similarity/containment between the candidate pairs, but it is not possible to anticipate false negatives.

LAZO addresses both of the problems above. First, it accepts sketches as they become available; it does not need to know the entire set size distribution a priori. Therefore, its estimation quality does not deteriorate over time as data changes. Second, it achieves good recall even for high thresholds. This in turns implies a big improvement in F-measure with respect to LSHEnsemble. We compare quantitatively with LSHEnsemble in our evaluation section, where we further discuss the results.

[1]Although inclusion dependencies imply theoretically that containment equals 1, dirty and missing data means this constraint is often relaxed, hence we are interested in high threshold and not necessarily 1.

III. THE LAZO METHOD

The key advantage of LAZO is that in addition to an OOPH sketch of the data, it only requires the cardinality of the individual sets to simultaneously estimate containment and similarity. We present the LAZO method in Section III-A and a heuristic to correct estimates in Section III-B.

A. Cardinality-Based Coupled JS/JC

The key insight that enables the estimation of JS and JC is that it is possible to define JS in terms of the cardinalities of its individual sets, instead of using the cardinalities of their intersection and union. The expression of JS for sets X and Y with known cardinalities, $|X|$ and $|Y|$, is given by:

$$JS(X,Y) = \frac{\min(|X|,|Y|) - \alpha}{\max(|X|,|Y|) + \alpha} \tag{3}$$

and the expression for JC is given by:

$$JC_x = \frac{\min(|X|,|Y|) - \alpha}{|X|} \tag{4}$$

where α is a parameter that we want to estimate. The expression for JS follows from the observation that the maximum Jaccard similarity between X and Y, $JS_{max}(X,Y)$ is:

$$JS_{max}(X,Y) = \frac{\min(|X|,|Y|)}{\max(|X|,|Y|)} \tag{5}$$

That is, the maximum possible JS between two sets is the cardinality of the smallest set, which is the maximum possible size of the intersection, divided by the cardinality of the largest set, which is the maximum size of the union. Two non-empty disjoint sets, which have a $JS = 0$, will have a value of $\alpha = \min(|X|,|Y|)$. They have an empty intersection and a union as big as $|X| + |Y|$. Hence, any similarity between the sets that is smaller than JS_{max} will *take* elements from the intersection and *add* them to the union (see Eq. 3).

The key property of this expression is that with an estimate of α it is possible to estimate both the intersection and the union of the sets independently because these now depend on the cardinalities of the sets and α. So in summary, in order to estimate JC, we need α. We explain next how to obtain α.

Obtaining an estimate of α. It is possible to estimate α by obtaining first an estimate of JS, $\hat{JS}$, using for example OOPH. With a value for $\hat{JS}$ available, we rearrange Eq. 3 to obtain α as:

$$\hat{\alpha} = \frac{\min(|X|,|Y|) - \hat{JS} * \max(|X|,|Y|)}{1 + \hat{JS}} \tag{6}$$

With a value for $\hat{\alpha}$, it becomes possible to estimate the intersection of the sets independently from their union, and therefore $\hat{JC}_x$ and $\hat{JC}_y$, as follows:

$$\hat{JC}_x = \frac{\min(|X|,|Y|) - \hat{\alpha}}{|X|}; \hat{JC}_y = \frac{\min(|X|,|Y|) - \hat{\alpha}}{|Y|} \tag{7}$$

Essentially, with an estimate for $\hat{JS}$ and the cardinality of the sets, $|X|$ and $|Y|$ available, it is possible to obtain a coupled estimation of both JS and JC, the metrics of interest.

B. Observing Error to Correct Estimates

One surprising consequence of the coupled-estimation of JS and JC based on set cardinalities is that it is possible in some cases to correct the original estimate of $\hat{JS}$ after observing the value of $\hat{JC}_x$ and $\hat{JC}_y$. We use $\hat{JC}_x$ to illustrate our error correction estimate, but the expressions apply equivalently to $\hat{JC}_y$. The real JC_x (not the estimate) is given in Eq. 4.

The expression contains a value for α. Of course, we do not know the real value, only the estimate computed above, $\hat{\alpha}$. For this reason, we can only estimate $\hat{JC}_x$.

The key insight of the error correction method is that there is an invariant that holds over JC: $\hat{JC}_x$ cannot be higher than $\hat{JC}^x_{max}$, which is defined as:

$$JC^x_{max} = \frac{\min(|X|, |Y|)}{|X|} \tag{8}$$

Thus, when the value obtained for $\hat{JC}_x$ (or $\hat{JC}_y$) exceeds JC^x_{max}, we know the estimation error is at least $\hat{JC}_x - JC^x_{max}$, and we can correct that error by adjusting $\hat{\alpha}$ so that $\hat{JC}_x = JC^x_{max}$. We refer to this correction as *error correction heuristic (ECH)*. The $\hat{\alpha}$ correction, in turn, improves the originally estimated $\hat{JS}$. Note that the estimation error may be larger than the adjustment we made above, however the adjustment is guaranteed to improve the estimate or leave it the same.

Summary. In this section, we showed how to obtain a coupled estimate of JS and JC, where the quality of JC depends on the estimated value for JS, and in which observable error in JC is used to improve the estimation of JS. We now explain how to use this with our LAZO implementation.

IV. LAZO IMPLEMENTATION

In this section we present LAZO's interface (IV-A) and its implementation, focusing on how to build, insert and query it IV-B. We finish with a discussion in Section IV-C.

