# Top-K Entity Resolution with Adaptive Locality-Sensitive Hashing

Vasilis Verroios
Stanford University
verroios@stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

## ABSTRACT

Given a set of records, entity resolution algorithms find all the records referring to each entity. In this paper, we study the problem of top-$k$ entity resolution: finding all the records referring to the $k$ largest (in terms of records) entities. Top-$k$ entity resolution is driven by many modern applications that operate over just the few most popular entities in a dataset. We propose a novel approach, based on locality-sensitive hashing, that can very rapidly and accurately process massive datasets. Our key insight is to adaptively decide how much processing each record requires to ascertain if it refers to a top-$k$ entity or not: the less likely a record is to refer to a top-$k$ entity, the less it is processed. The heavily reduced amount of processing for the vast majority of records that do *not* refer to top-$k$ entities, leads to significant speedups. Our experiments with images, web articles, and scientific publications show a 2x to 25x speedup compared to the traditional approach for high-dimensional data.

## 1. INTRODUCTION

Given a set of records, the objective in entity resolution (ER) is to find clusters of records such that each cluster collects all the records referring to the same entity. For example, if the records are restaurant entries on Google Maps, the objective of ER is to find all entries referring to the same restaurant, for every restaurant.

In many applications the *size* of a resolved entity (i.e., the number of records) reflects the *importance* or *populatity* of the entity. For example, if the records are bug reports and the entities are code bugs, the larger the entity, the more users have encountered the bug, and the more important it is to fix the bug. If we only have resources to fix say three bugs this week, then we only need to identify the top three bugs, i.e., we do not need to resolve all entities (which may be very time consuming), just the three largest. It also so happens that in most of these applications where importance matters, entity sizes follow a Zipfian distribution, so the top few entities are way more important or popular than the rest. As we will see, this fact makes it easier to find the top-$k$ entities without computing all entities. We next illustrate two additional applications where one typically only needs top-$k$ entities. Incidentally, two of the datasets we use for our experiments are from these applications.

*Proceedings of the VLDB Endowment,* Vol. 10, No. 7
Copyright 2017 VLDB Endowment 2150-8097/17/03.

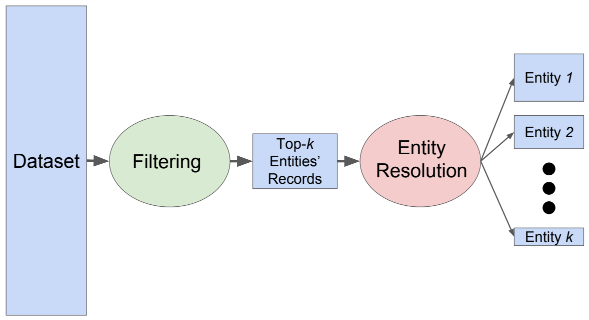


**Figure 1: Filtering stage in the overall workflow.**

First, consider news articles: in many cases, the same story (or a large portion of it) is copied/reproduced in different articles. In this case a resolved entity represents a single story or news event, and its records are the articles that refer to that story. The larger an entity, the more web sites or newspapers decided to present that story, so the more popular (arguably important) that story is. If we are preparing a news summary for readers with limited time, it makes sense to include the top-$k$ stories. (Of course, the summary may also include other hand-picked stories.) Again, for the summary of the popular stories one does not need to resolve all entities. Note that computing all entities in this application can be prohibitive: there can be hundreds of thousands of articles, resolution is quadratic in the number of articles, and we want to produce our summary in a very timely manner.

Second, consider viral images in social media: images that are being copied (possibly transformed) and shared, sometimes without a direct connection to the original post. Large entities represent images that have been shared a lot and can be much more interesting than other images for a number of reasons, e.g., copyright issues or they indicate what images are being shared the most. In all those cases, finding the top-$k$ entities can be sufficient. For instance, serious copyright violations may require an expert's attention and intervention and since experts' time is limited, it can be better allocated in content that is being extensively shared (i.e., one of the top-$k$ entities).

Overall, we can find a motivating example for any application where the user experience drastically improves by awareness of the popular entities. Other interesting examples include: popular questions in search engine query logs, suspicious individuals appearing very often in specific areas of an airport, and viral videos in video streaming engines.

The naive approach for finding the largest entities is to first apply an ER method on the whole dataset to find all entities, and then output only the largest entities. However, for applications where finding the largest entities is important, we can expect the datasets to be large and change rapidly over time. Hence, it is quite wasteful and inefficient to apply ER on the entire dataset. For instance, a traditional ER algorithm may require the computation of pairwise similarities for every two records in the dataset. (In large datasets, the cost of computing the similarity of every two records is prohibitive, e.g., a dataset of 100 thousand records would require the computation of almost 5 billion pairwise similarities.) Note also that computing each pairwise similarity may be an expensive operation for many applications, e.g., records containing images.

In this paper, we focus on a lightweight preprocessing stage that receives the whole dataset as an input and tries to filter out all the records that do not belong to the $k$ largest entities; where $k$ is an input parameter. The output of this filtering stage is then fed to an ER algorithm that produces one cluster of records for each of the top-$k$ entities and, possibly, aggregates the records in each cluster to produce a summary for each entity (Figure 1). The filtering stage output may contain a few records that do not belong to the $k$ largest entities, or a few records from the $k$ largest entities may not be part of the output. Nevertheless, if the number of such errors is limited, we expect that the subsequent ER algorithm can recover and produce (almost) the same final outcome as if the filtering stage was not present. The purpose of the filtering stage is to enable the efficient processing of large datasets, by having a linear cost to the number of records in the dataset. Since the filtering stage output is expected to be much smaller than the whole dataset, the ER algorithm can afford a quadratic (or even higher) cost to the input size, or even involve human curation.

Various different aspects of entity resolution have been studied over the years [16, 37, 35, 18, 9]. The most related topic to the problem studied here, is the one of blocking [24, 5, 7, 25, 17, 11, 12]: the goal of blocking is to split a dataset into blocks, such that the records of each entity (ideally) do *not* split across two or more different blocks. Moreover, the cost of blocking must be low, typically linear to the number of records in the dataset. Nevertheless, blocking mechanisms are designed for entity resolution over the entire dataset. Especially for high-dimensional data (e.g., web articles, images, videos, audio), there is a significant computational cost to process each record in the dataset, to decide in which blocks to place the record. In this paper, we argue that we can find the small portion of the dataset referring to the top-$k$ entities, with a very low cost for each record in the rest of the dataset, that does *not* refer to a top-$k$ entity.

Our approach consists of a linear-cost filtering stage algorithm that uses Locality-Sensitive Hashing (LSH) [19] in an adaptive way. LSH essentially enables efficient blocking for high-dimensional data. In the most common setting, LSH uses a set of hash tables and applies a large number of hash functions on each record of the dataset. Every two records that LSH places in the same bucket, in one of the tables, are considered "similar", i.e., referring to the same entity for ER. Nevertheless, LSH requires a large number of hash functions to be applied on each record. That is, while the cost of applying LSH is linear to the size of the input, the cost of processing each record is high. The *Adaptive LSH* approach we propose is capable of processing the vast majority of records in the dataset with a very low cost per record, showing a speedup of up to 25x, compared to traditional LSH blocking.

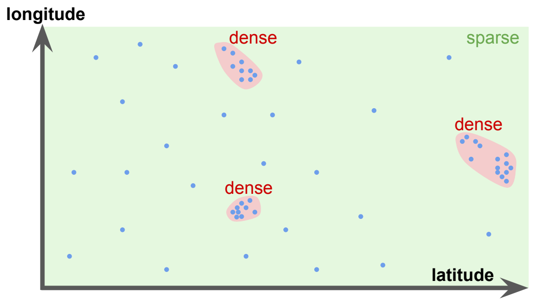


**Figure 2: Key insight in Adaptive LSH.**

Figure 2 illustrates our approach's key insight using a simple cartoon sketch. The points refer to restaurant records, where for each record only the location (longitude/latitude) is known. The records of the top-$k$ entities lie on the "dense" areas (i.e., areas with many records) in this space, while the records that can be filtered out lie on "sparse" areas. The key insight is that only a small number of LSH functions need to be applied to the records that lie on "sparse" areas and the full set of LSH functions needs to be applied only for the records that lie on "dense" areas. Adaptive LSH starts by applying a small number of LSH functions on each record in the dataset. It then detects which areas are sparse and which are dense and continues by applying more LSH functions only to the records in the dense areas, until it converges to the records referring to the top-$k$ entities. Thus, Adaptive LSH can very efficiently filter out records that are highly unlikely to be records referring to top-$k$ entities, and only requires a higher cost for records that refer (or are quite likely to refer) to the top-$k$ entities.

The rest of the paper is organized as follows: we start with an overview of Adaptive LSH, in Section 2, in Section 3 we present the LSH clustering functions, a key component of our approach, in Sections 4 and 5 we discuss the algorithm and details of Adaptive LSH, and in Sections 6 and 7 we discuss our experimental results with datasets of web articles, images, and scientific publications.

## 2. APPROACH OVERVIEW

We start with the problem definition and an overview of the Adaptive LSH approach through a discussion of its three main concepts in Sections 2.2, 2.3, and 2.4.

### 2.1 Problem Definition

Let us denote the set of records in the dataset by

$$R = \{r_1, \dots, r_{|R|}\}$$

Each record $r_i$ refers to a single entity. In the ground truth clustering

$$\mathcal{C}^* = \{C_1^*, \dots, C_{|\mathcal{C}^*|}^*\}$$

cluster $C_j^*$ contains all the records referring to entity $j$.

Assume a descending order on cluster size in our notation, i.e., $|C_i^*| \geq |C_j^*|$ for $i < j$. The objective of the filtering stage, in Figure 1, is to, very efficiently, find the set of records $\mathcal{O}^*$ that belong to the $k$ largest clusters in $\mathcal{C}^*$:

$$\mathcal{O}^* = \{r_j : r_j \in C_i^*, i \leq k\}$$

Each method we study in this paper outputs a set of records $\mathcal{O}$, which we compare against the ground truth set $\mathcal{O}^*$. In particular, we measure the:

$$precision = \frac{|\mathcal{O} \cap \mathcal{O}^*|}{|\mathcal{O}|}, \quad recall = \frac{|\mathcal{O} \cap \mathcal{O}^*|}{|\mathcal{O}^*|}$$

and

$$F1\ \ score = \frac{2 * precision * recall}{precision + recall}$$

## 2.2 Clustering Functions

To achieve a high precision and recall with a very low execution time, Adaptive LSH relies on a sequence of $L$ clustering functions $(g_j : S \to \{C_i\})_{j=1}^{L}$. Each function $g_j$ receives as input a set of records $S \subseteq R$ and clusters those records into a set of non-overlapping clusters $\{C_i\}$. The input set $S$ can be the whole dataset $R$, or a single cluster produced by a previous function in the sequence.

The functions in the sequence are probabilistic and have the following four properties:

1. *conservative evaluation*: the functions attempt to cluster the records of any ground truth cluster $C_i^*$ under the same cluster in the output. That is, a cluster in the output of any function $g_j$ may contain two or more ground truth clusters, but a ground truth cluster should very rarely split into two (or more) of the output clusters.
2. *increasing accuracy*: the further a function is in the sequence, the higher the chances of outputting the ground truth clustering $\mathcal{C}^*$ when applied on $R$; or the ground truth clusters in a subset $S$, for any subset $S$.
3. *increasing cost*: the further a function is in the sequence, the higher the cost of applying the function on any subset of records $S$.
4. *incremental computation*: the computation of the functions in the sequence can be performed incrementally. That is, the computation required by a function $g_i$ consists of the computation required by $g_{i-1}$ plus some additional computation.

## 2.3 Sequential Function Application

Our approach starts by applying the most lightweight function in the sequence, $g_1$, on the whole dataset $R$, and continues by applying subsequent functions on the "most-promising" (for being a top-$k$ entity) clusters.

Let us illustrate the concept of sequential function application via the example in Figure 3. The most "lightweight" function $g_1$ is applied on the whole dataset $R$ and splits it into the first round clusters (three in this figure) $C_1^{(1)}, C_2^{(1)}, C_3^{(1)}$; the superscript denotes the round. In the second round, the next function in the sequence, $g_2$, is applied on one of the clusters from the first round; $C_1^{(1)}$ in this example. In each round, our approach selects the largest, in terms of number of records, cluster that is *not* an outcome of the last function in the sequence. The intuition for this choice is that a large cluster has to be processed sooner or later, to find out if it belongs to the top-$k$ or not. We prove that the largest-cluster selection rule is actually optimal, in Section 4. Function $g_2$ splits $C_1^{(1)}$ into two clusters $C_1^{(2)}, C_2^{(2)}$. The other two clusters $C_2^{(1)}, C_3^{(1)}$ from the first round, are also added, unchanged, to the list of clusters after Round 2. In the third round, cluster $C_3^{(2)}$ is selected. Since cluster $C_3^{(2)}$ is the outcome of a $g_1$ function, the next function to be applied on $C_3^{(2)}$ is $g_2$.

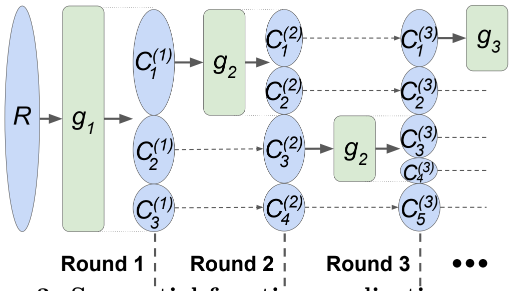


**Figure 3: Sequential function application example.**

The sequential function application stops when the $k$ largest clusters, at the end of a round, are an outcome of the last function in the sequence: the union of records in the $k$ clusters are returned as the output of the filtering stage. In this case, based on Properties 1 and 2, each of the $k$ clusters is very likely to refer to exactly one of the ground truth clusters in $\mathcal{C}^*$. In addition, based on Property 1, all other ground truth clusters are very likely to be smaller than the $k$ clusters and, thus, it is "safe" to conclude that the $k$ clusters after the last round are the top-$k$ clusters in $\mathcal{C}^*$. Of course, our approach may introduce errors and the output may not be identical to the ground truth output $\mathcal{O}^*$, as the objective is a fast filtering process that significantly reduces the size of the initial dataset. Nonetheless, even when those errors are non-negligible, we can trade precision for recall and control the output's quality with a small cost in performance, as we discuss in the experimental section.

