# 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 (LSH), 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 web 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 cases, the popularity of different entities is not the same: few entities collect a large number of records referring to them, while for most entities there are is a single or a couple of records referring to them. Moreover, there are many applications that only need to find those few entities with the large number of records referring to them, and do not need all other less popular entities. Let us illustrate a few such applications.

First, consider an application that collects answers/solutions from forums regarding software bugs. In this case, the records are the forum threads and the entities are the different software-

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

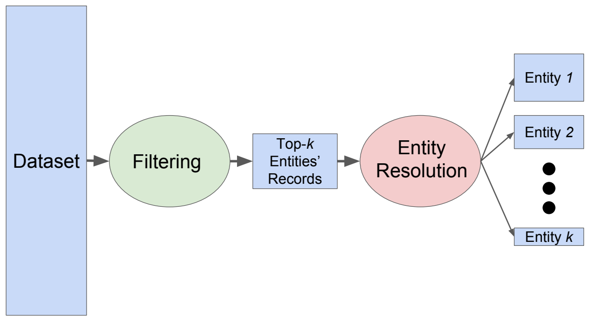


**Figure 1: Filtering stage in the overall workflow.**

bug cases. The objective here is to improve the quality of answers by constructing a single answer (or finding the best answer) to each distinct software-bug case. Constructing a single aggregated answer possibly requires human effort, hence, such an application can only focus on the few most popular software bugs; that obviously impact a large number of users. (After all, for the vast majority of bug cases with a single or a few forum threads discussing them, users can just go over those discussions and find the best solution to the issue.)

Second, consider an application that finds the different opinions on social media for a specific topic. For example, the topic can be "NBA Finals 2016 Game 7" and the top-2 opinions (expressed in most posts) could be "Steph Curry choked in the last minutes" and "Warriors need to sign Kevin Durant". Here, the records would be the different posts in social media and the entities the different opinions. Clearly, the users of this application are not interested in all the opinions for each topic (and cannot go over all opinions) but want to get a feeling about what most people think by looking at the top opinions.

Overall, we can find a motivating example for any application where the user's experience drastically improves by the application being aware of the records/entries referring to the top entities and possibly cleaning/aggregating the information for those entities (e.g., popular questions in search engine query logs, viral videos in video streaming engines, suspicious individuals that appear very often in specific locations of an airport). In addition, finding the entities that collect a large number of records referring to them, may be very useful from a data analytics perspective: for a given application understanding which entities collect many entries and why, can lead to changes in the functionality/interface that can improve the overall user experience.

The naive approach for finding the largest entities, in terms of number of records referring to them, is to first apply an ER algorithm to find all entities, and then output only the largest entities. However, most ER algorithms require the computation of pairwise similarities for every two records in a dataset. We can expect that the applications where finding the largest entities is important, generate large datasets and the cost of computing the similarity of every two records in large datasets is prohibitive. In the previous "NBA Finals 2016 Game 7" example, we can expect the dataset to consist of at least 100 thousand posts/messages: 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 very large datasets, by having a linear cost to the number of records in the dataset. Since the filtering stage output is expected to be orders of magnitude smaller than the whole dataset, the ER algorithm can afford a quadratic (or even higher) cost to the size of the input, or even involve human curation.

Various different aspects of entity resolution have been studied over the years [17, 38, 36, 19, 10]. The most related topic to the problem studied here, is the one of blocking [25, 6, 8, 26, 18, 12, 13]: 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) [20] 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.

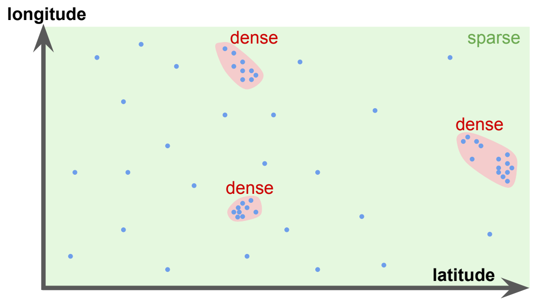


**Figure 2: Key insight in Adaptive LSH.**

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 Section 6 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, as discussed 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 $(g_j : S \rightarrow \{C_i\})_{j=1}^{L}$. 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 $g_2$.