As described in the previous section, LAZO is not a sketching method. Rather it is a way to estimate JC and improve the JS estimate from an existing sketch such as MinHash or OOPH. Specifically, our implementation builds an *index* using a supplied sketch, and then uses that index to return columns with non-zero JS and JC for any user-supplied input column.

A. LAZO *Interface*

The LAZO index is queried with a column represented by its sketch. For example, if we want to find columns with a similarity or containment relationship with respect to a *query column*, c_1, we query LAZO with the c_1 sketch, which will then return a set of *candidates*. Each candidate consists of a tuple $< key, js, jc >$ to identify the underlying column with its key, as well as the js and jc score that indicates the strength of similarity and containment. LAZO's main interface is:

- related(q) This function returns any *candidate* that has non-zero Jaccard similarity or Jaccard containment with respect to the input parameter q. Each candidate consists of: key, similarity and containment scores.

In other words, with a single call it is possible to understand the strength of both similarity and containment relationships between an input sketch and all other sketches. To find the relationships between all-pairs, we call the above function for every column.

We present results for single- and multi-thread implementations of LAZO in the evaluation section.

We explain now how to build, insert, and query the LAZO index to support the above interface.

B. LAZO*: Sketching, Inserting, Querying*

Estimating JC depends on obtaining a good estimate of $\hat{\alpha}$, which depends, in turn, on having access to a numerical estimate of JS. Obtaining this similarity score is the central challenge addressed by our LAZO index implementation.

A naive way to achieve our requirements would be to insert the sketch into multiple LSH indexes, each configured with a different threshold. Then, we could query the indexes and identify the highest threshold at which a specific column becomes a candidate. That threshold is a lower similarity threshold for the candidate and the immediately next threshold is its upper similarity bound. At this point, the estimate is within the lower and upper bound.

This basic design, however, multiplies the storage required and the indexing time, because a single sketch must be indexed in multiple LSH indexes—and the number of indexes depends on the desired granularity of thresholds. Furthermore, because it is necessary to query all indexes to obtain the numerical JS estimate, the query cost also increases.

The LAZO Index. We propose an alternative design that tackles both problems. When configuring an LSH index for a particular threshold, δ, and number of permutations, K, K is divided into bands, b with a number of rows so that $b{*}r <= K$. The band size is related to the similarity threshold. Therefore, different thresholds lead to different number of bands and rows. We say that a sketch consists of multiple *segments*, where each segment consists of the r values in each b. Each segment is then indexed into a different hash table, which is used to find collisions across sketches. Multiple thresholds will then require multiple different hash tables, all of which must be queried to find candidates. The intuition of our solution is that because we know all the thresholds we need to query, we know all the band sizes, and by finding the greatest common divisor (GCD) of those bands, we obtain the minimum sketch slice that needs to be indexed. The GCD band size will correspond to one of the lower thresholds. When querying for higher thresholds, their bands will contain several GCD bands, each stored in a different hash table. We need to query the individual hash tables and aggregate the results by taking their intersection.

Algorithm 1: Initialize the LAZO Index

```
input : $K$, number of permutations of underlying sketch,
        $D$, similarity threshold granularity
output: $I$, the LAZO index ready to be used
1 num_thresholds ← 1/$D$;
2 bands_and_rows ← getOptimalBandsAndRows($K$, num_thresholds);
3 gcd_band_size ← gcdOf(bands_and_rows);
4 num_ranges ← $K$ / gcd_band_size;
5 hash_tables ← [ hash_table for $k$ in num_ranges ];
6 hash_ranges ← [ $i$ * gcd_band_size for $i$ in num_ranges ];
```

We compare both implementations in the evaluation section and give more analytical detail about the costs in this section, after presenting our alternative design.

We describe next: i) how to prepare a sketch to be inserted in the LAZO index; ii) how to insert the sketch and; iii) how to query the index to retrieve candidates.

1) Building the LAZO *Sketch:* LAZO uses an existing JS sketch (i.e., OOPH or MinHash) as well as the cardinality of the column. In case the cardinality is not already available, we provide a wrapper that uses HyperLogLog [9] to obtain an estimate of the cardinality for which the sketch is being built. We show experimentally in the evaluation section that this estimate works well in practice.

2) Building the LAZO *Index:* We describe how to initialize the LAZO index (algorithm 1) and how to insert sketches into it (algorithm 2). The initialization algorithm takes as input parameters K, the number of permutations used to build the sketches, and D, which is a parameter that determines the granularity of the numerical estimate of JS. It returns I, which is the LAZO index ready to be used.

During the initialization stage, the algorithm uses D to determine the granularity of the JS estimates (line 1). For example, for $D = 0.05$, we have thresholds from 0 to 1 in increments of 0.05, totaling 20 total similarity thresholds. These values seem to provide fine-grained enough thresholds to suffice all applications we have found. For each threshold, the algorithm computes the optimal number of bands and rows in line 2 by using the method described in [10]. Once the bands for all thresholds have been computed, we obtain the band size as the greatest common divisor of all bands in line 3 which will be used to determine the number of ranges in which to divide sketches (line 4), and hence the required number of hash tables. The remaining of the initialization algorithm (lines 5 to 6) creates the hash tables and stores the sketch ranges corresponding to each one of the hash tables. The ranges indicate how to obtain the sketch segments.

Inserting sketches. When inserting a new sketch, the algorithm must store the cardinality for the sketch (line 2), slice the sketch into segments, according to the ranges determined during initialization (line 3), and finally store each segment into its corresponding hash table (lines 4 to 7), along with the key corresponding to the input sketch.