When the sequential function application terminates, we expect that for the vast majority of records only the first few functions in the sequence will have been applied. Going back to the discussion for Figure 2, all the records lying on "sparse" areas will have a few functions applied on, and the full sequence of functions will only be applied on the records lying on "dense" areas. Hence, the amount of processing applied on most records will be considerably lower than the amount of processing applied on the records on the "dense" areas that are likely to belong to the top-$k$ entities.

Note also that because of Property 4, the sequential function application is performed incrementally. For instance, part of the computation required for applying $g_2$ on $C_1^{(1)}$, is already performed by $g_1$ and does not need to be repeated.

## 2.4 Locality-Sensitive Hashing

The third key concept in our approach is using LSH [19] as the main component of the clustering function sequence. We give an overview of LSH and discuss how to build clustering functions with Properties 1 to 4 in the next section.

## 3. CLUSTERING FUNCTIONS

In this section, we present the clustering functions used in our approach. Our goal is to provide an overview without going into the technical details that involve LSH. All details can be found in Appendix A.

The clustering functions rely on distance metrics: the smaller the distance between two records, the more likely the two records are to refer to the same entity. To illustrate, consider the following example:

`EXAMPLE 1` *Consider a set of records where each record consists of a single photo of a person, processed so that the distance between eyes and the distance of nose tip to mouth are computed. The two distances form a two dimensional*

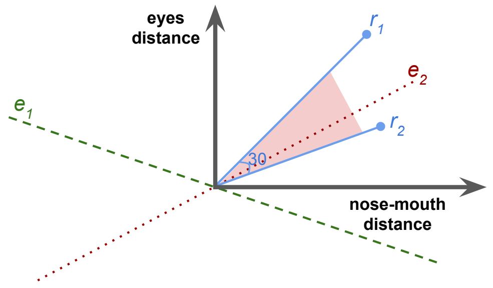


**Figure 4: Random hyperplanes example.**

*vector. Consider the cosine distance, i.e., the angle between the vectors of two records. If two photos show the same person we can expect the ratio of eyes distance to nose-mouth distance to be roughly the same in the two photos and, hence, the angle between the two respective vectors to be small.*

In addition, the clustering functions assume a distance threshold $d_{thr}$: if the distance between two records is less than $d_{thr}$, the two records are considered a *match*.

Clearly, real datasets consist of records with multiple fields. Therefore, there is a separate distance metric for each field and it may be more effective to use multiple distance thresholds. For example, consider a set of records, where each one consists of a person photo and fingerprints. In this case, there could be two thresholds and two records would be considered a match if the photos' distance was lower than the first threshold, *or* if the fingerprints' distance was lower than the second threshold. To keep the discussion in the next sections concise, we focus on the simplest case of a single field/threshold. In Appendix C, we discuss how to extend all the mechanisms in our approach for the general case, where each record consists of many fields.

Two records can also be considered a match, via transitivity. That is, if two records $a$ and $b$ are within the distance threshold, and $b$ is also within the threshold with a record $c$, records $a$ and $c$ are also considered a match.

To find the matches without having to compute the $\binom{|R|}{2}$ distances, the clustering functions use LSH. LSH is based on hash functions that are applied individually on each record. The smaller the distance between two records, the more likely a hash function is to give the same value when applied to each of the two records. One example of such hash functions is the random hyperplanes for the cosine distance:

EXAMPLE 2 *Consider again the dataset of photos, in Example 1. Consider two random hyperplanes (lines) through the origin, in the two dimensional space representing the photos. Figure 4 depicts two such lines, $e_1$ and $e_2$. In addition, consider the vectors, $r_1$ and $r_2$, for two photos in the dataset. The cosine distance between $r_1$ and $r_2$ is 30 degrees. Note that the difference between $e_1$ and $e_2$, is that $r_1$ and $r_2$ are on the same side for line $e_1$, but for different sides for line $e_2$. The hash function in this case is simple a random line and the hash value is 1 or $-1$, depending on which side of the line the input record lies. In general, the smaller the angle between two records, the higher the likelihood of selecting a random line where the two records lie on the same side of the selected line: note that the likelihood for records $r_1$ and $r_2$ is $1-\frac{30}{180}$, while, in general, the likelihood is $1-\frac{\theta}{180}$, if $\theta$ is the angle between the two records.*

LSH applies a large number of such hash functions on each record. The outcome of those functions is used to build hash tables: the index of each bucket, in each table, is formed by concatenating the outcome from a number of hash functions. The following example illustrates the tables built by LSH:

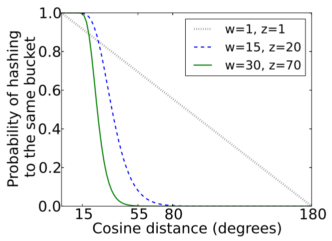


**Figure 5: Probability of hashing to the same bucket**

EXAMPLE 3 *Consider again the hash functions and dataset, from Example 2. Assume LSH uses two hash tables: for each table, three hash functions (random lines through the origin) are selected. Since the outcome of each function is binary, there are $2^3 = 8$ buckets in each table. Now consider the event of two records hashing to the same bucket in at least one of the two tables. If the angle between the two records is $\theta$, the probability of this event is: $1-\left(1-(1-\frac{\theta}{180})^3\right)^2$.*

The number of hash tables $z$, and the number of hash functions per table $w$, are selected so that: (a) if two records are within the distance threshold $d_{thr}$, the probability of the two records hashing to the same bucket in at least one table, must be very close to 1.0 and (b) if the distance between two records is greater than $d_{thr}$, the probability of the two records hashing to the same bucket in at least one table, must be as close to 0.0 as possible (details in Section 5.1).

Figure 5 illustrates the probability of two records hashing to the same bucket, in at least one table for the setting of Examples 1 to 3. The x-axis shows the cosine distance and the y-axis the probability for three $w, z$ value pairs.

Consider a threshold $d_{thr}$ of 15 degrees. As Figure 5 shows, the more hash functions used, the more sharply the probability of hashing to the same bucket drops, after the threshold. On the other hand, applying more functions on each record, incurs a higher cost.

Each clustering function in the sequence relies on an LSH scheme with $z$ tables and $w$ hash functions per table: the further a function is in the sequence, the larger the values of $w$ and $z$ are. We call these clustering functions *transitive hashing* functions:

DEFINITION 1 (Transitive Hashing) *A transitive hashing function $H$, based on an LSH scheme with $z$ tables and $w$ hash functions per table, receives as input a set of records $S$, and splits $S$ into a set of clusters $\{C_i\}$ as follows: consider the graph $G = (S, E)$, where $(r_1, r_2) \in E$ iff records $r_1$ and $r_2$ hash to same bucket of at least one of the $z$ hash tables. Function $H$ outputs one cluster $C_i$ for each of the connected components of $G$.*

In Appendix B, we present an efficient implementation for transitive hashing functions.

Note how transitive hashing functions satisfy the three properties stated in Section 2.2:

1. *conservative evaluation*: even when $w$ and $z$ are small, pairs of records within the threshold are very likely to be placed in the same bucket, in at least one of the tables; based on point (a) above.
2. *increasing accuracy*: the further a function is in the sequence, the larger the values of $w$ and $z$ are, and the less the false matches are.
3. *increasing cost*: the further a function in the sequence, the larger the values of $w$ and $z$, and the higher the cost of applying that function on any subset of records $S$.
4. *incremental computation*: function computation is performed incrementally, as the hash values from previous sequence functions are re-used by functions that follow.

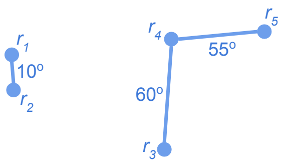


**Figure 6: Transitive Hashing example.**

We conclude this section with an example for transitive hashing functions:

EXAMPLE 4 *Consider the set of records $S = \{r_1, r_2, r_3, r_4, r_5\}$, and two transitive hashing functions: $H_1$ with $z = 20$ tables, each using $w = 15$ hash functions, and $H_2$ with $z = 70$ tables, each using $w = 30$ hash functions. Figure 6 depicts the cosine distance between each two records in S (no edge for pairs with a distance greater than 80 degrees). With high probability $H_2$ outputs $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}, \{r_5\}\}$: as the plot in Figure 5 points out, the likelihood of two records hashing to the same bucket when the distance between them is greater than 55 degrees, for the $w = 30, z = 70$ curve, is very low. Now assume that for $H_1$, records $r_3$ and $r_4$ hash to the same bucket in one of the 20 hash tables. (As the plot in Figure 5 points out, there is a good chance of two records hashing to the same bucket when the distance between them is 60 degrees, for the $w = 15, z = 20$ curve.) Moreover, assume records $r_4$ and $r_5$ hash to the same bucket in one of the 20 hash tables, as well. Then, with high probability, $H_1$ outputs $\{\{r_1, r_2\}, \{r_3, r_4, r_5\}\}$.*

## 4. ADAPTIVE LSH

### 4.1 Algorithm

**INPUT**
parameter $k$, records $R$, distance metric $d$, threshold $d_{thr}$, sequence of transitive hashing functions $H_1, \ldots, H_L$
**OUTPUT**
$k$ largest connected components in graph $G = (R, E)$, where $(r_1, r_2) \in E$ iff $d(r_1, r_2) \leq d_{thr}$.

The sequence of clustering functions used by Adaptive LSH is a sequence of transitive hashing functions $H_1, \ldots, H_L$, where function $H_i$ is based on an LSH scheme with $z_i$ tables and $w_i$ hash functions per table, where $w_i \leq w_{i+1}, z_i \leq z_{i+1}$, $i \in [1, L)$. Sequence $H_1, \ldots, H_L$ is given as input (two integers $w_i$, $z_i$, for each function $H_i$) and in Section 5 we discuss how to select the functions in this sequence.

As discussed in Section 2.3, Adaptive LSH selects the largest cluster to process in each round, regardless of which function each cluster is an outcome of. In Section 4.2, we prove that selecting the largest cluster in each round is optimal under mild assumptions.

Besides the sequence of transitive hashing functions, Adaptive LSH also uses an additional function that computes the matches in a cluster of records given as input, using the exact record pair distances:

DEFINITION 2 (Pairwise Computation) *The pairwise computation function P receives as input a set of records S and splits S into a set of clusters $\{C_i\}$ as follows: consider the graph $G = (S, E)$, where $(r_1, r_2) \in E$ iff $d(r_1, r_2) \leq d_{thr}$. Function P computes the distances between pairs of records and outputs one cluster $C_i$ for each of the connected components of graph G.*

When a cluster $C$ is the outcome of a function $H_i$ and the application of the next function in the sequence, $H_{i+1}$, has a cost greater than the cost of applying function $P$ on $C$, Adaptive LSH applies $P$ instead of $H_{i+1}$ on cluster $C$. This is usually the case when cluster $C$ is small and computing the distances for, potentially, all pairs in $C$, is preferable to computing a large number of hashes for each record in $C$. Thus, the termination rule for Adaptive LSH (as discussed in Section 2.3) is extended as follows: terminate once the $k$ largest clusters, in the list of clusters at the end of a round, are an outcome of an $H_L$ or $P$ function.

**Algorithm 1** *Adaptive LSH*

```
Input: $R$ - Set of all records
Input: $k$ - top-k parameter
Input: $d$ - distance metric
Input: $d_{thr}$ - distance threshold
Input: $H_1, \ldots, H_L$ - sequence of functions
Input: $cost_P, cost_1, \ldots, cost_L$ - cost model parameters
Output: top-k entities
 1: $\{C_i^{(1)}\} := H_1(R)$
 2: for each Round $j$ do
 3:   $C$ := largest cluster in $\{C_i^{(j)}\}$; $\{C_i^{(j)}\} := \{C_i^{(j)}\} \setminus C$
 4:   $t$ := sequence number of function $H_t$ that produced $C$
 5:   if $(cost_{t+1} - cost_t) * |C| \geq cost_P * \binom{|C|}{2}$ then
 6:     $\{C_i\} := P(C)$
 7:   else
 8:     $\{C_i\} := H_{t+1}(C)$
 9:   end if
10:   $\{C_i^{(j+1)}\} := \{C_i\} \cup \{C_i^{(j)}\}$
11:   if largest $k$ clusters in $\{C_i^{(j+1)}\}$ are all an outcome of function $H_L$ or $P$ then
12:     return largest $k$ clusters in $\{C_i^{(j+1)}\}$
13:   end if
14: end for
```

To decide when to apply the pairwise computation function $P$, the algorithm relies on a simple cost model:

DEFINITION 3 (Cost Model) *The cost of applying function P, on a set of records S, is $cost_P * \binom{|S|}{2}$. The cost of applying function $H_i$ in the sequence, on a set S, is $cost_i * |S|$. Moreover, the cost of applying function $H_i$ on a record r, when function $H_j$, $j < i$, is already applied on r, is $cost_i - cost_j$. After the completion of the algorithm, the overall cost is $\sum_{i=0}^{L} n_i * cost_i + n_P * cost_P$, when function $H_i$ is the last sequence function applied on $n_i$ records and $n_P$ is the overall number of similarities computed by function P.*

In Appendix E.2, we run experiments to evaluate how sensitive adaptive LSH is to the cost model: we manually add noise to the model's cost estimations and measure how the execution time changes. Algorithm 1 gives the detailed description of the process outlined in Section 2.3.

### 4.2 Largest-First Optimality

In this section we prove optimality for the Largest-First strategy of Algorithm 1 under two assumptions (explained after the proof).

THEOREM 1 *Consider the family of algorithms that:*

1. *do not "jump ahead" to function P, i.e., if a cluster C is an outcome of a function $H_i$, the algorithm can only apply function P on C, when $(cost_{i+1} - cost_i) * |C| \geq cost_P * \binom{|C|}{2}$ (Line 5 on Algorithm 1).*
2. *do not "terminate early", i.e., terminate only when the k largest clusters are an outcome of either an $H_L$ or P function.*

*Algorithm 1 gives the minimum overall cost compared to any other algorithm of this family.*

PROOF: Here we provide an overview for the proof; the complete proof can be found in Appendix D.1. We use the notion of an *execution instance*: in an execution instance, the outcome of applying a function $H_i$ or $P$ on a set of records $S$, is the same across all algorithms. In other words, all algorithms would observe the exact same clusters if they would select the same cluster to process in each step. We prove the theorem by contradiction: we assume that another algorithm gives a lower overall cost than Algorithm 1 for a given execution instance, we consider all cases where such an algorithm would have a lower overall cost, and we prove that, in all those cases, one of the two conditions defining the family of functions must be violated.