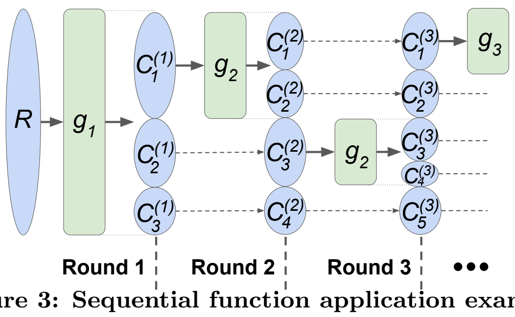


**Figure 3: Sequential function application example.**

The sequential function application stops when the $k$ largest clusters, in the list of 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 be not be identical to the ground truth output $\mathcal{O}^*$, as the objective is a high-performance filtering stage 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 in the dataset 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 the "sparse" areas will have a few functions applied on, and the full sequence of functions will only be applied on the records lying on the "dense" areas. Hence, the amount of processing applied on the vast majority of 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 $g_1$ (when applied on $R$), and does not need to be repeated.

## 2.4 Locality-Sensitive Hashing

The third key concept in our approach is using Locality-Sensitive Hashing (LSH) [20] as the main component of the sequence of clustering functions. We give an overview of LSH and we 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 vector. Consider as a distance function 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 $\binom{|R|}{2}$ pairwise 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:

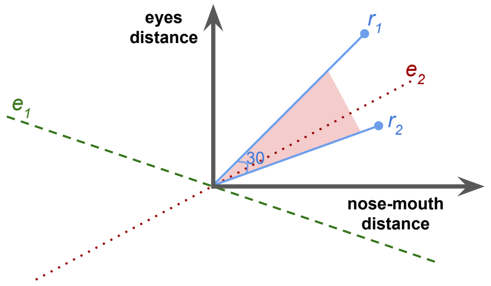


**Figure 4: Random hyperplanes example.**

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 the concatenation of the outcome from a number of hash functions. The following example illustrates the hash tables built by LSH:

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: $1 - \left(1 - (1 - \frac{\theta}{180})^3\right)^2$.*

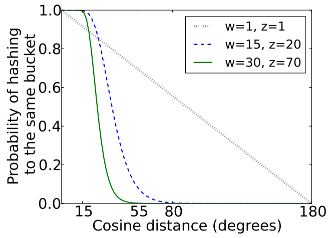


**Figure 5: Probability of hashing to the same bucket**

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 always be very close to one 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 zero as possible (details in Section 5.1).

The plot of Figure 5 illustrates the probability of hashing to the same bucket, in at least one table, using the setting of Examples 1 to 3. The x-axis shows the angle distance between two records and the y-axis the probability for three $w, z$ value pairs: 1) $w = 1, z = 1$, 2) $w = 15, z = 20$, and 3) $w = 30, z = 70$.

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 attempt to 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*: the computation of the functions can be performed incrementally, as the hash values from previous functions in the sequence are re-used by the functions that follow.

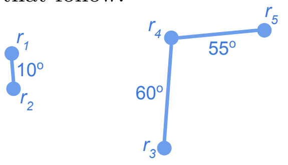


**Figure 6: Transitive Hashing example.**

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

In this section, we describe the algorithm of the Adaptive LSH approach outlined in Section 2, and we prove the optimality of the largest-first selection rule.