3) Querying the LAZO *Index:* We explain in detail the querying process in algorithm 3. The algorithm takes the LAZO index, I, the sketch that represents the query column, $sketch$, and then, optionally, either a JS or JC threshold, $js\_threshold$ or $jc\_threshold$ respectively. It returns C, which is a collection of tuples, $< key, js, jc >$. Each tuple

Algorithm 2: Insert in LAZO Index

```
input : $I$, the LAZO index,
        key, an identifier for the sketch to be inserted,
        sketch, the sketch to be inserted
1 cardinality ← get_cardinality(sketch);
2 I.[key] ← cardinality;
3 segments ← getSegments(sketch, $I$.hash_ranges);
4 for $i \in I.num\_ranges$ do
5     segment ← segments[$i$];
6     hash_table ← $I$.hash_tables[$i$];
7     insertSegment(hash_table, segment, key);
```

contains the candidate c represented with its key, as well as its js and jc score with respect to $sketch$. The candidates, C, are obtained as follows:

The algorithm iterates over the thresholds configured in the index, along with the corresponding bands and rows (line 4), starting with the highest threshold. Recall the bands and rows associated to each threshold were computed in algorithm 2, and are available in I. For each threshold, the algorithm determines how many of the segments fit into the corresponding band size. Because the segments were created using the greatest common divisor of all bands, it is guaranteed that the number will be an integer (line 6). Next, the algorithm finds all candidates that appear in *all* the segments corresponding to the entire band. That is, it queries each hash table corresponding to the band, and takes the intersection of the results (lines 8 to 11). This gives the candidates for one band. All candidates across bands are union together and stored along with the threshold at which they became candidates (line 12). Finally, the outer loop filters out candidates that have already been seen (if a set is a candidate at threshold 0.7, it is at threshold 0.6 too), and it associates the remaining ones with their threshold, which is the JS estimate we obtain.

Applying LAZO. With a set of candidates available, the next step is to obtain the estimates for intersection and union by using the LAZO method introduced in the previous section (line 15). The intersection and union estimates are then used to estimate JS (line 16). At this point it is possible to obtain $\hat{\alpha}$ (line 19), which is in turn used to estimate JC, in line 20 and 21. Note that even when only querying for jcx, we also obtain jcy because it is cheap and helps us find more opportunities to detect estimation errors. The error correction heuristic is then used in line 22 before returning the candidates. The final step of the algorithm is to filter out the candidates based on the input thresholds (if specified), in line 23.

C. Discussion

We discuss two aspects related to the LAZO index before presenting our evaluation results.

Implementation analysis. The *base* implementation creates an independent LSH index for each threshold, i.e., $\sum_{i=0}^{S} K/b_i$, with S the number of thresholds and b_i the band size for a specific threshold. Our design only needs $K/GCD(S)$ hash tables, where GCD is the greatest common divisor of the band sizes. Since GCD is as large as the smallest band, our solution will always consume less than or the same space than the base one. For example, for typical values of $D = 0.05$ and $K = 512, 64$ the Lazo index reduces the number of hash

Algorithm 3: Query LAZO Index

```
// Query the LAZO Index
input :  $I$, the LAZO index,
         sketch, the sketch representing query column,
         js_threshold
         jc_threshold
output:  $C$, a set of tuples of the form $key, js, jc$.
// Retrieve candidates
$C$ ← set();
candidates ← map[$\delta_i$− > [key]];
seen_candidates ← set();
for $\delta_i$, bands, rows in $I$.bands_and_rows do
    for $b \in$ bands do
        segments_per_band ← rows / $I$.gcd_band_size;
        band_candidates ← set();
        for $i \in$ segments_per_band do
            segment ← getSegment(sketch, $I$.hash_ranges, $i$, $b$,
              $I$.gcd_band_size);
            hash_table ← $I$.hash_tables[$i$];
            band_candidates ← intersection(hash_table[segment]);
    candidates.store($\delta_i$, band_candidates);
    candidates ← filter_seen_candidates(candidates, $\delta_i$, seen_candidates);
// Aggregate Candidates
for $c \in$ candidates do
    ix, un, ← apply_lazo($c$);
    js ← ix / un;
    min_card_set ← min(card(c), card(sketch));
    max_card_set ← max(card(c), card(sketch));
    $\alpha$ ← obtain_alpha(min_card_set, max_card_set, js);
    jcx ← (min_card_set - $\alpha$) / card(sketch);
    jcy ← (min_card_set - $\alpha$) / card($c$);
    js, jcx, jcy ← correct_estimates(card(sketch), card(c));
    if $js > js_{threshold}$ or $jc > jc_{threshold}$ then
        . add(($c, js, jcx, jcy$), $C$);
return $C$
```

tables by 4x and 5.5x respectively. The practical performance implications are shown in the evaluation section: indexing in Lazo is an order of magnitude faster and querying is 2-10x faster.

Lazo cost. With respect to a traditional LSH index, the LAZO index introduces the additional cost of applying the LAZO method. This cost is linear with the number of candidates, which in real datasets is much smaller than the number of original sets. Of course, in exchange for this small cost we obtain numeric scores for Jaccard similarity and containment simultaneously. We demonstrate in the evaluation section that the linear cost introduced by LAZO is negligible in the context of a full-fledged pipeline.

V. EVALUATION

• **How well does LAZO estimate Jaccard containment?** We want to understand LAZO's Jaccard containment estimation quality, (Section V-A).

• **Does LAZO improve accuracy of similarity estimates?** We demonstrate in Section V-B how LAZO improves the estimation quality of MinHash and OOPH through its error correction heuristic (ECH).