$\square$

The rationale for the two conditions of Theorem 1 is simple. Condition 2 states that we consider algorithms that try to minimize the errors by only returning clusters "thoroughly checked" by an $H_L$ or $P$ function. For Condition 1, accurately predicting if "jumping ahead" to function $P$ will eventually give a lower cost for a cluster $C$, is quite challenging, may not necessarily show significant gains (even if predicting right), and requires a significant overhead of keeping estimates for the structure (e.g., subclusters) of $C$, as we discuss in Appendix D.2.

Besides the guarantee of the lowest cost for finding the $k$ largest clusters, the Largest-First strategy provides one more guarantee, for an incremental mode of Algorithm 1. Let us first describe the incremental mode. When a user wants to monitor the results during the filtering process, the incremental mode can output intermediate results before filtering is competed. To operate in incremental mode Algorithm 1 is modified. In Line 11, we use a different condition: if the largest cluster in the set of clusters $\{C_i^{(j+1)}\}$, is an outcome of either an $H_L$ or $P$ function, we output this cluster, remove it from $\{C_i^{(j+1)}\}$, and the loop of Lines 2 to 14 continues until $k$ clusters are returned.

The second guarantee states that the incremental mode of Algorithm 1, applied with input $k$, minimizes the cost of finding the $k'$ largest clusters, for any $k' < k$. For instance, if a user is monitoring the results during filtering for an input $k = 2$, the Largest-First strategy guarantees that the cost (or time) for finding the largest cluster is minimized; another strategy could minimize the cost of finding the second largest cluster before finding the largest cluster and still be optimal with respect to the top-2 clusters. This can be a very useful property assuming that the higher a cluster is in the top-$k$ list, the higher the user's interest is. The second guarantee of the Largest-First strategy is formally stated in Theorem 2.

THEOREM 2 *For an input $k$, Algorithm 1 reaches to a state where the $k'$ largest clusters are an outcome of either an $H_L$ or $P$ function, for any $k' < k$, with a lower cost compared to any other algorithm in the family defined in Theorem 1.*

PROOF: Again, we provide just an overview for the proof here; the complete proof is included in Appendix D.1. For a $k' < k$, we consider applying Algorithm 1 with input $k'$ and with input $k$. Assuming the same execution instance, we observe that Algorithm 1 executes in the exact same way for both inputs, until it terminates when applied with input $k'$. The proof of Theorem 2 is based on this observation and the result from Theorem 1.

$\square$

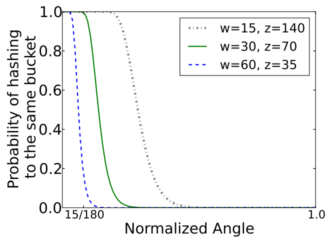


**Figure 7: Example $(w, z)$ values for Program 1 to 3.**

## 5. DESIGNING THE FUNCTION SEQUENCE

Let us now focus on how to design the transitive hashing function sequence, provided as input to Algorithm 1. The discussion for the sequence design is divided in two parts. In the first part, we discuss how to select a $(w, z)$-scheme (i.e., an LSH scheme with $z$ tables and $w$ hash functions per table) given a *budget* for the total number of hash functions (i.e., $w * z = budget$). In the second part, we discuss how to select the budget for each function in the sequence.

### 5.1 Selecting the (w,z)-scheme

Given a cost *budget*, the objective is to select the parameters $w$ and $z$ of a $(w, z)$-scheme, for the $i$-th function $H_i$ in the sequence. To simplify the discussion, we assume that the cost of applying function $H_i$ is proportional to the overall number of hash functions, and that $w$ and $z$ must be factors of *budget*, i.e., $w * z = budget$. An extension for the cases where these two assumptions do not hold is straightforward as we discuss in the end of the section.

Parameters $w$ and $z$ are selected based on the following optimization program:

$$\min_{w,z} \qquad \int_0^1 \Big[1 - \big[1 - p^w(x)\big]^z\Big] dx \qquad (1)$$

$$s.t. \qquad w * z = budget \qquad (2)$$

$$1 - \big[1 - p^w(x)\big]^z \geq 1 - \epsilon, \quad x \leq d_{thr} \qquad (3)$$

Function $p(x)$ is the probability of selecting a hash function that gives the same hash value for two records at a distance $x$, where $0 \leq x \leq 1$. (Function $p(x)$ depends on the distance metric and is given as input.) As illustrated in Example 3 (and analyzed in Appendix A) the probability of hashing to the same bucket, in a $(w, z)$-scheme, is given by: $1 - \big[1 - p^w(x)\big]^z$. The budget constraint is given in Equation 2 and the distance threshold constraint is given in Equation 3. (Parameter $\epsilon$ used in the distance threshold constraint, is also given as input.) The objective in Equation 1 states that the probability of hashing to the same bucket (for pairs of records with distance greater than the threshold $d_{thr}$), should be minimized.

EXAMPLE 5 *Consider the cosine distance as a distance metric, function $p(x) = 1 - x$ (where $x$ is the normalized angle, i.e., for an angle $\theta$, $x = \frac{\theta}{180}$), a distance threshold of $d_{thr} = \frac{15}{180}$, a parameter $\epsilon = 0.001$, and a budget of 2100 hash functions. Let us examine three pairs of $(w, z)$ values: $(15, 140)$, $(30, 70)$, and $(60, 35)$. The plot in Figure 7 is equivalent to the one in Figure 5 (angle distance between two records on the x-axis, probability of the two records hashing to the same bucket, given their distance, on the y-axis). Pair $(15, 140)$ minimizes the objective function value in Equation 1 (area under the curve), but violates the distance threshold constraint in Equation 3. Both pairs $(30, 70)$ and $(60, 35)$ satisfy the two constraints, with pair $(30, 70)$ giving a lower objective function value.*

To find the optimal $(w, z)$ values for Program 1 to 3, we can perform a binary search over $w$ values such that $\frac{budget}{w}$ is an integer. Note that the greater the value of $w$, the lower the value of the objective function (see Figure 7). Moreover, if the distance threshold constraint is not satisfied for a value of $w$, it will also not be satisfied for any greater values.

In practice, we may also want to examine $(w, z)$ values, where $\frac{budget}{w}$ is not an integer. In this case, we would have to adjust the probability expression in Equations 1 and 3: expression $\left[1 - p^w(x)\right]^z$ becomes $\left[1 - p^w(x)\right]^z * \left[1 - p^{w'}(x)\right]$, where $z = \lfloor \frac{budget}{w} \rfloor$ and $w' = budget - w * z$. In addition, we would have to exhaustively search over all possible values for $w, z$ that satisfy the budget constraint. That is, for $w \in [1, budget]$, we would examine if the distance threshold constraint is also satisfied, and keep the $(w, z)$ value pair minimizing the objective function.

Furthermore, we may also want to take into account a cost model, in the Program 1 to 3. For instance, consider two value pairs $(w_1, z_1)$ and $(w_2, z_2)$, such that $w_1{*}z_1 = w_2{*}z_2 = budget$. There are cases where the actual cost of applying a function based on a $(w_1, z_1)$-scheme is different compared to the cost for a function based on a $(w_2, z_2)$-scheme. (For example, when matrix multiplication is involved, the scheme $(w_1, z_1)$ may be more cost effective if $w_1 > w_2$.) In those cases, Equation 2 needs to include a cost function that reflects the actual cost based on a specific $(w, z)$ value pair.

## 5.2 Selecting the budget

We use two simple strategies to select the budget for each transitive hashing function $H_i$, in the sequence:

- **Exponential:** The budget for function $H_i$ is a multiple of the budget that was available for the previous function in the sequence, $H_{i-1}$. For example, if the budget for $H_1$ is 4 hash functions and we multiply the budget by 2 for every function in the sequence, the budget for $H_2$ will be 8 hash functions, the budget for $H_3$ will be 16 functions, and so on.

- **Linear:** The budget for function $H_i$ is a multiple of a constant. For example, if the constant is 100, the budget for $H_1$ is 100 hash functions, the budget for $H_2$ is 200 hash functions, the budget for $H_3$ is 300 hash functions, and so on.

In Appendix E.2, we discuss experiments with different parameter values for the two strategies and we draw conclusions regarding which strategy and values work better in each case.

## 6. EXPERIMENTAL SETTING

In this section, we give an overview of the methods, metrics, and datasets used in our experiments.

## 6.1 Methods

The methods we explore extend across two orthogonal dimensions, as we discuss next.

### 6.1.1 adaLSH vs Alternatives

We compare adaptive LSH with different blocking approaches and a traditional transitive closure algorithm (Pairs). Since our main focus is high-dimensional data, we try different blocking variations that rely on LSH.

***adaLSH***: The adaptive LSH approach we propose in this paper. The default mode is the Exponential (Section 5.2) starting with 20 hash functions for the first clustering function in the sequence; i.e., the first function applies 20 hash functions, the second 40, the third 80, and so on.

***LSH-X***: Different blocking variations that rely on LSH, adjusted for the problem studied in this paper. LSH starts by applying the same number $X$ of hash functions on every single record in the dataset. Given the number of hash functions $X$ and a distance threshold, LSH selects the number of hash tables $z$ and the number of hash functions per table $w$, by solving the same optimization problems with adaptive LSH (see Section 5.1 and Appendix C). (By solving such a problem we find the "optimal" $w, z$ values that satisfy $w * z \leq X$.) After the first stage of applying $X$ hash functions on all records, LSH uses the pairwise computation function $P$ (Definition 2) to verify if pairs of records in the same bucket are indeed within the distance threshold. To make the comparison fair, we use three additional optimizations for LSH methods: (1) LSH terminates early when there are $k$ clusters that have been "verified" using function $P$ that are larger than any other cluster not yet verified, (2) when applying function $P$ we skip checking pairs of records that are already "transitively closed" by other pairs and, hence, belong to the same cluster, and (3) we use the same efficient implementation and data structures with adaptive LSH (see Appendix B). In Appendix E.1, we also discuss experiments with a variation of LSH, that only applies the first stage and does not apply function $P$ at all. This variation, assumes that all pairs of records within each hash table bucket are within the distance threshold, and applies transitive closure on those pairs to find the $k$ largest clusters. In the plots, we just use LSH if only one LSH variation is used in the experiment. If more than one LSH variations are used, we use LSH$X$ (e.g., LSH640 applies 640 hash functions on each record) if function $P$ is applied after the first stage and LSH$X$nP otherwise.

***Pairs***: Essentially, Pairs is the application of the pairwise computation function $P$ on the whole dataset. Again, we use the above optimizations (2) and (3), i.e., we skip pairs already "transitively closed" and use the efficient implementation described in Appendix B.

### 6.1.2 Improving Accuracy

We study two different approaches for improving the accuracy for the filtering stage output:

**Return more clusters than the needed $k$:** One way to include more records from the top-$k$ entities in the filtering output, is to return more than the $k$ largest clusters found during filtering, i.e., return the $\widehat{k}$ largest clusters, for a $\widehat{k} > k$. However, note that by returning more than $k$ clusters the filtering output increases. In Section 7.3, we study the trade-offs between accuracy and performance as we increase $\widehat{k}$.

**Recovery process**: After applying an ER algorithm on the filtering output (Figure 1), we can apply an additional recovery process to retrieve additional records for top-$k$ entities that were *not* included in the filtering output. In particular, after applying an ER algorithm on the filtering output, we have $k$ clusters of records. We can then apply cleansing and aggregation on each cluster to create an summary for the information on the entity represented by each cluster. For instance, for a customer we could collect all the first and last names, email accounts, and telephone numbers she uses, or, for a news article, we could construct the most complete version covering all sentences used in different versions. The recovery process compares all the records from the initial dataset that were not included in the filtering output, with each of the $k$ clusters, after applying ER (and possibly cleansing/aggregation) on the filtering output. The goal is to find records that refer to the same entities with the entities in the $k$ clusters, that were mistakenly left out from the filtering output. Note that if all the records for a top-$k$ entity are left out from the filtering output, there is no way for this recovery process to retrieve the records for that entity and include the entity in the extended top-$k$ result.

## 6.2 Metrics

We use metrics that evaluate the *performance* and *accuracy* of the different filtering approaches.

### 6.2.1 Accuracy

**Treating filtering output as a single set of records**: in this case we compare the set of records in the filtering output with the set of records that refer to the top-$k$ entities as determined by the ground truth. In the experiments presented here, we use the precision, recall, and F1 score, as defined in Section 2.1. (We will be referring to these three metrics as Precision/Recall/F1 Gold, throughout the section.)

**Treating filtering output as a set of clusters**: To weigh the accuracy of higher ranked clusters higher, we can consider the filtered records, not a set, but a set of clusters. That is, we consider a filtering output clustering $\mathcal{C} = \{C_1, \dots, C_{|\mathcal{C}|}\}$, such that all records in each cluster $C_i$ of $\mathcal{C}$ refer to the same entity and each cluster in $\mathcal{C}$ refers to a different entity. We compare $\mathcal{C}$ to the ground truth clustering $\mathcal{C}^*$ (Section 2.1) and we compute the *mean Average Precision (mAP)* and *mean Average Recall (mAR)*. For example, assume we are looking for the top-2 entities and consider a clustering $\mathcal{C} = \big\{\{a, b, c, f\}, \{e\}\big\}$ and a ground truth clustering $\mathcal{C}^* = \big\{\{a, b, c\}, \{e, g\}\big\}$. The precision on the top-1 cluster is $\frac{3}{4}$ and the precision on the top-2 clusters is $\frac{|a,b,c,e|}{|a,b,c,f,e|} = \frac{4}{5}$. Hence, the mean average precision in this case is $\frac{0.75+0.8}{2} = 0.775$; while the mAR is $\frac{1.0+0.8}{2} = 0.9$.

Note that we do *not* consider the accuracy of the final result after applying an actual ER algorithm on the filtering output. If the ER algorithm is "perfect" the output will be exactly the same with clustering $\mathcal{C}$. Of course an actual ER algorithm would, most likely, introduce more errors, however, we can expect that the better the ER algorithm is, the closer the final result would be to clustering $\mathcal{C}$. Hence, the accuracy metrics we obtain for the filtered set should closely approximate the accuracy of the final result. (To get the actual accuracy of the final result would require knowing precisely the ER algorithm used, so the results would be specific to a single ER algorithm. We believe that an approximate accuracy value is more useful in our context.)