### 4.1 Algorithm

The input and output of the algorithm are:

**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.

**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 cluster $C$
 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
```

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.

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 the pairwise computation 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 of records $S$, is $cost_i * |S|$. Moreover, the cost of applying function $H_i$ in the sequence on a record $r$, when function $H_j$, $j < i$, is already applied on record $r$, 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 pairwise similarities computed by function $P$.*

In Section 6.10, 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. In Line 1, the algorithm applies the first function in the sequence, $H_1$, to all records. Then, in successive rounds, the largest cluster $C$ from the previous round is selected (Line 3), and the algorithm applies a transitive hashing function or the pairwise computation function on $C$, taking into account the cost model (Lines 3 to 9). The algorithm terminates when the $k$ largest clusters are an outcome of the last function in the sequence or an outcome of the pairwise computation function (Line 12).

## 4.2 Largest-First Optimality

In this section, we prove optimality for the Largest-First strategy of Algorithm 1, after stating the optimality assumptions, in Theorem 1. We discuss in Appendix D, the cases where it could make sense for an algorithm *not* to follow these assumptions.

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 $S_1$, 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. □

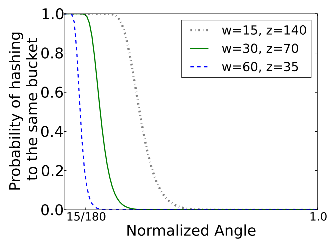


**Figure 7: Three example value pairs for parameters $(w, z)$ of Program 1 to 3.**

## 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 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:

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

$$s.t. \quad w * z = budget \tag{2}$$

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

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; as Figure 7 also points out. Moreover, if the distance threshold constraint is not satisfied for a value of $w$, it will not be satisfied for any greater values as well.

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 the experimental evaluation, in Section 6.11, we try different parameter values for the two strategies and we draw conclusions regarding which strategy and values work better in each case.

## 6. EXPERIMENTAL EVALUATION

In our experiments we use datasets of web articles [2], scientific publications [1], and images [3].

## 6.1 Metrics

We compare filtering-stage approaches that output a set records, as depicted in Figure 1. Hence, the execution time of each approach does not include the application of any specific entity resolution algorithm, after the filtering stage. To quantify how accurate each approach is, we compare the approach's output to the ground truth output $\mathcal{O}^*$, as discussed in Section 2.1. In addition, we use one more metric to quantify the errors introduced by the probabilistic nature of adaptive LSH and the other LSH approaches: we run the pairwise computation function $P$ (Definition 2) on the whole dataset, and we compare the set of records in the top-$k$ clusters function $P$ gives, to an approach's output. In particular, we use the following five metrics:

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

**Precision Gold**: consider the set of records in the ground truth top-$k$ entities and the set of records in the output of each method. The gold precision is the percentage of output records that belong to the ground truth set of records.

**Recall Gold**: the percentage of records in the ground truth top-$k$ entities that belong to the output of a method.

**F1 Gold**: the harmonic mean of precision and recall, i.e., $\frac{2*p*r}{p+r}$, where $p$ is the gold precision and $r$ the gold recall.

**F1 Target**: in this case 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.

## 6.2 Datasets

We used three datasets in our experiments:

***Cora* [1]**: a dataset of around 2000 scientific publications, extensively used in the entity resolution literature. Each record consists of the title, the authors, the venue, and other related information regarding the publication (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 and uniformly at random pick a record from the selected entity, for each record added to the original dataset. Since each record has multiple fields, we use 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 distance 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 [31]. 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). For each image, we extract 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 three thresholds in the experiments, $2, 3$, and 5 degrees.

## 6.3 Methods

We compare adaptive LSH (adaLSH) with the predominant alternative on high-dimensional data: blocking using traditional LSH. We try different variations of the traditional LSH approach and we also try a baseline transitive closure algorithm (Pairs) that computes the exact pairwise distances between the records in the dataset.

***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***: The traditional LSH blocking approach adjusted for the problem studied in this paper. In particular, 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 keep the comparison fair, we use three additional optimizations for LSH: (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). We also tried a variation of LSH, where we only apply the first stage and do 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 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.

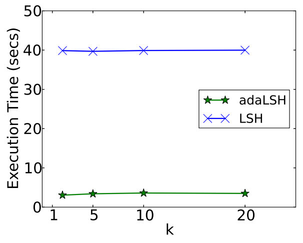
## 6.4 Findings' Overview

Below we provide an overview of our main findings:

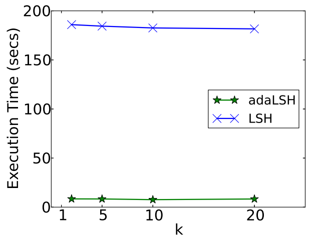
- adaLSH gives a 2x to 25x speedup compared to traditional LSH, depending on the dataset. (The speedup compared to Pairs can become arbitrarily large, as the size of the dataset increases.)
- the value of $k$ only slightly affects the execution time for adaLSH. In particular, the execution time for adaLSH 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-$k$ represent less than 10% of all records).
- adaLSH always gives the same (or a very slightly different) outcome with Pairs. Thus, adaLSH only introduces minimal errors due to its probabilistic nature. Nevertheless, this outcome can be considerably different from the ground truth, in cases where a suboptimal distance threshold(s) is used. Even in those cases, we can increase the recall by having adaLSH (or LSH) return the records from more than $k$ clusters: while the precision drops, the size of the output increases by only a factor of 2 to 3 compared to the set of records in the ground truth top-$k$ entities, in the cases we studied.
- we tried to find if there is a specific variation for LSH that performs as good as adaLSH, in terms of the execution time, in each experimental setting. As expected, there is usually a different best LSH variation in each different setting. More interesting however, is the fact that adaLSH always gives an important speedup compared to the best LSH variation, in any setting.
- we added noise in the cost model (Definition 3) used in adaLSH. It appears that even when the estimations for the cost of applying a hash function on a record and the cost for computing a pairwise similarity are not very accurate, the execution time of adaLSH is just slightly affected.
- 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 an important speedup compared to each LSH variation for at least one distribution.