• **Is the LAZO overhead low enough to make the index practical?** We demonstrate in Section V-C with large datasets that the overhead is negligible in practice.

• **Microbenchmarks:** We compare the LAZO index against the baseline implementation. We analyze the effect of the parameter D on accuracy. Finally, we study the effect of cardinality estimation on the JS and JC estimates.

Datasets. We use the following datasets:

• MassData: This dataset consists of public governmental data from Massachusetts. It contains data about parking tickets and citizen complaints among many other things. It contains around 300 relations and around 6K columns.

• ChEMBL: This dataset is a well-known public chemical database containing information about drugs and compounds. The dataset consists of 70 relations with around 500 columns of many different sizes. We use ChEMBL 22 [20].

• DWH: This dataset contains relations with information about subjects, buildings, faculty, employees and other aspects of our university. The dataset consists of around 160 relations and about 1.7K columns.

• datagov: This dataset consists of 10K CSV files with around 100K columns and 210GB size in total. It was extracted by downloading all the CSV datasets from USA data.gov [1] and filtering out those files that had some quality problems in it, such as bad headers. We use it only to demonstrate the overhead of LAZO because we could not extract ground truth in reasonable time.

• canadagov: This dataset was used in [22] and made publicly available. It consists of around 100K columns from Canada open data and around 5GB on disk.

Setup. We run our experiments in an Intel(R) Xeon(R) CPU E7-4830 with 256GB RAM. We implement LAZO in Java [2] and use a value $D = 0.05$ for the LAZO index in all experiments unless we indicate the opposite. LAZO is trivial to parallelize. To demonstrate its scalability, we wrapped up LAZO in a multi-threaded implementation and run an experiment where we indexed 10K columns of 1K values each, and then found all pairs with non-zero similarity and containment. We measured the runtime when using different number of threads (up to 16). The results of Figure 9 (Center) demonstrate that the library scales linearly. In the remainder of the evaluation section we use a 1-thread implementation.

A. *Estimating Containment*

We evaluate the quality of containment estimation and compare it with that obtained by LSHEnsemble [22], the state of the art solution that had achieved the better results so far (we also considered ALSH [18], but LSHEnsemble achieves better results in the context of relational data, as shown in [22]). We configure LSHEnsemble with the same K as LAZO and use the same cardinality estimates.

Previously we showed how the estimation quality of LSH-Ensemble deteriorates if data is not inserted at once (Fig. II-A in Section II). When data is changing and evolving, the LSHEnsemble index will eventually need to be recomputed from scratch because its containment estimate requires data-dependent information. LAZO does not suffer from this problem because the method is data-oblivious. Despite this qualitative difference, we want to understand the estimation quality in comparison with LSHEnsemble even when data is unchanging. For this experiment, we assume all sketches are precomputed at once, and compare against LSHEnsemble in this setting.

[2]Open source code available at https://github.com/mitdbg/lazo

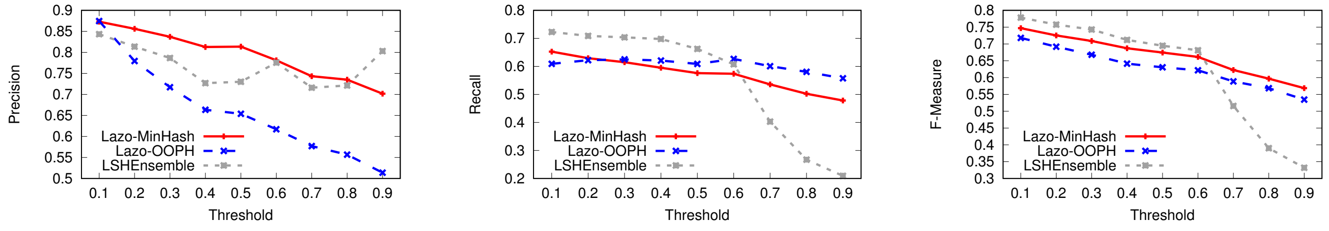

Fig. 3. Containment comparison of LAZO-MinHash/OOPH and LSHEnsemble. Showing precision, recall and F-measure for the MassData dataset

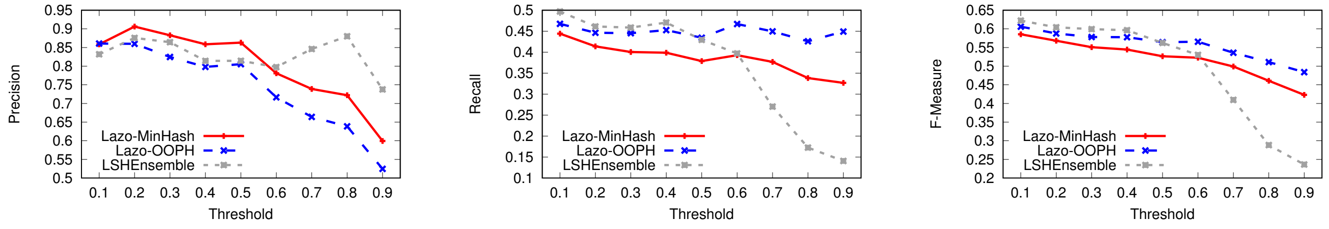

Fig. 4. Containment comparison of LAZO-MinHash/OOPH and LSHEnsemble. Showing precision, recall and F-measure for the ChEMBL dataset