We also compute the Precision Gold, Recall Gold, F1 Gold, mAP, and mAR, after the recovery process of Section 6.1.2 is applied. (We call those metrics *Precision/Recall/F1/mAP/mAR with Recovery*.) For the same reasons as before, we consider a "perfect" recovery process that outputs a set of clusters $\mathcal{C} = \{C_1, \dots, C_{|\mathcal{C}|}\}$ when applied on the output $\mathcal{O}$ of a filtering approach: for each entity referenced by a record in $\mathcal{O}$, we collect all the records for that entity on the whole dataset, in a single cluster in $\mathcal{C}$. For mAP/mAR, we compare $\mathcal{C}$ to the ground truth clustering $\mathcal{C}^*$, while for Precision/Recall/F1, we compare the union of all records in $\mathcal{C}$ to the union of all records in $\mathcal{C}^*$.

### 6.2.2 Performance

**Execution Time**: the time it takes for a filtering method to compute the output.

**Dataset Reduction**: we compute the reduction percentage from the filtering stage, e.g., if the filleting output consists of 100 records while the whole dataset is 1000 records, the reduction percentage is 10%.

**Speedup w/o Recovery**: we compare the time it takes to apply ER on the whole dataset ($WholeTime$) to the time it takes to apply ER on the reduced dataset ($ReducedTime$), after the filtering stage, taking into account the time spent on filtering ($FilteringTime$). Hence, the speedup is the ratio $\frac{WholeTime}{FilteringTime+ReducedTime}$. Since we are not considering particular ER algorithms, we assume a "benchmark ER algorithm" is applied to compute $WholeTime$ and $ReducedTime$. This benchmark ER algorithm computes all the pairwise similarities in the whole or reduced dataset. On one hand, our speedup numbers are conservative, because we expect the run time of a real ER algorithm to be higher, as the benchmark ER algorithm does not take into account the actual clustering time. On the other hand, a real ER algorithm many not compute all similarities, so it would perform better than our benchmark algorithm. But without an extensive evaluation of multiple real ER algorithms in different domains, which is beyond the paper's scope, we believe the use of a benchmark ER algorithm yields valuable insights.

**Speedup with Recovery**: The speedup in this case is the ratio $\frac{WholeTime}{FilteringTime+ReducedTime+RecoveryTime}$, where $RecoveryTime$ is the run time for the recovery process. To compute the $RecoveryTime$, we consider a "benchmark recovery algorithm", that computes the pairwise similarity between each record in the filtering output with each record that was *not* included in the filtering output. For instance, if the filtering output consisted of a single record, we would compare that record with every other record in the dataset. (Again, we believe that a simple benchmark recovery algorithm is sufficient for our purpose and yields valuable insights.)

## 6.3 Datasets

We used 3 datasets in our experiments. (The ground truth entities are available in each dataset's material.)

***Cora* [1]**: a dataset of around 2000 scientific publications, extensively used in the entity resolution literature. Each record consists of the title, authors, venue, and other publication information (e.g., volume, pages, year). Together with the original dataset, we used 2x, 4x, and 8x versions. For example, the 2x version contains twice as many records as the original dataset. To extend the original dataset, we uniformly at random select an entity $a$ and uniformly at random pick a record $r_a$ referring to the selected entity $a$, for each record added to the dataset. Since each record has multiple fields in Cora, filtering methods decide if two records are a match using an AND distance rule (see Appendix C.1) with two distance thresholds. In particular, we create three sets of shingles for each record: one for the title, one for the authors, and one for the rest of the information in the record. We use the following AND rule: two records are considered a match when (i) the average jaccard similarity for the title and author sets are at least 0.7 *and* (ii) the jaccard similarity for the rest of the information is at least 0.2.

***SpotSigs*** **[2]**: a dataset of around 2200 web articles: each article is based on an original article and, thus, all articles having the same origin are considered the same entity (e.g., news articles discussing the same story with slight adjustments for different web sites). The main body of each article is transformed to a set of spot signatures based on the process described in the original paper [30]. Two records are considered a match when the jaccard similarity of their sets is at least 0.4. (We also tried thresholds of 0.3 and 0.5 in some experiments.) We also used a 2x, a 4x, and an 8x version of the dataset in the experiments, where each version is generated with the same sampling process as in Cora.

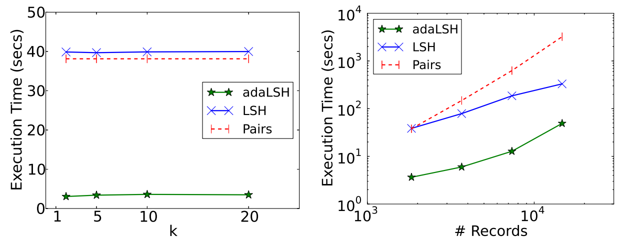
***PopularImages*** **[3]**: three datasets of 10000 images each. The images that are transformations (random cropping, scaling, re-centering) of the same original image, are considered the same entity. The unique original images are 500 popular images used and shared extensively on the web and social media, and are the same for all three datasets. The main difference between the three datasets is the distribution for the number of records per entity. They all follow a zipfian distribution, however, the exponent is different in each dataset (e.g., the top-1 entity consists of around 500, 1000, and 1700, in each dataset respectively). To compute the similarity between images, we extract for each image an RGB histogram: for each histogram bucket, we count the number of pixels with an RGB value that is within the bucket RGB limits. The RGB histogram forms a vector and we consider two images a match when the cosine distance between the images' vectors is less than an angle threshold: we used thresholds of $2, 3$, and 5 degrees in the experiments.

## 7. EXPERIMENTAL RESULTS

The experimental results are organized in four sections. First, we give an overview of our main findings in Section 7.1. In Section 7.2, we compare adaLSH to other alternatives, as discussed in Section 6.1.1, on a number of basic settings. In Section 7.3, we study the performance-accuracy trade-offs for filtering, as discussed in Section 6.1.2. In Section 7.4, we present the most interesting results from all the other settings we explored in our evaluation.

### 7.1 Findings' Overview

- adaLSH gives a 2x to 25x speedup compared to traditional blocking approaches relying on LSH. (The speedup compared to Pairs can become arbitrarily large, as the size of the dataset increases.) Moreover, the execution time for adaLSH just slightly increases as the value of $k$ increases, as long as the records in the top-$k$ entities comprise a relatively small portion of the overall dataset (e.g., the top-1 entity represents 5% of all records and the top-$k$ entities represent less than 10% of all records).
- adaLSH always gives the same (or a very slightly different) outcome as Pairs. Thus, adaLSH only introduces minimal errors due to its probabilistic nature.
- The filtering outcome can be considerably different from the ground truth set of records for the top-$k$ entities, since filtering approaches rely on simple ER rules to achieve a low run time. Even in those cases, we can increase the Recall or the mAP and mAR, by having a filtering method return the records from more than $k$ clusters; i.e., increase the value of $\widehat{k}$. While the size of the filtering output increases, the "Speedup with(w/o) Recovery" can still be very high (e.g., higher than 3x in SpotSigs4x), when using adaLSH.



(a) different $k$ values (b) different dataset sizes

**Figure 8: Execution time on Cora.**

- adaLSH always gives an important speedup compared to the best LSH variation, in any setting: we searched for the best, in terms of execution time, variation of LSH blocking in each experimental setting and we compared with adaLSH, to find out that adaLSH is always considerably faster than the best, hand-picked, LSH method.
- The distribution of number of records per entity affects the execution time of adaLSH and the LSH variations. In the distributions we tried, adaLSH would always give the best performance and showed, for at least one distribution, an important speedup compared to each LSH variation.

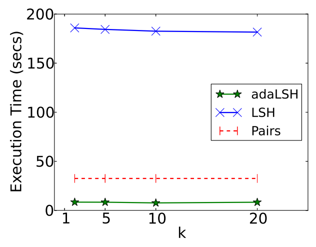
### 7.2 adaLSH vs Alternatives

We compare adaLSH, LSH1280, and Pairs, for different values of $k$ (number of top entities) and for different dataset sizes. (For LSH-$X$ methods we tried $X$ values that are a power of two and since a typical number of hash functions for LSH is 1000, we use LSH1280 in the experiments of this section. In Section 7.4, we have an extensive comparison of adaLSH with other LSH blocking variations and as we find out the optimal value of $X$ for LSH-$X$ varies from 80 to 2560, in the settings we tried.) The metrics used here are the filtering Execution Time and the F1 Gold.

#### *7.2.1 Performance*

We start by examining the execution time of adaLSH, LSH1280, and Pairs, on Cora, for different $k$ values. Our objective here is to assess the execution time, as we increase the number of entities that must be retrieved. We run experiments for $k = 2, 5, 10$, and 20. In the plot of Figure 8(a), the x-axis shows the $k$ value, and the y-axis the execution time. adaLSH gives a 10x speedup compared to LSH1280, for any $k$ value: while LSH needs to apply 1280 hash functions on all records, adaLSH starts by applying only 20 hash functions on all records and then adaptively decides which records to process further. Pairs needs almost the same time with LSH1280 and, hence, the speedup of adaLSH is also approximately 10x compared to Pairs. Furthermore, the execution time for adaLSH just slightly increases as $k$ increases. This means that the amount of computation adaLSH performs to find the top-2 entities comprises a large percentage of the overall computation for finding the top-20 entities.

Next, we study how the execution time increases for each approach as we increase the dataset size. The log-log plot in Figure 8(b) depicts the results on experiments for Cora, Cora2x, Cora4x, and Cora8x (dataset size on x-axis). We use $k = 10$ here; same results hold for other $k$ values. We observe that there is always a very large speedup for adaLSH compared to LSH1280, ranging from 9x to 20x, on the different dataset sizes. Note also that the speedup from Cora4x to Cora8x decreases. Nevertheless, we have observed that, in general, adaLSH speedup compared to LSH methods remains large (and can even increase) as we increase the dataset


(a) different $k$ values

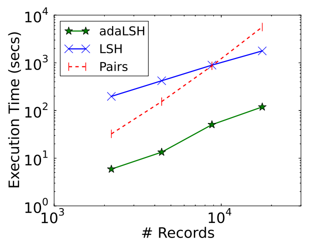

(b) different dataset sizes

**Figure 9: Execution time on SpotSigs.**

size. The only case where we found the speedup to be limited is when the records of the top-1 entity comprise a very large percentage of the overall dataset: in this case, the run time of applying $P$ on the top-1 entity dominates the execution time of both approaches (more on this in Section 7.4.2). Compared to Pairs, the speedup of adaLSH keeps increasing as the dataset size increases (e.g., 60x on Cora8x).

Let us now switch to the SpotSigs dataset, which is a higher dimensional dataset compared to Cora. We will compare the results on Cora from Figure 8(a) to the results for the same experiment on SpotSigs, in Figure 9(a). The main difference is that the cost of applying a hash function on SpotSigs is much higher compared to Cora, so the execution time for both adaLSH and LSH increases. Still, adaLSH is not affected as much as LSH: the execution time for LSH increases to around 180 seconds while for adaLSH it goes to around 7 seconds, thus, we get an impressive speedup of 25x by applying adaLSH. Furthermore, we see that, adaLSH also gives a substantial speedup of 5x compared to Pairs.

The log-log plot in Figure 9(b) is analogous to the plot of Figure 8(b), for SpotSigs1x, 2x, 4x, and 8x, when $k = 10$. The speedup of adaLSH compared to Pairs increases from 5x on SpotSigs to 50x on SpotSigs8x, while the speedup compared to LSH ranges from 15x to 25x. LSH is slower than Pairs on small datasets and shows a better performance than Pairs, only when the dataset is larger than 9000 records.

### 7.2.2 *Accuracy*

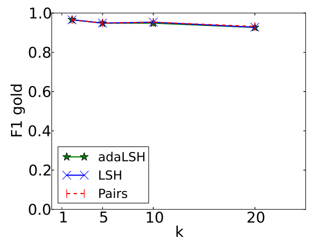
The last part in this section focuses on accuracy: we compare the outcome of the different filtering approaches to the set of records for the top-$k$ entities and we compute the F1 Gold. The plots in Figures 10(a) and 10(b) give the F1 Gold (y-axis) for the same experiments of Figures 8(a) and 9(a), on Cora and SpotSigs, respectively.

All three approaches give an almost identical F1 score, as they compute very similar clusters. Thus, the probabilistic nature of adaLSH and LSH1280 does not introduce errors. (However, there are other LSH methods that do introduce errors, as we discuss in Appendix E.1.) For Cora the filtering output is very close to the ground truth, i.e., we get a very high F1 score on all $k$ values. Nevertheless, the F1 score in case of SpotSigs (Figure 10(b)) is low (around 0.8) for $k = 5$ and 10: as the filtering stage relies on simple record matching rules to boost performance, we can expect that the accuracy will not always be high, on all datasets.

In the next section, we study more closely the errors introduced by the filtering stage and how we can limit those errors by increasing the size of the filtering output.

## 7.3 Improving Accuracy

As discussed in Section 6.1.2, we can handle situations where the filtering outcome may not be sufficiently accurate, by increasing the number of clusters $\widehat{k}$ that a filtering


(a) Cora

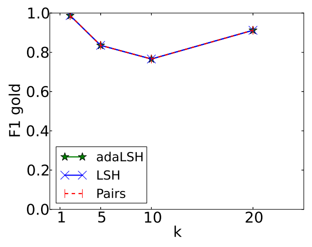
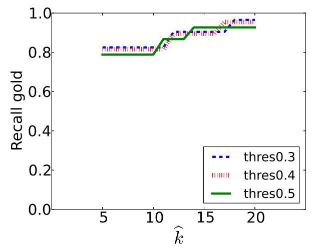

(b) SpotSigs

**Figure 10: F1 Gold for different $k$ values.**


(a) Recall

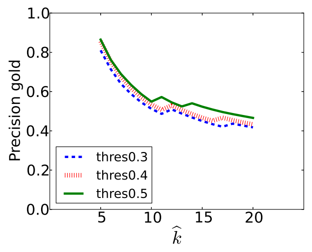

(b) Precision

**Figure 11: Precision/Recall on SpotSigs, for $k = 5$.**

method must return. Remember that the main goal of the filtering stage is to reduce the size of the initial dataset. Therefore, by increasing $\widehat{k}$, we can get a recall that is very close to 1.0, while still significantly reducing the size of the original dataset, as we discuss in this section.