## 6.5 Different k values

We start by examining the execution time of adaLSH and LSH on Cora, for different values of $k$, i.e., number of top entities. Our objective here is to assess how much additional overhead is induced, as we increase the number of entities that must be retrieved. We run experiments for $k = 2, 5, 10$, and 20, and we use LSH1280; we try other variations for LSH in Section 6.8. In the plot of Figure 8(a), the x-axis shows the $k$ value, and the y-axis the execution time. Interestingly, 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. More interesting is the 10x speedup compared to LSH, for any $k$ value. LSH applies 1280 hash functions on all records, while adaLSH starts by applying 20 hash functions on all records and then adaptively decides which records to process further.



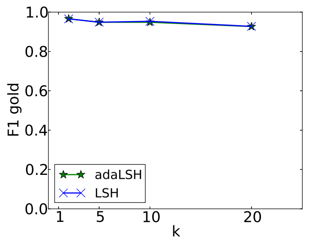
(a) Cora



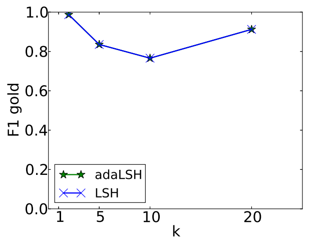
(b) SpotSigs

**Figure 8: Execution time for different $k$ values.**

Let us now compare the results from Figure 8(a) to the results for the same experiment on the SpotSigs dataset, in Figure 8(b). The main difference in this case is that the cost of applying a hash function increases in this dataset, 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, giving an impressive speedup of 25x.



(a) Cora



(b) SpotSigs

**Figure 9: F1 Gold for different $k$ values.**

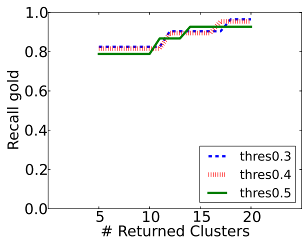
## 6.6 Accuracy

Now let us examine how accurate the outcome of adaLSH and LSH is. The plots in Figures 9(a) and 9(b) give the F1 Gold for the same experiments of Figures 8(a) and 8(b), on Cora and SpotSigs, respectively.

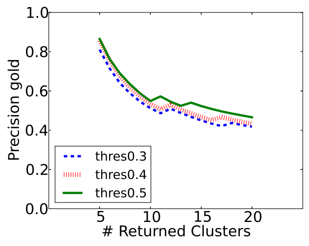
Both methods give a very similar F1 score, as they both compute clusters that are almost identical to the ones Pairs would produce. Hence, the probabilistic nature of the two methods does not introduce errors. (We will see in Section 6.8 that some other variations of LSH do, however, introduce errors.) Still, the F1 score in case of SpotSigs in Figure 9(b) is low (around 0.8) for $k = 5$ and 10, meaning that transitive closure applied with this specific distance threshold does not give a very accurate outcome.

To handle such situations where the outcome may not be sufficiently accurate, we can increase the number of clusters that the methods return. Remember that the main goal of the filtering stage is to reduce the size of the initial dataset. Therefore, by increasing the size of the methods' output, we can get a recall that is very close to 1.0, while still returning an output that is just a few times larger than the size of the ground truth output; i.e., the set of records in the ground truth top-$k$ entities.

To illustrate, we focus on a $k$ value of 5 for SpotSigs, where the F1 score is just above 0.8, as we saw in Figure 9(b). In the experiment of Figure 10(a), the x-axis shows the number of clusters we ask a method to return. For example, for an x-axis value of 10, the method returns the records for the 10, instead of 5, largest clusters found. 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 10(a) shows the recall 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, there are three curves in Figure 10(a): one for each of the three similarity thresholds we tried, 0.3, 0.5, and the default 0.4.



(a) Recall



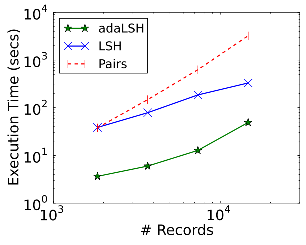
(b) Precision

**Figure 10: Precision/Recall on SpotSigs for $k = 5$.**

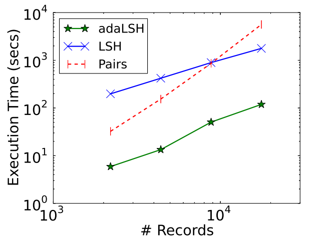
The recall for all thresholds is almost the same and, more importantly, follows the same trend: recall keeps going up as we include more clusters in the output, to reach very close to 1.0 for 20 clusters.

It is important to note here that the same effect of increasing the recall, would not necessarily hold if we would relax 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 cluster that will become the largest one. However, the new largest cluster may now contain 100% of the records of the ground truth top-2 entity, but none of the top-1 entity records. Hence, the recall would drop from 80% to 0% if we would relax the threshold, in this case.

By increasing the size of the output, the precision inevitably drops. Figure 10(b) shows the precision for the same setting with Figure 10(a). As we increase the number of clusters returned from 5 to 20, the precision drops from 80% to almost 40%. Nevertheless, even for a precision of 40%, the size of the output is only $\frac{1}{0.4} = 2.5$ times larger than the ground truth output.