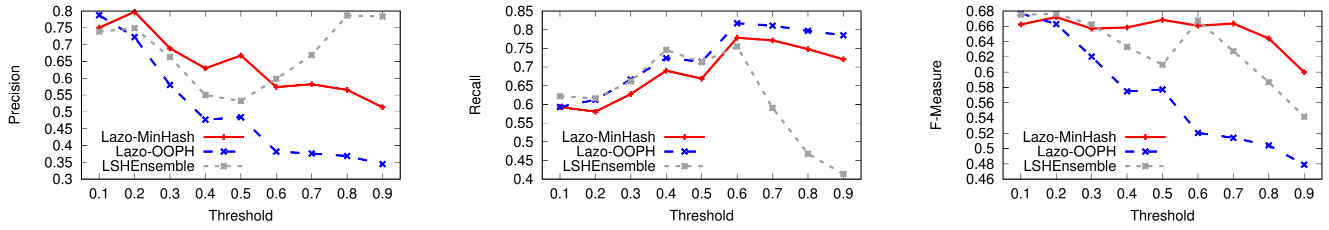

Fig. 5. Containment comparison of LAZO-MinHash/OOPH and LSHEnsemble. Showing precision, recall and F-measure for the DWH dataset

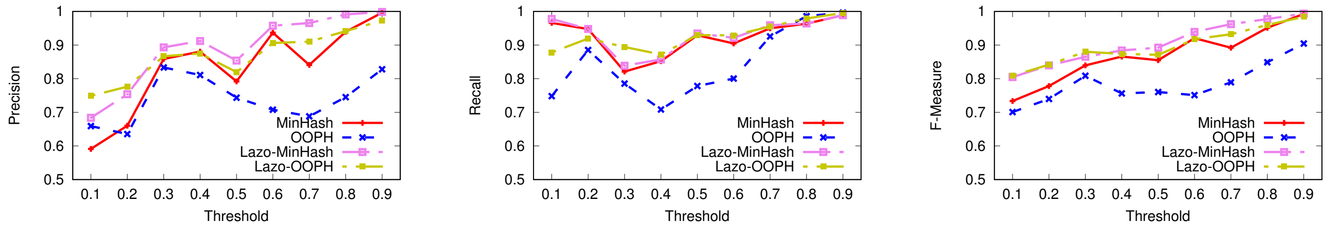

Fig. 6. LAZO similarity estimate improvement when using ECH over MinHash and OOPH baselines on the MassData dataset

The results are shown in Fig. 3, Fig. 4 and Fig. 5, for the MassData, ChEMBL and DWH datasets, respectively. To measure the estimation quality, we obtained ground truth from the three datasets considered. To do this, we compute the exact containment between each pair of columns in each dataset and measure precision, recall and F-measure for different containment thresholds. We compare LSHEnsemble against LAZO when using MinHash as the underlying sketch, LAZO-MinHash and when using OOPH, LAZO-OOPH.

The main result is that LAZO performs better than LSHEnsemble for higher containment thresholds—which are important to detect inclusion dependencies in the presence of dirty and missing data, and hence essential in data discovery scenarios. In particular, for thresholds above 0.6, the recall of LSHEnsemble degrades quickly, with a gap of up to 0.35 for all datasets. The recall deterioration of LSHEnsemble explains the better performance of LAZO in F-measure for MassData and ChEMBL. In the case of DWH, LAZO-MinHash still performs better than LSHEnsemble, but LAZO-OOPH achieves a worse F-measure—although the recall is still much better, there is a drop in precision for this dataset.

Precision and Recall. In the discovery scenario, achieving good recall is more desirable than achieving good precision. Good recall means we find more inclusion dependencies, and precision could be improved by running a post-processing step where the candidates are exactly verified, if the application requires it—we do not run it here because we are interested in the method's baseline performance. The results show that LSHEnsemble's precision is comparable to that of LAZO-MinHash across all similarity thresholds. This comes at the cost of the drop in recall that, as we have discussed, explains the lower F-measure achieved in comparison to LAZO.

When comparing the results between LAZO-MinHash and LAZO-OOPH, we observe that for the MassData and ChEMBL datasets, both sketches achieve similar results. There is however a difference of 0.06 with respect to LSHEnsemble and 0.12 with respect to LAZO-MinHash in the case of the DWH dataset for high thresholds, which occurs because of the lower precision achieved in this case.

In conclusion, LAZO achieves better estimation results for high thresholds, which is important to link datasets with each other. It achieves a similar estimation accuracy for low and mid thresholds than LSHEnsemble and more importantly, unlike

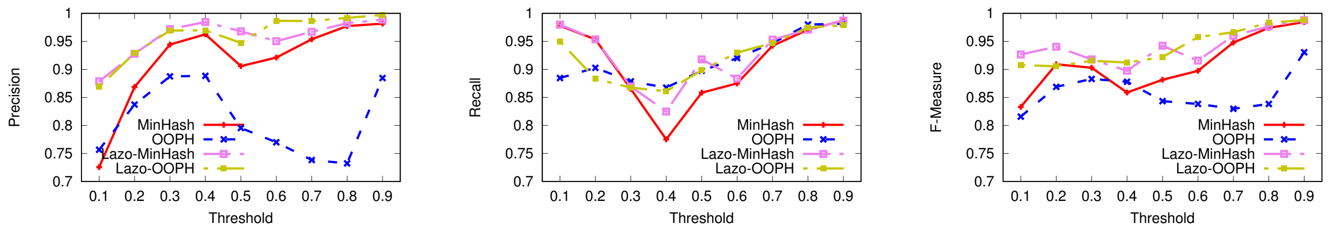
Fig. 7. LAZO similarity estimate improvement when using ECH over MinHash and OOPH baselines on the ChEMBL dataset