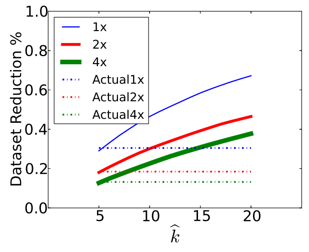
### 7.3.1 *Precision and Recall Gold*

To illustrate the trade-off between precision and recall, we focus on a $k$ value of 5 for SpotSigs, where the F1 score is just above 0.8, as we saw in Figure 10(b). In the experiment of Figure 11(a), the x-axis shows $\widehat{k}$, i.e., the number of clusters we ask a method to return. Then, we compute the precision and recall of this set of records against the set of records in the ground truth top-5 entities. Figure 11(a) shows the Recall Gold on the y-axis. Since adaLSH and LSH give practically the same output with Pairs, we plot just one curve for all three methods. Moreover, we include the results for three different similarity thresholds, 0.3, 0.5, and the default 0.4 (one curve for each threshold), in Figure 11(a).

The recall for all thresholds is almost the same and, more importantly, follows the same trend: recall keeps going up as $\widehat{k}$ increases, to reach very close to 1.0 for 20 clusters.

By increasing the output's size, the precision inevitably drops. Figure 11(b) shows the Precision Gold for the same setting with Figure 11(a). As we increase $\widehat{k}$ from 5 to 20, the precision drops from 80% to almost 40%. A lower precision means that a lower percentage of the original dataset will be filtered out, which in turn means that the Speedup w/o (with) Recovery will be lower. We quantify the trade-off between accuracy and reduction in Sections 7.3.2 to 7.3.4.

It is important to note here that we would not necessarily increase the recall, by relaxing the similarity threshold, instead of including more clusters in the output. Consider, for example, trying to find the top-1 entity in a dataset. Let us assume that for a similarity threshold of 0.9, the largest cluster a method finds, contains only 80% of all the records in the ground truth top-1 entity. If we relax the threshold to 0.8, the method may merge the second largest cluster with a smaller one, which would now form the largest cluster. However, the new largest cluster may be actually referring to the ground truth top-2 entity and contain all the records for that entity, but none of the top-1 entity records. Hence, the recall would drop from 80% to 0% by relaxing the threshold.


(a) Dataset Reduction

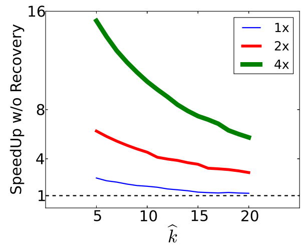
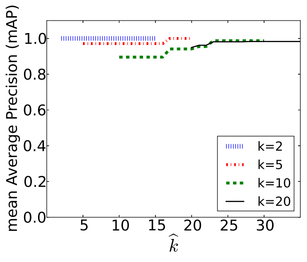

(b) Speedup w/o Recovery

**Figure 12: Reduction % and Speedup on SpotSigs.**


(a) mean Average Precision

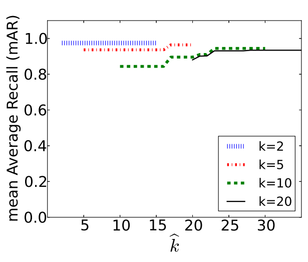

(b) mean Average Recall

**Figure 13: mAP and mAR on SpotSigs.**

### 7.3.2 Reduction and Speedup

Figure 12(a), depicts the Dataset Reduction percentage (y-axis), for different $\widehat{k}$ values, on different dataset sizes (SpotSigs1x, 2x, and 4x), for $k = 5$. In addition to the three curves, we plot one horizontal dashed line for each dataset size, indicating the actual percentage of records for the ground truth top-$k$ entities (Actual1x, 2x, and 4x). (Again, adaLSH and LSH give the same output as Pairs, so we plot just one curve for all three methods.) As $\widehat{k}$ increases, a filtering method ends up returning a larger portion of the dataset. However, for larger datasets this percentage is still rather small even for a large number of clusters returned (e.g., less than 40% on SpotSigs4x, for $\widehat{k} = 20$).

To see how those percentages translate to speedups, we plot the Speedup w/o Recovery (y-axis) in Figure 12(b), where we use adaLSH for the filtering stage. That is, in the Speedup w/o Recovery formula of $\frac{WholeTime}{FilteringTime + ReducedTime}$ (see Section 6.2), the $FilteringTime$ is the execution time for adaLSH. We see that the speedup increases as the dataset size increases, and even for a reduction percentage of 40% (SpotSigs4x at $\widehat{k} = 20$), we get a significant speedup of 6x.

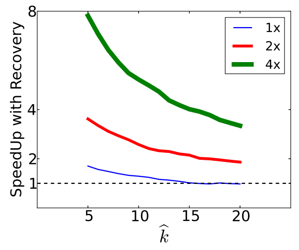
### 7.3.3 mean Average Precision and Recall

Next, we study how accuracy, in terms of mAP and mAR, increases, as $\widehat{k}$ increases. The goal is to understand how easily we can reconstruct the ground truth outcome, when we apply a "perfect" ER algorithm (see Section 6.2) on the reduced dataset. Figures 13(a) and 13(b) show the mAP and mAR, respectively, on the y-axis, while the x-axis shows $\widehat{k}$. We plot results for different $k$ values; one curve per $k$ value. We see that for all values of $k$, the mAP eventually reaches 1.0 as we increase $\widehat{k}$; results for mAR are slightly worse.

The comparison between the mAP/mAR metrics in Figures 13(a), 13(b) with the precision and recall in Figures 11(a), 11(b), points out a beneficial fact: accuracy for higher-ranked entities is higher. For instance, for $k = \widehat{k} = 5$, the set-based precision and recall are around 0.8, while the ranked-cluster precision and recall are higher than 0.9.

### 7.3.4 Recovery

The last experiment of this section focuses on the recovery process discussed in Section 6.1.2, for the Speedup with Recovery and mAP(mAR) with Recovery metrics. Figure 14(a) corresponds to Figure 12(b) and depicts the Speedup with Recovery with adaLSH used for filtering, while Figure 14(b) corresponds to Figures 13(a) and 13(b) and depicts the mAP with Recovery; the results for the mAR with Recovery are almost identical to the ones for mAP with Recovery.


(a) Speedup with Recovery

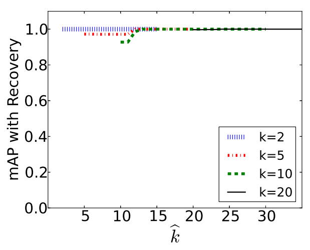
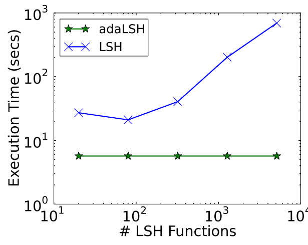

(b) mAP with Recovery

**Figure 14: Applying recovery on SpotSigs.**


(a) SpotSigs

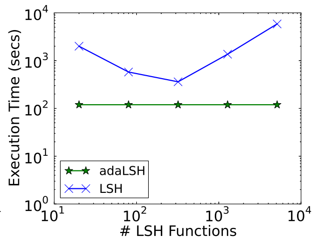

(b) SpotSigs8x

**Figure 15: adaLSH vs different LSH variations.**

In Figure 14(a), we see that the speedup decreases as $\widehat{k}$ increases. As expected, the speedup is lower compared to the Speedup w/o Recovery since we also include the run time for recovery, in this case. Nevertheless, as the dataset size increases the speedup increases: in SpotSigs4x, even for a large $\widehat{k}$ value of 20, we get an almost 4x speedup. With respect to accuracy, we observe that the mAP with Recovery, in Figure 14(b), very quickly reaches to 1.0 for all $k$ values, as we increase $\widehat{k}$.

Whether it is useful to increase $\widehat{k}$ and/or use recovery depends on the desired accuracy level and performance. But our results do suggest that both techniques are useful tools: they can noticeably increase accuracy while still retaining significant speedups over a non-filtered approach.

## 7.4 Other Settings

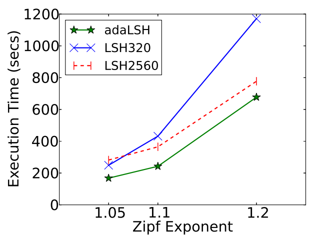
In addition to the experiments presented so far, we have also explored many other settings. Here we describe a few of the more interesting results from all the experiments we ran.

### 7.4.1 adaLSH vs best LSH variation

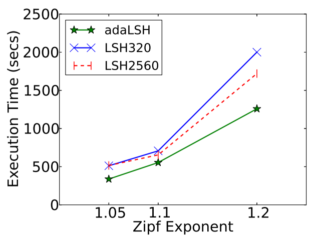
Until now we used only LSH1280 in the plots. Here, we explore the different LSH blocking variations. Clearly, knowing in advance which LSH variation is better in each case is not possible. Our goal here is to examine how much better adaLSH is, compared to the best LSH variation, if we knew in advance the best LSH variation to use in each case.

The log-log plot in Figure 15(a) shows the execution time on SpotSigs and $k = 10$, for five LSH variations: LSH20 to LSH5120. That is, the x-axis shows the number of hash functions used by LSH and the y-axis the execution time. (For adaLSH we plot the same execution time for all x-axis values.) We see that adaLSH gives a 4x speedup even when compared to the best LSH variation; LSH80 in this case.

When the size of the dataset increases we expect that some other LSH variation will perform better. Indeed, as we see in Figure 15(b), for SpotSigs8x and $k = 10$, LSH320



(a) $d_{thr} = 3$ degrees



(b) $d_{thr} = 5$ degrees

**Figure 16: Execution time on PopularImages.**

is now the lowest execution time variation. Still, adaLSH gives a 3x speedup compared to LSH320.

Hence, as the experiments in this section point out, a big advantage of adaLSH compared to LSH variations is that adaLSH does not need tuning with respect to the number of hash functions to apply. In addition, adaLSH can apply a different number of hash functions on different records and manages to achieve 3-4x speedups compared to the optimally tuned LSH version.

We have also run experiments with other LSH blocking variations that can be found in Appendix E.1.

### 7.4.2 *Entity sizes' distribution*

The last experiments discussed in this section focus on the PopularImages dataset. Our objective here is *not* to illustrate how large the adaLSH speedup can be, but instead study a more challenging scenario for adaLSH. When using the cosine distance for RGB histograms, for almost every image in the dataset, there are images that refer to a different entity but have a similar histogram with that image. (Clearly, there are better features we can extract for each image, still, the cosine distance for RGB histograms serves our propose well, here.)

In addition, we wanted to study how the distribution of records per entity affects the performance of each method, in this case. The three datasets in PopularImages, follow zipfian distributions, with exponents of 1.05, 1.1, and 1.2, respectively. For instance, in the 1.05-exponent dataset, the top-1 entity consists of around 500 records, the top-2 entity of around 250 records, and the top-3 of around 150 records, while in the 1.2-exponent dataset, the top-1 entity consists of around 1700 records, top-2 entity of around 800 records, and the top-3 entity of around 500 records.

Figure 16(a) depicts the execution time for adaLSH, LSH320, and LSH2560, for a cosine distance threshold of 3 degrees and $k = 10$. The x-axis shows the zipfian exponent and the y-axis the execution time for each method. (Pairs takes almost one hour to run and we do not include it in this plot.)

Even in this, far from ideal, scenario, adaLSH gives a 1.5 speedup for a zipfian exponent of 1.05, and a 1.7 speedup for exponents of 1.1 and 1.2, compared to LSH320. Compared to LSH2560, adaLSH gives a 1.7 speedup for an exponent of 1.05 and a 1.5 speedup for an exponent of 1.1. The results for a threshold of 5 degrees appear in Figure 16(b). The adaLSH speedup ranges from 1.2 to 1.5 compared to LSH2560 and from 1.3 to 1.6 for LSH320.

Note that the execution time for both thresholds increases as the exponent increases. The main reason for this increase is the sizes of the top entities, that, as discussed above, increase as the exponent increases. For example, applying the pairwise computation function $P$ on the top-1 entity

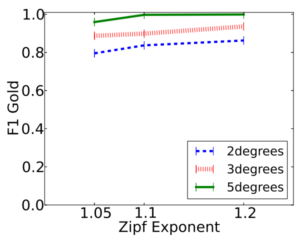


**Figure 17: F1 Gold on PopularImages.**

often takes more than 50% of the execution time. LSH320 that applies less hash functions than LSH2560 in the first stage, ends up applying function $P$ on clusters even larger than the top-1 entity, and the increase in execution time, as the exponent increases, is even more evident.

The execution time also increases as we relax the distance threshold from 3 to 5 degrees, as we see when comparing Figures 16(a) and 16(b). The reason is again the larger sizes for the clusters that need to be "verified" using function $P$: a relaxed threshold gives larger clusters.

Note also that LSH320, which clearly underperforms here, was (together with LSH80) the most effective LSH variation in the experiments, in Figures 15(a) and 15(b). This illustrates, again, that the most effective LSH variation can be very different in each case. On the other hand, adaLSH always gives a better performance without requiring tuning.

The last effect we discuss here, is the tradeoff between performance and accuracy, as the distance threshold and the zipfian exponent change. In Figure 17, we plot the F1 gold (y-axis) for thresholds $2, 3$, and 5 and for exponents 1.05, 1.1, and 1.2 (x-axis), for $k = 10$. All three methods give almost the same F1 score, so we just use one curve for each threshold. As the distance threshold drops from 5 to 2 degrees, there are images that refer to the same entity, but still do not get clustered together because of the more strict threshold. As we see in Figure 17, the more strict the threshold, the lower the F1 score. Moreover, we see that the lighter the tail (the higher the exponent) is, the higher the F1 score becomes, as the top-10 entities form larger clusters and errors happen to a lesser extent. Overall, while a smaller threshold lets methods run faster, it also introduces more errors, in this case.