(a) Cora



(b) SpotSigs

**Figure 11: Execution time for different dataset sizes.**

## 6.7 Different dataset sizes

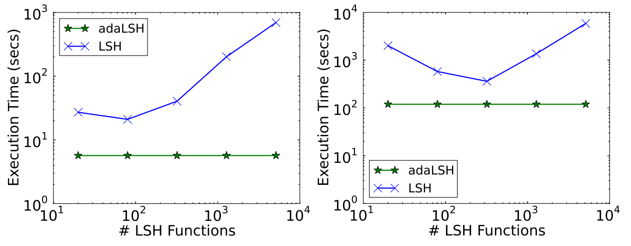
Let us now study the effect of the dataset size in the performance of the different methods. The log-log plot in Figure 11(a) shows the number of records in the dataset on the x-axis and the execution time on the y-axis, for Cora,

Cora2x, Cora4x, and Cora8x. We use a $k$ value of 10 in this experiment; same results hold for other $k$ values.

We observe that there is always a large speedup for adaLSH compared to LSH (we use LSH1280 as before), ranging from 9x to 20x, on the different dataset sizes. As the dataset size increases, the speedup of adaLSH compared to Pairs keeps increasing (e.g., 60x on Cora8x).

Now, let us switch to the SpotSigs dataset: Figure 11(b) shows the execution time for SpotSigs, SpotSigs2x, SpotSigs4x, and SpotSigs8x. As mentioned in Section 6.5, the cost of applying a hash function in SpotSigs is higher than in Cora. This causes LSH to run slower on SpotSigs than the baseline Pairs: LSH needs the dataset to be at least 9000 records to show a better performance than Pairs.

On the other hand, adaLSH gives an important 5x speedup over Pairs even for a small dataset size of 2000 records, that increases to 50x for SpotSigs8x. The speedup compared to LSH is even greater here than in Cora, ranging from 15x to 25x.



(a) SpotSigs (b) SpotSigs8x

**Figure 12: LSH vs adaLSH, for different LSH variations.**

## 6.8 LSH vs adaLSH

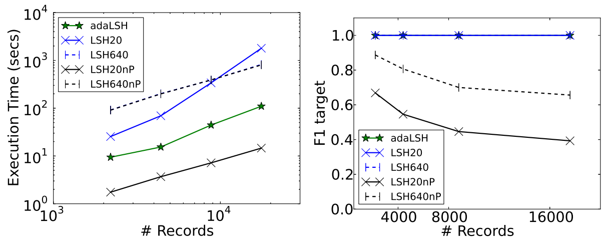
Until now we used only LSH1280 in the plots. Here, we dig into the different variations of LSH. Clearly, knowing in advance which LSH variation is better in each case is not possible. Our goal here is to examine how much better adaLSH is, compared to the best LSH variation, if we knew in advance the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

We start by plotting the execution time in Figure 12(a), on SpotSigs and $k = 10$, for five LSH variations: LSH20, LSH80, LSH320, LSH1280, and 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 12(b), for SpotSigs8x and $k = 10$, LSH320 is now the lowest execution time variation. Still, adaLSH gives a 3x speedup compared to LSH320.

We tried one more setting in this section, that illustrates the tradeoff between accuracy and performance when we run only the first stage of LSH. Hence, we tried four variations: LSH20, LSH20nP, LSH640, and LSH640nP. (As discussed in Section 6.3, the nP variations do *not* apply function $P$ after the first stage.) Figure 13(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.

Of course, the nP variations, and especially LSH20nP, are much less accurate than the other methods as Figure 13(b)



(a) Execution time (b) F1 target

**Figure 13: LSH vs adaLSH, performance/accuracy**

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.

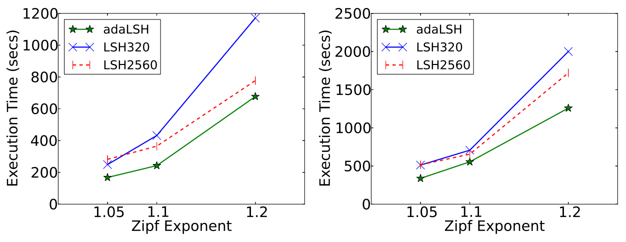
We close this section with another interesting perspective on the results from Figure 12(a). Note that the computation performed by LSH20nP is actually the computation that adaLSH performs in the first round; as discussed in Section 6.3, in the adaLSH used in the experimental evaluation, the first function in the sequence applies 20 hash functions on all the records. As Figure 12(a) shows, the overall computation adaLSH performs takes just 5 to 10 times more than the computation it performs on the first round.

## 6.9 Entity sizes' distribution

We now switch to the PopularImages dataset. Our objective here is *not* to illustrate how large the adaLSH speedup can be, but instead study a case where the "sparse" areas (areas with a few records) in the dataset are limited. That is, 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 more efficient ways of performing this task, 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. For example, how the existence of entities with a very large number of records referring to them, affects the execution time.