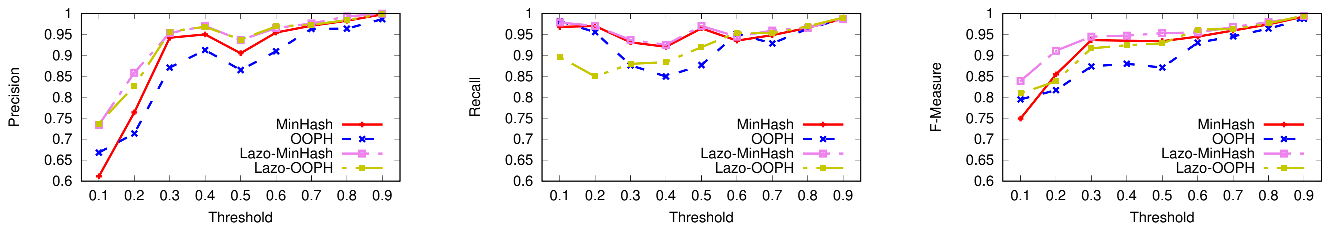
Fig. 8. LAZO similarity estimate improvement when using ECH over MinHash and OOPH baselines on the DWH dataset

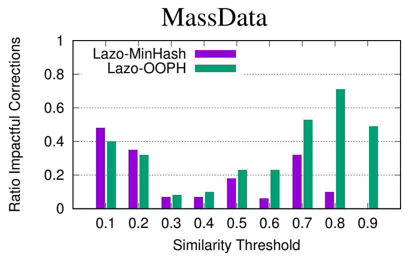
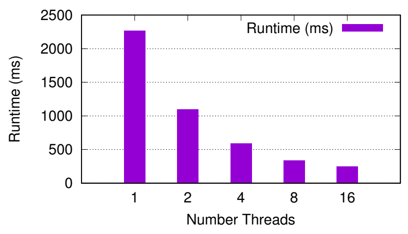
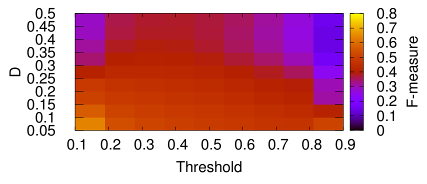
Fig. 9. (Left) Ratio of impactful corrections over number of similar pairs when using ECH; (Center) Lazo Index performance with number of threads; (Right) Sensitivity of parameter D versus similarity threshold.

LSHEnsemble, it does not deteriorate when new data arrives.

B. Improving JS Estimation

In this section we study how much does LAZO improve the similarity estimation. We run MinHash and OOPH as baselines and compare against LAZO-MinHash and LAZO-OOPH, which both use ECH. We measure precision, recall and F-measure, as in the previous experiment and show results for both the MinHash and OOPH sketches without LAZO, which serve as baselines. Both MinHash and OOPH must be configured with the desired number of permutations, K. We show results for $K = 64$ for space reasons, but the results we obtain apply to other values of K as well. In particular, we also improve the estimates for K values of 128, 256 and 512. The results are shown in Fig. 6, Fig. 7 and Fig. 8.

The results show that LAZO improves the estimation quality of both MinHash and OOPH for all thresholds and across all datasets. Specifically, despite the good F-measure of MinHash, LAZO-MinHash still achieves a higher value. The most important result is that LAZO-OOPH performs better or similar than MinHash in all datasets, while being faster to build than MinHash (as shown in Fig. II-A). In other words, with LAZO we improve OOPH's estimation quality to the point of achieving similar estimation accuracy as MinHash, which is more expensive to build as we showed in Section II.

Across all datasets, the recall values for all methods are closer to each other than the precision values. In other words, the reason OOPH performs worse than MinHash is because it produces significantly more false positives (has lower precision) than MinHash. Since LAZO-OOPH bridges the quality gap, the corollary is that LAZO is capable of correcting many of those false positives, improving the precision and therefore the F-measure of OOPH. To fully understand the effect of the ECH we run one additional experiment:

Analyzing effect of ECH. For the three datasets, the improved accuracy is due to ECH; we verified this by turning off the correction and verifying the results. When we disable ECH, the results we obtain match those of MinHash and OOPH, confirming the improvement is due to ECH. However, we wanted to understand better the specific effect of ECH when correcting those values. To do this, we measured how many times LAZO uses ECH to correct an estimate, and, of those, how many times the correction led to removing a false positive, which we call an *impactful correction*. We then measured the ratio of those impactful corrections with respect to the total number of similar pairs for each threshold in the three datasets. The intuition is that for thresholds in which we observed a worse precision, we should observe a larger percentage of impactful corrections to verify that the estimation improvement is in fact due to ECH.

The results in Fig. 9 confirm the intuition; the percentage is higher when the precision gap between the underlying sketch, e.g., MinHash or OOPH, and the corrected estimate, i.e., LAZO-MinHash and LAZO-OOPH respectively, is larger. It also shows how, in general, the number of impactful corrections in the case of LAZO-OOPH is much higher than LAZO-MinHash.