## 8. RELATED WORK

**Blocking:** To enable entity resolution on large datasets, many blocking approaches have been suggested for different settings [24, 5, 7, 25]. Paper [12] proposes a mechanism that automatically learns hash functions for blocking and is applicable on heterogeneous data expressed in multiple schemas without requiring predefined blocking rules. Blocking over heterogeneous data is also the topic in paper [27]. The framework in paper [17], is able to produce blocks that satisfy size constraints, i.e., blocks are not larger than an upper threshold (e.g., for performance) and/or blocks are not smaller than a lower threshold (e.g., for privacy). Distributed blocking is the topic of paper [11] that models the communication-computation trade-off and proposes strategies to distribute the pairwise comparison workload across nodes. In paper [36], the concept of iterative blocking is introduced, where results from one block are used when processing other blocks, iteratively, to improve accuracy, by detecting additional record matches, and reduce execution time, by skipping unnecessary record comparisons. Blocking using LSH is applied in Helix [14], a large scale data exploration system. Supervised meta-blocking [28] uses a set of training examples to efficiently re-structure a collection of blocks and avoid unnecessary record comparisons while minimizing the number of missed record matches.

**Entity Resolution:** A good overview of traditional ER approaches can be found in surveys [16] and [37]. Here, we will try to cover a few more recent studies with a connection to the setting in this paper. Paper [35] uses a set of positive (records that match) and negative examples (records that do not match) to find the best similarity functions and thresholds to use in a dataset; note that our approach could be combined with such a method that selects similarity functions and computes the "right" threshold for each function. Examples can also be provided in an active manner as research in crowd entity resolution suggests [6, 34, 33, 32, 38, 13]. An alternative of using examples, is defining constraints for matching records, through declarative/interactive frameworks [9, 15]. Entity resolution is also studied in settings where the data is distributed across multiple nodes [4], and the goal is to reduce the bandwidth usage while maintaining a low execution time. Incremental ER is the focus in paper [18], where data updates can be handled efficiently and can also provide evidence to fix previous errors.

**Locality-Sensitive Hashing:** Here, we briefly discuss a number of LSH variations that involve some notion of adaptivity; although quite different from the LSH adaptivity concept introduced in this paper. Multi-Probe LSH [23] reduces the number of hash tables it uses, by probing multiple buckets in each table when searching for items similar to a query item (e.g., image or video). Another similar concept is the entropy-based LSH [26, 21], which trades time for space requirements, for nearest-neighbor search on Euclidean spaces. Bayesian [29] and sequential hypothesis testing LSH [10] use the hash values generated in the first stage of LSH, to efficiently verify if each two records in the same hash bucket are indeed within the threshold. Paper [22] focuses on a specific LSH family of functions, the minwise hashing functions [8] for jaccard similarity: based on a theoretical framework, only a few bits are kept for each hash value, in order to reduce space and computational overhead. Paper [31] also focuses on a specific function family, random projections for cosine similarity, and proposes a mechanism that trades accuracy for space, in an online setting for LSH. The last LSH variation discussed here, is adaptive with respect to nearest-neighbor queries [20]: a notion of accuracy, with respect to queries, is defined for each hash function and, at query time, the most appropriate hash functions are selected.

## 9. CONCLUSION

We proposed adaptive LSH, a novel approach for finding in large datasets the records referring to the top-$k$ entities. This problem is motivated by many modern applications that focus on the few most popular entities in a dataset. The main component of our approach is a sequence of clustering functions that adaptively apply locality-sensitive hashing (LSH). The large cost savings come from applying only the few first lightweight functions in the sequence on the vast majority of records and detecting with a very low cost that those records do *not* refer to the top-$k$ entities. Our approach is general and applicable in all types of data where a distance metric can model how likely two records are to refer to the same entity. The filtering output is a drastically reduced dataset that can be used to very accurately and efficiently find the top-k entities using the full ER algorithm.

Our experiments involved different types of data: multi-field publication records, web articles, and images. We compared adaptive LSH to the common, for high dimensional data, LSH-blocking approach. The speedup ranges from 2x to 25x compared to traditional LSH blocking, while introducing only negligible errors due to the approach's probabilistic nature. Furthermore, our adaptive LSH approach does *not* require knowing in advance the right number of hash functions to apply and may apply a different number of hash functions on different records. We also presented two schemes for improving accuracy further, with modest performance overhead.

In the future, we plan to explore a couple of interesting directions for adaptive LSH. First, we believe that adaLSH can offer large performance gains in online settings, where we do not have a fixed dataset and input records arrive dynamically: we plan to study mechanisms that decide, for instance, how many hash tables and clusters to maintain or decide, for a new record, between applying hashing or comparing with existing clusters. Second, we can apply adaLSH in an incremental way, so that it outputs clusters during its execution, allowing users observe results before its completion. We briefly discuss this incremental mode in Section 4.2. We also plan to study how we can pipeline the execution of the filtering stage with ER algorithms.

## 10. REFERENCES

[1] Cora dataset. `people.cs.umass.edu/~mccallum/data/cora-refs.tar.gz`.

[2] Gold set of near duplicates. `http://mpi-inf.mpg.de/~mtb/spotsigs/GoldSetOfNearDuplicates.tar.gz`.

[3] Popular images. `stanford.edu/~verroios/datasets/popimages.zip`.

[4] N. Ayat, R. Akbarinia, H. Afsarmanesh, and P. Valduriez. Entity resolution for distributed probabilistic data. *Distributed and Parallel Databases*, 31(4):509–542, 2013.

[5] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification*, 2003.

[6] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *KDD*, 2012.

[7] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, 2006.

[8] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *WWW*, 1997.

[9] D. Burdick, R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. A declarative framework for linking entities. *TODS*, 41(3):17:1–17:38, 2016.

[10] A. Chakrabarti and S. Parthasarathy. Sequential hypothesis tests for adaptive locality sensitive hashing. In *WWW*, 2015.

[11] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[12] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.

[13] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Large-scale linked data integration using probabilistic reasoning and crowdsourcing. *The VLDB Journal*, 22(5):665–687, 2013.

[14] J. Ellis, A. Fokoue, O. Hassanzadeh, A. Kementsietsidis, K. Srinivas, and M. J. Ward. Exploring big data with helix: Finding needles in a big haystack. *SIGMOD Rec.*, 43(4):43–54, 2015.

[15] A. Elmagarmid, I. F. Ilyas, M. Ouzzani, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Nadeef/er: Generic and interactive entity resolution. In *SIGMOD*, 2014.

[16] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[17] J. Fisher, P. Christen, Q. Wang, and E. Rahm. A clustering-based framework to control block sizes for entity resolution. In *KDD*, 2015.

[18] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental record linkage. *PVLDB*, 7(9):697–708, 2014.

[19] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[20] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros. Query adaptative locality sensitive hashing. In *ICASSP*, 2008.

[21] M. Kapralov. Smooth tradeoffs between insert and query complexity in nearest neighbor search. In *PODS*, 2015.

[22] P. Li and A. C. König. Theory and applications of b-bit minwise hashing. *Commun. ACM*, 54(8):101–109, 2011.

[23] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[24] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, 2000.

[25] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, 2006.

[26] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, 2006.

[27] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Beyond 100 million entities: Large-scale blocking-based resolution for heterogeneous data. In *WSDM*, 2012.

[28] G. Papadakis, G. Papastefanatos, and G. Koutrika. Supervised meta-blocking. *PVLDB*, 7(14):1929–1940, 2014.

[29] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[30] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: Robust and efficient near duplicate detection in large web collections. In *SIGIR*, 2008.

[31] B. Van Durme and A. Lall. Efficient online locality sensitive hashing via reservoir counting. In *HLT*, 2011.

[32] V. Verroios and H. Garcia-Molina. Entity resolution with crowd errors. In *ICDE*, 2015.

[33] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. In *VLDB*, 2012.

[34] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

[35] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[36] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, 2009.

[37] W. Winkler. Overview of record linkage and current research directions. *Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC*, 2006.

[38] C. Zhang, R. Meng, L. Chen, and F. Zhu. Crowdlink: An error-tolerant model for linking complex records. In *ExploreDB*, 2015.

## APPENDIX

## A. LOCALITY-SENSITIVE HASHING

The clustering functions of the approach proposed in this paper use LSH, as discussed in Section 3. In this Appendix, we present the details for LSH.

LSH uses a set of hash tables and applies a number of hash functions on each record, such that two records that are "close" to each other, based on the given distance metric and distance threshold, hash to the same bucket, in at least one of the tables.

In particular, LSH is based on the notion of $(d_t, \rho d_t, p_1, p_2)$-sensitive functions:

DEFINITION 4 (Locality-Sensitive Family) *For a given distance metric $d$, a $(d_t, \rho d_t, p_1, p_2)$-sensitive family $\mathcal{F}$ consists of hash functions, where each function $h : R \to B$ maps a record $r \in R$ to a bucket $b \in B$ and has the following two properties for records $r_1$ and $r_2$:*

- *if $d(r_1, r_2) \leq d_t$, then $h(r_1) = h(r_2)$ with probability* at least $p_1$.
- *if $d(r_1, r_2) \geq \rho d_t$, then $h(r_1) = h(r_2)$ with probability* at most $p_2$.

That is, when selecting a hash function $h \in \mathcal{F}$ uniformly at random, for two records $r_1$ and $r_2$ with $d(r_1, r_2) \leq d_t$, the probability of selecting a function $h$ with $h(r_1) = h(r_2)$ is at least $p_1$. If $d(r_1, r_2) \geq \rho d_t$ the probability of selecting a function $h$ with $h(r_1) = h(r_2)$ is at most $p_2$. Note that $\rho$ needs to be greater than one and $p_1$ greater than $p_2$, for $\mathcal{F}$ to be useful. Intuitively, when picking a hash function from a $(d_t, \rho d_t, p_1, p_2)$-sensitive family $\mathcal{F}$, we will pick with high probability a function that hashes to the same bucket two records that are very similar, and we will pick with high probability a function that hashes to different buckets two records that are not very similar.

EXAMPLE 6 *Consider again Example 2. When selecting uniformly at random a hyperplane through the origin, the likelihood of picking a hyperplane like $e_2$ (where $r_1$ and $r_2$ lie on different sides) is $\frac{30}{180}$; while the likelihood of picking a hyperplane like $e_1$ is $\frac{180-30}{180}$. As discussed in Example 2, we consider each random hyperplane as a hash function, $h : R \to \{b_1, b_2\}$, that hashes each vector/record, $r \in R$, to two buckets, $b_1$ or $b_2$, depending on which side of the hyperplane vector $r$ lies. In general, the family of hash functions defined by the random hyperplanes, is $(\theta_1, \theta_2, \frac{180-\theta_1}{180}, \frac{180-\theta_2}{180})$-sensitive, for $\theta_1, \theta_2 \in [0, 180]$ and $\theta_1 < \theta_2$. That is, if the distance between two vectors/records is $\theta$, the likelihood of picking a hash function that hashes the two vectors to the same bucket is exactly $\frac{180-\theta}{180}$.*

Although in Example 6 the space is two dimensional, it is not hard to see that the family of hash functions defined by random hyperplanes, is $(\theta_1, \theta_2, \frac{180-\theta_1}{180}, \frac{180-\theta_2}{180})$-sensitive for any number of dimensions, for the cosine distance.

A $(d_t, \rho d_t, p_1, p_2)$-sensitive family can be "amplified" using an *AND-construction* or an *OR-construction*:

DEFINITION 5 (AND-construction) *Given a $(d_t, \rho d_t, p_1, p_2)$-sensitive family $\mathcal{F}$, a $(d_t, \rho d_t, p_1^w, p_2^w)$-sensitive family $\mathcal{F}'$ is constructed by selecting $w$ functions, $h_1, h_2, \dots h_w \in \mathcal{F}$, to define a function $h' \in \mathcal{F}'$ such that $h'(r_1) = h'(r_2)$ iff $h_i(r_1) = h_i(r_2)$ for all $i \in [1, w]$, for two records $r_1$, $r_2$.*

DEFINITION 6 (OR-construction) *Given a $(d_t, \rho d_t, p_1, p_2)$-sensitive family $\mathcal{F}$, a $(d_t, \rho d_t, 1-(1-p_1)^z, 1-(1-p_2)^z)$-sensitive family $\mathcal{F}'$ is constructed by selecting $z$ functions, $h_1, h_2, \dots h_z \in \mathcal{F}$, to define a function $h' \in \mathcal{F}'$ such that $h'(r_1) = h'(r_2)$ iff $h_i(r_1) = h_i(r_2)$ for at least one $i \in [1, z]$, for two records $r_1$, $r_2$.*

For the AND-construction, if the probability of selecting a hash function $h \in \mathcal{F}$ with $h(r_1) = h(r_2)$ is $p_1$ (or $p_2$), it follows that the probability of selecting $w$ functions $h_1, h_2, \dots h_w \in \mathcal{F}$, with all of them having $h_i(r_1) = h_i(r_2)$ ($i \in [1, w]$), is $p_1^w$ (or $p_2^w$).

For the OR-construction, if the probability of selecting a hash function $h \in \mathcal{F}$ with $h(r_1) = h(r_2)$ is $p_1$ (or $p_2$), it follows that the probability of selecting $z$ functions $h_1, h_2, \ldots h_z \in \mathcal{F}$, with none of them having $h_i(r_1) = h_i(r_2)$ $(i \in [1, z])$, is $(1 - p_1)^z$ (or $(1 - p_2)^z$). Hence, the probability of at least one having $h_i(r_1) = h_i(r_2)$ is $1-(1-p_1)^z$ (or $1-(1-p_2)^z$).

The two constructions can be combined together to form an AND-OR construction. In particular, a $(d_t, \rho d_t, p_1, p_2)$-sensitive family $\mathcal{F}$ is first transformed to a $(d_t, \rho d_t, p_1^w, p_2^w)$-sensitive family $\mathcal{F}'$ using an AND-construction, and then $\mathcal{F}'$ is transformed to a $(d_t, \rho d_t, 1-(1-p_1^w)^z, 1-(1-p_2^w)^z)$-sensitive family $\mathcal{F}''$ using an OR-construction.

As discussed in Section 3, the AND-OR construction can be thought of as a hashing scheme of $z$ hash tables: in each of the $z$ tables two records $r_1$ and $r_2$ hash to the same bucket if $h_i(r_1) = h_i(r_2)$ for all of the $w$ hash functions $h_i$, for that table. (For each table there is an independent selection of $w$ functions $h_i \in \mathcal{F}$.)

## B. IMPLEMENTATION DETAILS

Here, we discuss how to efficiently implement all the key components of Algorithm 1, in Section 4. In particular, we discuss the implementation for transitive hashing functions (Line 8) and the pairwise computation function (Line 6), finding the largest cluster (Line 3), and the termination condition (Line 11). We start with the description of two data structures, and then we focus on how the two structures are used in the lower level operations in Algorithm 1.

### B.1 Data Structures

The data structures used in the implementation are a parent-pointer tree structure and a bin-based structure. The parent-pointer tree structure is used by transitive hashing functions and the pairwise computation function, while the bin-based structure is used for finding the largest cluster and in the termination condition.

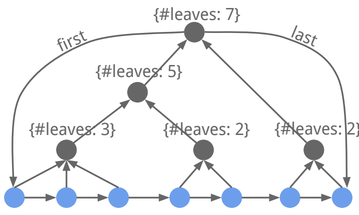


**Figure 18: Parent-pointer tree.**

The parent-pointer tree structure is depicted in Figure 18. Each node has a pointer to the parent, leaf nodes have a pointer to the first leaf on the right, and the root has a pointer to the first and last leaves. Each parent-pointer tree represents a cluster: the leaves of the tree refer to the records that belong to the cluster. In addition, each node stores the number of leaves that are successors of that node.

The bin-based structure is an array of $log(|R|)$ bins; where $|R|$ is the number of all records in the dataset. In each bin, the roots of different parent-pointer trees are stored. The root of a parent-pointer tree with $x$ leaves, is stored on the $log(\lfloor x \rfloor)$-th bin of the array. For example, an array for $|R| = 10$ records would have four bins: the first bin would store trees with 1 leaf, the second bin trees with 2 or 3 leaves, the third bin trees with 4 to 7 leaves, and the fourth bin trees with 8 to 10 leaves.

### B.2 Transitive Hashing Functions

A transitive hashing function $H_i$ based on a $(w_i, z_i)$-scheme, uses $z_i$ hash tables. For each record $r$ of an input set $S$, $z_i$ bucket indices (consisting of $w_i$ hashes each) are computed. Based on those hashes, record $r$ is added to each of the $z_i$ tables. (Note that the computation of hashes is incremental and uses the hashes computed from the previous function in the sequence $H_{i-1}$, on record $r$.) Hashing function $H_i$ uses a number of parent-pointer trees: each cluster in the output refers to one parent-pointer tree. When function $H_i$ is invoked, there are no trees and none of the input records belongs to a tree. Moreover, the $z_i$ hash tables are empty, i.e., each invocation of function $H_i$ uses a different set of tables; to avoid a possible merge of clusters from different invocations. To process a cluster of records stored in a parent-pointer tree, function $H_i$ uses the "first" pointer in the root to reach the first leaf, processes that record, then uses the "right" pointer of the first leaf to access the next record in the cluster, and so on. When a record $r_1$ is added to a hash table there are four cases:

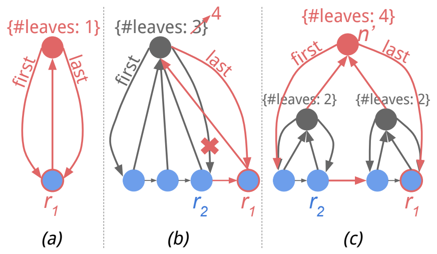


**Figure 19: Tree updates when adding a record $r_1$ to a hash table.**

1. the bucket in the table is empty and record $r_1$ has *not* been added yet to a parent-pointer tree: a new tree is created with record $r_1$ being the single leaf of that tree, as depicted in Figure 19a.
2. the bucket in the table is empty and record $r_1$ has already been added to a parent-pointer tree: just add $r_1$ in the bucket.
3. the bucket in the table is *not* empty and record $r_1$ has *not* been added yet to a parent-pointer tree: find the root of the tree of the record $r_2$ that was last added in the bucket and add record $r_1$ to this tree, by updating all tree pointers (updates appear in red on Figure 19b).
4. the bucket in the table is *not* empty and record $r_1$ has already been added to a parent-pointer tree: find the root of the tree of the record $r_2$ that was last added in the bucket. If the root for $r_2$ is the same with the root of the tree of $r_1$ (i.e., the two records belong to the same tree), just add $r_1$ to the bucket. Otherwise, merge the two trees into one: use a new node $n'$ as a root and update all pointers, as depicted in Figure 19c.

Note that all records of a bucket in a hash table are under the same tree. To find the root of a bucket's tree, the process starts from the record that was last added in the bucket, in cases 3 and 4, because it is more likely that the path to the root is shorter, compared to when starting from the, say, first record added in the bucket.

The complexity of adding a record $r$ to a hash table is $O(log(|C_r|))$, where $C_r$ is the cluster where record $r$ belongs in the output of function $H_i$.

## B.3 Pairwise Computation Function

The pairwise computation function $P$ also uses parent-pointer trees. When the distance between two records is less than the distance threshold, the trees of the two records are merged; the process is similar to the one discussed in the previous section. In addition, for two records that belong to the same tree, $P$ can safely skip the distance computation for those two records. Nevertheless, note that in our cost model (Line 5 in Algorithm 1) we are being conservative and assume that the cost of function $P$ involves the computation of all pairwise distances.

## B.4 Finding the Largest Cluster

Upon completion of a function $H_i$ or $P$, the output clusters(trees) are added to the bin-based array. When the largest cluster must be found for the next iteration, the search starts from the last *non-empty* bin in the array and the largest cluster in that bin is returned and removed from the bin.

Adding a cluster to the bin-based array is a constant-time operation and we expect that the clusters in the last *non-empty* bin to always be much fewer than all the clusters stored in the array.

## B.5 Termination Condition

To efficiently compute when the loop of Algorithm 1 must terminate, we use an array of "final" clusters. When the largest cluster selected in an iteration, is an outcome of a function $H_L$ or a function $P$, the cluster is not processed but it is added instead to the array of final clusters. Once $k$ clusters are added in the final clusters array, Algorithm 1 terminates and the clusters in the final clusters array are returned as the output. Note that this condition is equivalent to the condition in Line 11 of Algorithm 1.

## C. COMPLEX DISTANCE RULES

When records have multiple fields, distance rules may involve more than one field, as discussed in Section 3. The main workflow of Adaptive LSH, summarized in Algorithm 1, remains the same in that case, however, some of the details of transitive hashing functions and the design of the function sequence change.

We focus on distance rules that consist of: AND rules, OR rules, and weighted average rules. Next, we discuss how to design the sequence of transitive hashing functions for each type of rules and conclude this section with a brief discussion on the case of more complicated rules that combine several AND, OR, and weighted average rules.

## C.1 AND rules

To keep the discussion simple, we assume the AND rule involves only two record fields: given a distance metric and threshold for each field, two records $r_1 = \{f_1^{(1)}, f_1^{(2)}\}$ and $r_2 = \{f_2^{(1)}, f_2^{(2)}\}$ refer to the same entity if

$$d(f_1^{(1)}, f_2^{(1)}) \leq d_{thr}^{(1)} \quad AND \quad d(f_1^{(2)}, f_2^{(2)}) \leq d_{thr}^{(2)}$$

In the AND-OR hashing scheme used by function $H_i$ in the sequence, the hash value for each of the hash tables will be formed using both fields $f^{(1)}$ and $f^{(2)}$. In particular, given a Locality-Sensitive family of hash functions for each field, for each hash table used by function $H_i$, we pick $w$ hash functions from the family of field $f^{(1)}$ and $u$ hash functions from the family of field $f^{(2)}$. The hash value for each hash table is a concatenation of the $w$ and $u$ hash values.

Consider functions $p_1(x_1)$ and $p_2(x_2)$ that give the probability of selecting a hash function that gives the same hash value for two records at a distance $x_1$ ($x_2$) on field $f^{(1)}$ ($f^{(2)}$); $0 \leq x_1, x_2 \leq 1$. Assuming $z$ tables are used, the probability of two records at a distance $x_1$ on field $f^{(1)}$ and $x_2$ on field $f^{(2)}$, hashing to the same bucket in any of the $z$ tables, is:

$$1 - \left[1 - p_1^w(x_1) p_2^u(x_2)\right]^z$$

To decide the values $w$, $u$, $z$, for a given *budget* of hash functions, we use a generalization of Program 1 to 3:

$$\min_{w,u,z} \quad \int_0^1 \int_0^1 \left[1 - \left[1 - p_1^w(x_1) p_2^u(x_2)\right]^z\right] dx_1 dx_2 \quad (4)$$

$$s.t. \quad (w+u) * z = budget \quad (5)$$

$$1 - \left[1 - p_1^w(x_1) p_2^u(x_2)\right]^z \geq 1 - \epsilon, \ x_1 \leq d_{thr}^{(1)}, x_2 \leq d_{thr}^{(2)} \quad (6)$$

Just as in Program 1 to 3, we can also search over $w, u, z$ values, where $\frac{budget}{w+u}$ is not an integer, by adjusting the probability expression, or take into account cost functions that reflect the actual cost of computing each hash value.

Note one more important detail here: we may have to add some constraints in Program 4 to 6 that reflect the solutions obtained for previous functions in the sequence. That is, if the previous function in the sequence is using $w'$ functions from the family of field $f^{(1)}$ and $u'$ hash functions from the family of field $f^{(2)}$, on each table, we need to apply constraints $w \geq w'$ and $u \geq u'$. Those constraints are related to the incremental computation property of the sequence of clustering functions: there are already $w'$ plus $u'$ hash values computed for each table, so, ideally, we want to use all of them for the next function in the sequence.

## C.2 OR rules

An OR rule for records of two fields states that two records $r_1 = \{f_1^{(1)}, f_1^{(2)}\}$ and $r_2 = \{f_2^{(1)}, f_2^{(2)}\}$ refer to the same entity if

$$d(f_1^{(1)}, f_2^{(1)}) \leq d_{thr}^{(1)} \quad OR \quad d(f_1^{(2)}, f_2^{(2)}) \leq d_{thr}^{(2)}$$

For an OR rule, the AND-OR hashing scheme used by function $H_i$ in the sequence, has hash tables that involve only field $f^{(1)}$ and tables that involve only field $f^{(2)}$. Assuming a $(w, z)$-scheme is used for field $f^{(1)}$ and a $(u, v)$-scheme is used for field $f^{(2)}$, the probability of two records at a distance $x_1$ on field $f^{(1)}$ and $x_2$ on field $f^{(2)}$, hashing to the same bucket in any of the $z + v$ tables, is:

$$1 - \left[1 - p_1^w(x_1)\right]^z \left[1 - p_2^u(x_2)\right]^v$$

To decide the values $w, z, u, v$, for a given *budget* of hash functions, we use the following program:

$$\min_{w,z,u,v} \quad \int_0^1 \int_0^1 \left[1 - \left[1 - p_1^w(x_1)\right]^z \left[1 - p_2^u(x_2)\right]^v\right] dx_1 dx_2 \quad (7)$$

$$s.t. \quad w * z + u * v = budget \quad (8)$$

$$1 - \left[1 - p_1^w(x_1)\right]^z \geq 1 - \epsilon, \ x_1 \leq d_{thr}^{(1)} \quad (9)$$

$$1 - \left[1 - p_2^u(x_2)\right]^v \geq 1 - \epsilon, \ x_2 \leq d_{thr}^{(2)} \quad (10)$$

## C.3 Weighted average rules

Handling weighted average rules requires a slightly different approach compared to the AND and OR rules.

A weighted average rule uses a list of weights $\alpha_1, \ldots, \alpha_F$ ($\sum_i \alpha_i = 1$), for records of $F$ fields, and a single distance threshold $d_{thr}$. Two records $r_1 = \{f_1^{(1)}, \ldots, f_1^{(F)}\}$ and $r_2 = \{f_2^{(1)}, \ldots, f_2^{(F)}\}$ refer to the same entity if

$$\bar{d}(r_1, r_2) = \sum_{i=0}^{F} \alpha_i d(f_1^{(i)}, f_2^{(i)}) \leq d_{thr}$$

For a weighted average rule a $(w, z)$-scheme is used for function $H_i$ in the sequence, just like in the case of a single field: the values for parameters $w$ and $z$ are chosen based on the process described in Section 5.1. Nevertheless, there is one important difference compared to the single field case. In order to select each of the $w$ hash functions, for each of the $z$ hash tables, the following process is used:

DEFINITION 7 (Weighted-Average Function Selection) *(a) randomly select one of the $F$ fields based on the distribution defined by the field weights $\alpha_1, \ldots, \alpha_F$, i.e., the probability of picking field $i$ is $\alpha_i$, and (b) select uniformly at random one of the hash functions from the locality-sensitive family for the selected field $i$.*

The process of Definition 7 has the theoretical properties summarized in the following two theorems.

THEOREM 3 *For each field $i$, consider a locality sensitive family $\mathcal{F}^{(i)}$, such that the probability of selecting a hash function $h_j \in \mathcal{F}^{(i)}$ with $h_j(r_a) = h_j(r_b)$, for any two records $r_a$ and $r_b$, is:*

$$Pr[h_j(r_a) \equiv h_j(r_b)] = 1 - d(f_a^{(i)}, f_b^{(i)})$$
$$\text{where } \ 0 \leq d(f_a^{(i)}, f_b^{(i)}) \leq 1$$

*If $h'_j$ is a hash function selected using the process of Definition 7, then:*

$$Pr[h'_j(r_a) \equiv h'_j(r_b)] = 1 - \bar{d}(r_a, r_b)$$

PROOF: The probability of selecting a field $i$ in step (a) of the process is $Pr[\textit{field i picked}] = \alpha_i$. Moreover, if field $i$ is picked then $Pr[h'_j(r_a) \equiv h'_j(r_b)] = [1 - d(f_a^{(i)}, f_b^{(i)})]$. Therefore,

$$\begin{aligned}
Pr[h'_j(r_a) \equiv h'_j(r_b)] &= \sum_{i=0}^{F} Pr[\textit{field i picked}][1 - d(f_a^{(i)}, f_b^{(i)})] \\
&= \sum_{i=0}^{F} \alpha_i [1 - d(f_a^{(i)}, f_b^{(i)})] \\
&= \sum_{i=0}^{F} \alpha_i - \sum_i \alpha_i d(f_a^{(i)}, f_b^{(i)}) \\
&= 1 - \bar{d}(r_a, r_b) \qquad \square
\end{aligned}$$

An example where Theorem 3 applies is the family of minhash functions for the Jaccard distance. A more general version of Theorem 3 is stated in Theorem 4:

THEOREM 4 *For each field $i$, consider a $(d_{thr}, \rho d_{thr}, p_1^{(i)}, p_2^{(i)})$-sensitive family. In this case, the family of functions $\mathcal{F}'$, where each function $h'_j \in \mathcal{F}'$ is selected using the process of Definition 7, is $(d_{thr}, \rho d_{thr}, \sum_{i=0}^{F} \alpha_i p_1^{(i)}, \sum_{i=0}^{F} \alpha_i p_2^{(i)})$-sensitive.*

PROOF: The proof is similar to the one of Theorem 3. $\square$

## C.4 Combining rules

In the last part of this appendix, we briefly discuss the case of ER rules that combine AND, OR, and weighted average rules. In this case, we need to combine the processes described in the previous parts of this section. To select the number of hash functions coming from the locality-sensitive hashing family of each field, we need to solve more general optimization programs compared to the ones discussed before. Nevertheless, the main principle in those programs is the same with the ones we discussed: the probability of hashing to the same bucket should be very close to 1.0 for pairs of records that satisfy the combined ER rule (e.g., Equation 6) and the overall volume under the probability curve should be minimized (e.g., Equation 4).