In summary, using LAZO with ECH significantly improves

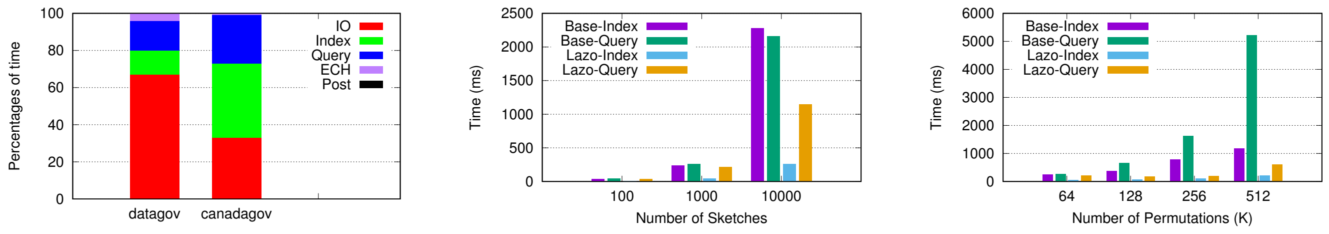

Fig. 10. Breakdown of end-to-end workflow (left); Micro-benchmarks measuring indexing and querying performance of Base and Lazo indexes

the estimation quality of OOPH, bringing its quality on par with that of MinHash, removing the tradeoff between quality and performance presented by OOPH. With LAZO it is possible to build sketches orders of magnitude faster (recall the results in Fig. II-A) without any quality loss.

C. *Is* LAZO *practical?*

LAZO adds two sources of overhead versus a traditional LSH index: i) it needs to obtain the JS score to estimate JC, which requires querying more hash tables (see Section IV for a discussion of how our design minimizes this cost); and ii) the cost of applying the LAZO method, which is constant per pair of columns. In this section, we show that neither of these costs make LAZO impractical, by conducting an experiment with several large data sets: datagov (100K columns, 210 GB) and canadagov (100K columns, 5 GB). The results are shown in Fig. 10 (leftmost figure).

The figure shows the fraction of time spent reading data, building sketches, indexing them in the LAZO index, querying, and applying LAZO. In the two datasets, 80% and 70% of the time is spent in reading data and creating the sketches; these times are unaffected by the LAZO method. Querying time is a small percentage in both datasets, because of specific features of our designed our explored in more detail in the next section. Last, applying the LAZO method incurs a negligible overhead.

Indexing is relatively more expensive in canadagov than in datagov because both datasets contain approximately the same number of columns but canadagov is around 40x smaller than datagov on disk. The runtime was 4 hours for datagov, and 22 minutes for canadagov. A direct runtime comparison with LSHEnsemble is hard because LSHEnsemble is written in Go, but as a reference LSHEnsemble takes double the time to index canadagov [22]. Recall from Section 2, Table I, that an all-pairs comparison on the smaller canadagov did not finish after having run for 24 hours.

With LAZO we obtain all containment and similarity scores for every pair of columns in the dataset. This would not be the case if we run a vanilla OOPH/LSH implementation, that would only return a set of candidates with similarity (and *not* containment) above a pre-specified threshold. We run such configuration nevertheless to get another reference point with respect to the total runtime. The total runtime is 3.35 hours (versus 4 hours for LAZO) for datagov and 12 minutes (versus 22 minutes) for canadagov. LAZO's low overhead means it can be used with large datasets.

D. *Microbenchmarks*

1) Lazo Index vs Baseline Index: Our LAZO implementation uses fewer hash tables than a baseline implementation using a different LSH index for each threshold. We explained the precise number required by each alternative in Section IV-C. Here, we demonstrate the performance difference on indexing and querying with a microbenchmark.

We measure the effect of the number of sketches as well as the number of permutations on both implementations, using $D = 0.05$. We refer to the configurations as Base-Index, Base-Query and Lazo-Index, Lazo-Query, respectively. In the first experiment (Fig. 10 center) we vary the number of sketches we insert and then we measure the time to insert them all, as well as the time to query the index with all the previously inserted sketches. The results show that the times in all cases grow linearly, which makes sense because we are using LSH. In the case of the Base configuration, however, times are higher than Lazo when querying and much higher on insertion, with gaps of up to 8x.

In the second experiment, (Fig. 10 right), we fix the number of sketches to 1000 and vary the number of permutations from 64 to 512. This has an impact in both indexes because it changes the number of necessary hash tables. In this case, both querying and indexing times grow faster in the case of the Base configuration than in the case of Lazo, with gaps of up to an order of magnitude for high values of K.

2) Sensitivity to D: When creating the LAZO index, we must choose a parameter, D, which determines the granularity of the Jaccard similarity estimate as explained in Section IV-B2. This granularity, we discussed, has an impact on both JC and JS estimation quality. Intuitively, we expect smaller values of D to yield better estimates at the cost of consuming more storage.

In practice, we use a small D because storage has not been a limiting factor in our setting due to our LAZO implementation, which reduces the necessary number of hash tables to maintain. Nevertheless, we study here how to save storage without compromising accuracy.

The results for the experiment are shown in Fig. 9 (Right) for the MASSDATA dataset (results for DWH and ChEMBL are similar and we do not include them for space reasons). We show how the F-measure (for the Jaccard similarity results) changes based on the similarity threshold and on the value of D. As expected, the smaller D, the better the estimate quality. This is true for all thresholds. As D grows, the estimate quality reduces faster for thresholds that are further from 0.5. This makes sense, the further the threshold is from 0.5 the more accurate the estimate must be to detect the right pairs. The more interesting outcome of this experiment is that for many thresholds, the storage needs can be reduced up to 3x—by increasing D—without having a high impact on the accuracy.