The more fields are involved in the ER rule, the more parameters are involved in the optimization program, and the more computationally heavy it is to solve the program. In practice, this is not an issue, however, for two reasons. First, the whole function sequence design process is run offline, before Adaptive LSH is applied on a dataset and the same sequence design usually suffices for many similar datasets. Second, depending on the program, an exhaustive search over all parameter values can often be avoided (e.g., binary search for Program 1 to 3).

## D. LARGEST-FIRST OPTIMALITY

In this Appendix, we give the full proofs for Theorems 1 and 2 (Section 4.2) and discuss when it could make sense for an algorithm *not* to follow the largest-first optimality assumptions.

## D.1 Proofs

THEOREM 1 *Consider the family of algorithms that:*
1. *do* not *"jump ahead" to function $P$, i.e., if a cluster $C$ is an outcome of a function $H_i$, the algorithm can only apply function $P$ on $C$, when $(cost_{i+1} - cost_i) * |C| \geq cost_P * \binom{|C|}{2}$ (Line 5 on Algorithm 1).*
2. *do* not *"terminate early", i.e., terminate only when the $k$ largest clusters are an outcome of either an $H_L$ or $P$ function.*

*Algorithm 1 gives the minimum overall cost compared to any other algorithm of this family.*

PROOF: We will prove that Algorithm 1 gives the minimum overall cost compared to any other algorithm, for any *execution instance*. In an execution instance, the outcome of applying a function $H_i$ or $P$ on a set of records $S$, is the same across all algorithms. In other words, all algorithms would observe the exact same clusters during their execution if they would select the same cluster to process in each step. Assume that another algorithm $\mathcal{B}$ in this family, gives a lower overall cost than Algorithm 1, for a given execution instance. Based on the cost model (Definition 3), for algorithm $\mathcal{B}$ to have a lower overall cost than Algorithm 1, there are three possibilities:
1. there must be a set of records $S_1$ such that: both Algorithms 1 and $\mathcal{B}$ apply $P$ on $S_1$, but the last function Algorithm 1 applies on $S_1$, before $P$, is $H_i$, and the last function applied on $S_1$ by $\mathcal{B}$, before $P$, is $H_j$ for $j < i$.
2. there must be a set of records $S_2$ such that: algorithm $\mathcal{B}$ applies $P$ on $S_2$ and the last function Algorithm 1 applies on $S_2$, is $H_i$, while the last function algorithm $\mathcal{B}$ applies on $S_2$, before $P$, is $H_j$ for $j < i$.

3. there must be a set of records $S_3$ such that: the last function Algorithm 1 applies on $S_3$ is either $P$ or $H_i$, and the last function $\mathcal{B}$ applies on $S_3$ is $H_j$ for $j < i$.

The first two possibilities violate Condition 1, in the definition of the family of algorithms, since they would require algorithm $\mathcal{B}$ to "jump ahead" to function $P$; otherwise, Algorithm 1 would apply the exact same functions on sets $S_1$ or $S_2$. Hence, we focus on the third possibility. Consider step $l$, where Algorithm 1 selects set $S_3$ (or a cluster that is a subset of $S_3$), to apply function $H_{j+1}$ (where $H_j$ is the last function algorithm $\mathcal{B}$ applies on $S_3$). At step $l$, set $S_3$ is the largest cluster for Algorithm 1 to select it. Since clusters always split in subsequent steps, $S_3$ will also be larger than the top-$k$ clusters after the final step of Algorithm 1. Hence, since we are focusing on the same execution instance, Condition 2 in the algorithms' family definition is violated, as the largest cluster after the final step of algorithm $\mathcal{B}$ will *not* be an outcome of an $H_L$ or $P$ function. $\square$

THEOREM 2 *For an input $k$, Algorithm 1 reaches to a state where the $k'$ largest clusters are an outcome of either an $H_L$ or $P$ function, for any $k' < k$, with a lower cost compared to any other algorithm in the the family defined in Theorem 1.*

PROOF: For a $k' < k$, consider applying Algorithm 1 with input $k'$ and with input $k$. Assume the same execution instance in both cases. Note that Algorithm 1 will execute in the exact same way for both inputs, until it terminates when applied with input $k'$. That is, Algorithm 1 will process the same clusters using the same functions and in the same order, in both cases, before the $k'$ largest clusters are an outcome of either an $H_L$ or $P$ function. We know from Theorem 1 that Algorithm 1 finds the $k'$ largest clusters with a minimum cost when applied with input $k'$. Since Algorithm 1 executes in the same way when applied with input $k$, we conclude that Algorithm 1 reaches to a state where the $k'$ largest clusters are an outcome of either an $H_L$ or $P$ function, with a minimum cost. $\square$

## D.2 Assumptions

Let us start with the second assumption and consider algorithms that *do* "terminate early". (That is, algorithms that may terminate even when the $k$ largest clusters are not all an outcome of either an $H_L$ or $P$ function.) In that case, the algorithm would either: (a) output a cluster which is not an outcome of an $H_L$ or $P$ function, or (b) would *not* output one of the $k$ largest clusters. In case (a), the algorithm should be fairly certain that this cluster would not split into smaller clusters if function $P$ (or $H_L$) was applied on it. In case (b), the algorithm should be fairly certain that the cluster not contained in the output, would split into small (smaller than the $k$ largest clusters that are an outcome of an $H_L$ or $P$ function) clusters. Thus, an algorithm should have a good estimation of how likely it is for clusters to split, if more functions in the sequence were to be applied on them, and how large the new clusters would be.

A good estimation of how likely are clusters to split and how large the new clusters would be, is also the key condition for an algorithm to potentially benefit from breaking the first assumption. Seeing how it could be beneficial to break the first assumption, is a bit more complicated. Let us illustrate with a simple example. Consider two clusters, $C_1$ of 10 records and $C_2$ of 12 clusters, and assume we are looking for the top-1 entity. Moreover, assume that: (a) $C_1$ either does not split at all, with 50% probability, or splits into two clusters of 5 records, with 50% probability, (b) $C_2$ either does not split at all, with 5% probability, or splits into two clusters of 9 and 3 records, with 50% probability, or splits into four clusters of 3 records each, with 45% probability, and (c) to find out if $C_1$ splits, we need to apply function $P$ on it, while for $C_2$, we can apply the next sequence-function, to find out if it splits, but we need to apply function $P$ to find out if it splits to four clusters or two clusters. In this example, it can be beneficial to first apply function $P$ on the smaller cluster $C_1$ first. If cluster $C_1$ splits into two clusters of 5 clusters each, then it only makes sense to directly apply $P$ on $C_2$ to find out if the largest cluster in $C_2$ consists of 9 or 3 records. (Note that the largest-first strategy would first apply the next sequence-function on $C_2$, so we would not be able to avoid the cost of that function on $C_2$.) Potentially, such a strategy could lead to a lower execution time compared to largest-first.

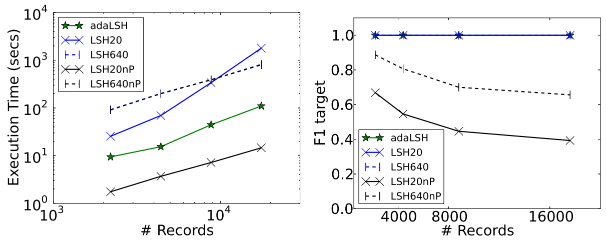
The bottom line is that an algorithm could benefit from breaking the two assumptions, only when it keeps estimates of the sizes of sub-clusters inside each cluster. Computing accurately such estimates may not always be possible, or may be so costly, in terms of execution time, that the overhead outweighs the benefits. We plan to investigate in future research if this could be a direction giving a non-trivial improvement.

## E. ADDITIONAL EXPERIMENTS

### E.1 LSH blocking variations

In this appendix, we discuss a setting that illustrates the tradeoff between accuracy and performance when we run only the first stage of LSH. To measure accuracy in this case we use a metric called F1 target: we consider as ground truth the outcome of function $P$ on the whole dataset and we compute the harmonic mean of precision and recall, just like we do for F1 Gold. The purpose of this metric is to quantify the errors introduced by the probabilistic nature of the LSH approaches.

We use four LSH variations: LSH20, LSH20nP, LSH640, and LSH640nP. (As discussed in Section 6.1, the nP variations do *not* apply function $P$ after the first stage.) Figure 20(a) depicts the results, for $k = 10$ on SpotSigs, SpotSigs2x, SpotSigs4x, and SpotSigs8x. We see that adaLSH gives an, at least, 4x speedup against all variations of LSH, besides LSH20nP.



(a) Execution time (b) F1 target

**Figure 20: LSH vs adaLSH, performance/accuracy**

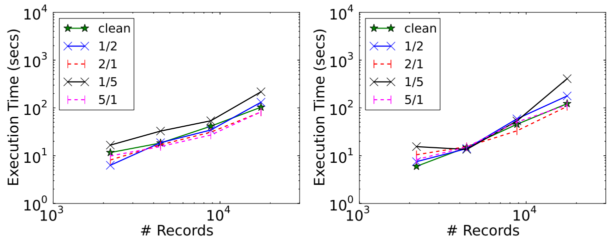
Of course, the nP variations, and especially LSH20nP, are much less accurate than the other methods as Figure 20(b) shows: the F1 Target is just 0.7 for LSH20nP on SpotSigs and drops to 0.4 on SpotSigs8x, while for LSH640nP the F1

Target drops from 0.9 to below 0.7. On the other hand, all other methods give an F1 Target very close to 1.0.

There is another interesting perspective on the results from Figure 15(a). Note that the computation performed by LSH20nP is actually the computation that adaLSH performs in the first round; as discussed in Section 6.1, in the adaLSH used in the experimental evaluation, the first function in the sequence applies 20 hash functions on all the records. As Figure 15(a) shows, the overall computation adaLSH performs takes just 5 to 10 times more than the computation it performs on the first round.

## E.2 adaptive LSH Tuning

Here, we discuss a couple of experiments regarding the fine tuning of adaptive LSH. First, we add noise to the simple cost model used by Algorithm 1 in Line 5. In particular, we multiply by a noise factor $nf$, the cost of applying the pairwise computation function $P$ on a cluster $C$: $cost_P * \binom{|C|}{2}$. That is, when factor $nf$ is less than one, the cost of applying $P$ is under-estimated and $P$ is applied sooner (and on larger clusters) compared to when no noise is added. On the other hand, when factor $nf$ is greater than one, the cost of applying $P$ is over-estimated and the application of $P$ is deferred until clusters are small enough.



(a) $k = 2$ (b) $k = 10$

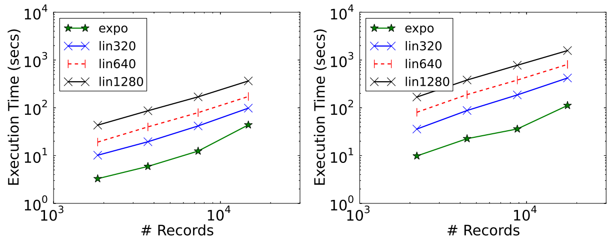
**Figure 21: Adding noise to the cost model.**

We tried four values for factor $nf$: $\frac{1}{2}, \frac{1}{5}, \frac{2}{1}$, and $\frac{5}{1}$. In Figure 21(a), the y-axis shows the execution time for $k = 2$, on SpotSigs, SpotSigs2x, SpotSigs4x, and SpotSigs8x (x-axis). In Figure 21(b), we run the same experiment for $k = 10$. Each curve refers to a different value for factor $nf$. We also plot the execution time for adaptive LSH without any noise added ("clean" curve). (Note that parameters $cost_P$ and $cost_i, 1 \leq i \leq L$, are estimated using 100 samples each.)

We draw one main conclusion from the plots of Figures 21(a) and 21(b): adaptive LSH is not sensitive to cost-model noise and the execution time may only be significantly affected for a very small $nf$ of $\frac{1}{5}$. That is, there is a considerable increase in the execution time for adaptive LSH, only when the cost of applying $P$ is heavily under-estimated and $P$ ends up being applied early and on larger clusters compared to when no (or a little bit of) noise is added.

The second experiment in this appendix, studies the different modes for budget selection, discussed in Section 5.2. We try:

- expo: the default Exponential mode, where the budget is doubled for every function in the sequence, starting from 20 hash functions for the first function.
- lin320, lin640, lin1280: the Linear mode, where the budget starts from 320, 640, or 1280 hash functions, for the first function, and is increased by 320, 640, or 1280 hash functions, for every function in the sequence.



(a) Cora (b) SpotSigs

**Figure 22: Different budget selection modes.**

Figure 22(a) shows the execution time for the four modes in the y-axis, on Cora, Cora2x, Cora4x, Cora8x (x-axis), for $k = 10$. Figure 22(b) refers to the same experiment for SpotSigs.

Clearly, the Exponential mode is the best option requiring a far lower execution time compared to other modes. Note that, in the Exponential mode (when the budget is doubled for every function in the sequence), the "amount" of processing performed on the selected cluster in each step, is almost the same with the amount of processing performed in all previous steps, on the records of the selected cluster. Hence, the Exponential mode is able to find the sweet spot in the trade-off between running fewer hash functions, overall, in many steps and running more hash functions in fewer steps.