3) Impact of exact cardinality estimates: In our implementation we use HyperLogLog to estimate the cardinality of the columns. HyperLogLog estimates the cardinality in one-pass only, and we share the hashed values we use for the sketches, so the runtime overhead it introduces is negligible. An open question, however, is whether the estimation error it introduces is significant or not for the JS and JC estimation. Our experimental section has demonstrated that we achieve good accuracy for containment and similarity with LAZO. We want to confirm the low impact on accuracy of using HyperLogLog. We computed the exact cardinalities of the sets for all the datasets we consider in this evaluation, use the exact cardinalities to compute similarities, and compare the results obtained with exact cardinalities versus the estimated.

The maximum error in the F-measure for any threshold and any dataset was of 0.07, but the average is of 0.01 as well as the median and the 75 percentile. The 25 percentile was 0. Given the small magnitude of error, we conclude that the assumption is useful to use LAZO in practice.

VI. Related Work

Estimating containment. Two existing methods to estimate Jaccard containment are LSHEnsemble [22] and Asymmetric Minwise Hashing (ALSH) [18]. LSHEnsemble achieves better quality results as shown in [22], this is true for sets with skewed sizes, which is the case of databases. Unlike LSH-Ensemble, which requires knowing the cardinality of all the sets before indexing, LAZO can be updated on the fly, as new sets become available, without accuracy loss, and it achieves better recall, which is important as discussed to find more opportunities to combine datasets.

BML [13] estimates containment using Hyperloglog sketches, but they still need to enumerate all n^2 pairs of sketches and then run a maximum-likelihood inspired algorithm. We cannot directly compare accuracy and performance because their code is not available. However, we can use numbers from their experimental section to gain insight on the different performance profiles of BML and LAZO. On a dataset with 4300 columns, their experiments show that their approach takes 600 seconds to obtain all the containment estimates using 6 cores. Our approach (as shown in Section 2), on a dataset with 5500 columns, takes around 80 seconds for the whole pipeline and around 30 seconds for creating the sketches and querying, using only 1 core, and it computes both similarity and containment scores for all candidates, instead of only containment as BML.

Theoretical results. Previous theoretical work has pointed out that the set intersection can be estimated by multiplying the estimated generalized Jaccard similarity by an estimation for the size of the union [15]. That work points out that as a consequence, results on similarity estimation imply results on containment. In this paper, we make this concrete by obtaining the estimate $\hat{\alpha}$ and using it to estimate both measures, and we do that without an estimation of the union set size, only by using individual set cardinalities.

Exact methods. Exact methods of computing jaccard similarity [4] and containment [6], [19] find the exact relationships between all-pairs of columns. For discovery scenarios with large number of columns exact methods are infeasible as shown in our experiments, hence the motivation for Lazo.

VII. Conclusions

LAZO estimates containment and similarity simultaneously. Its accuracy is good even for high containment thresholds, which means it can be used to discover inclusion dependencies. With ECH, we match MinHash accuracy with the faster-to-compute OOPH. The overheads that LAZO introduces are negligible, as shown in our large scale experiment with large open governmental data. In conclusion, LAZO is a practical method to use in discovery scenarios.

Acknowledgements. This work was supported by the MIT DSAIL project.

References

[1] Data.gov. https://www.data.gov, 2018. [Online; accessed 18-July-2018].
[2] T. D. Ahle, R. Pagh, I. Razenshteyn, and Francesco. On the Complexity of Inner Product Similarity Join. In *PODS*, 2016.
[3] A. Andoni, P. Indyk, et al. Practical and Optimal LSH for Angular Distance. In *NIPS*, 2015.
[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
[5] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *STOC*, 2002.
[6] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, 2018.
[7] R. C. Fernandez et al. Aurum: A data discovery system. 2018.
[8] A. Halevy, F. Korn, et al. Goods: Organizing Google's Datasets. In *SIGMOD*, 2016.
[9] S. Heule, M. Nunkesser, et al. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *EDBT*, 2013.
[10] J. D. U. Jure Leskovec, Anand Rajaraman. Mining Massive Datasets, 2018. [Online; accessed 18-July-2018].
[11] P. Li, K. W. Church, et al. Conditional Random Sampling: A Sketch-based Sampling Technique for Sparse Data. In *NIPS*, 2006.
[12] P. Li, A. B. Owen, et al. One Permutation Hashing. In *NIPS*, 2012.
[13] A. Nazi, B. Ding, V. R. Narasayya, and S. Chaudhuri. Efficient Estimation of Inclusion Coefficient using HyperLogLog Sketches. 2018.
[14] B. Neyshabur and N. Srebro. On Symmetric and Asymmetric LSHs for Inner Product Search. In *ICML*, 2015.
[15] R. Pagh, M. Stöckel, and D. P. Woodruff. Is Min-wise Hashing Optimal for Summarizing Set Intersection? In *PODS*, 2014.
[16] A. Shrivastava. Optimal densification for fast and accurate minwise hashing. In *ICML*, 2017.
[17] A. Shrivastava and P. Li. Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search. In *ICML*, 2014.
[18] A. Shrivastava and P. Li. Asymmetric Minwise Hashing for Indexing Binary Inner Products and Set Containment. In *WWW*, 2015.
[19] F. Tschirschnitz, T. Papenbrock, et al. Detecting Inclusion Dependencies on Very Many Tables. *TODS*, 2017.
[20] E. L. Willighagen, A. Waagmeester, et al. The chembl database as linked open data. *Journal of Cheminformatics*, 2013.
[21] E. Zhu, Y. He, et al. Auto-join: Joining Tables by Leveraging Transformations. In *VLDB*, 2017.
[22] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller. LSH Ensemble: Internet-scale Domain Search. *VLDB*, 2016.