The three datasets in PopularImages, follow zipfian distributions, with exponents of 1.05, 1.1, and 1.2, respectively. Hence, in the 1.05-exponent dataset, the top-1 entity consists of around 500 records, the top-2 entity of around 250 records, the top-3 of around 150 records, and so on. In the 1.1-exponent dataset, the top-1 entity consists of around 1000 records, top-2 entity of around 400 records, and the top-3 entity of around 300 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.



(a) $d_{thr} = 3$ degrees (b) $d_{thr} = 5$ degrees

**Figure 14: Execution time on PopularImages.**

Figure 14(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. Applying Pairs on each of these datasets takes almost one hour and we do not include it in this plot.

Even in this, far from ideal, scenario, adaLSH gives an important 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, a 1.5 speedup for an exponent of 1.1, while for an exponent of 1.2 adaLSH is just slightly better.

The results for a threshold of 5 degrees appear in Figure 14(b). The adaLSH speedup ranges from 1.2 to 1.5 compared to LSH2560 and from 1.3 to 1.6 for LSH320.

Besides the adaLSH speedup, there are a couple more interesting points in Figures 14(a) and 14(b). The execution time for both thresholds increase 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 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 14(a) and 14(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 that LSH320, which clearly underperforms here, was (together with LSH80) the most effective LSH variation in the experiments, in Figures 12(a) and 12(b). This illustrates, again, that the most effective LSH variation can be very different in different cases. On the other hand, adaLSH always gives a significantly better performance.

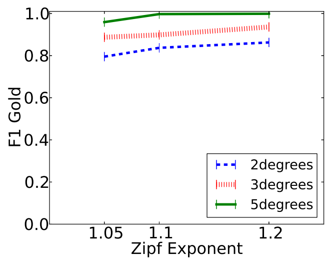
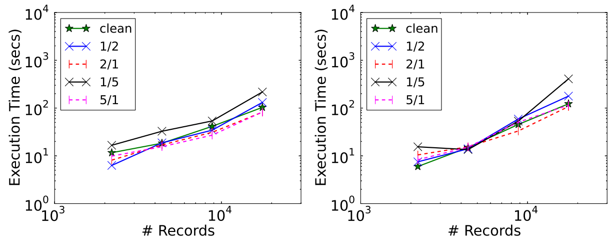


**Figure 15: F1 Gold on PopularImages.**

The last effect we discuss here, is the tradeoff between performance and accuracy, as the distance threshold and the zipfian exponent change. In Figure 15, 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 15, 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.

## 6.10 Adding noise to the cost model

Here, 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 16: 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 16(a), the y-axis shows the execution time for $k = 2$, on SpotSigs, SpotSigs2x, SpotSigs4x, and SpotSigs8x (x-axis). In Figure 16(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 $cost_i, 1 \le i \le L$, are estimated using 100 samples each.)

We draw one main conclusion from the plots of Figures 16(a) and 16(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.

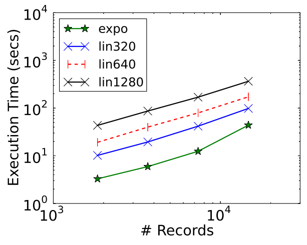
## 6.11 Exponential vs Linear sequences

The last experiment we include in this paper, studies the different modes for budget selection, discussed in Section 5.2. We try:

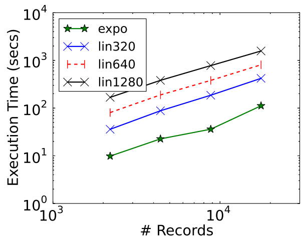
- expo: the default Exponential mode, where the budget is doubled for every function in the sequence, starting from 20 hash functions for the first function.
- lin320, lin640, lin1280: the Linear mode, where the budget starts from 320, 640, or 1280 hash functions, for the first function, and is increased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

Figure 17(a) shows the execution time for the four modes in the y-axis, on Cora, Cora2x, Cora4x, Cora8x (x-axis), for $k = 10$. Figure 17(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



(a) Cora



(b) SpotSigs

**Figure 17: Different budget selection modes.**

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.

## 7. RELATED WORK

We organize the related work in three parts: entity resolution, blocking, and locality-sensitive hashing.

**Blocking:** To enable entity resolution on large datasets, many blocking approaches have been suggested for different settings [25, 6, 8, 26]. Paper [13] 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 [28]. The framework in paper [18], 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 [12] that models the communication-computation trade-off and proposes strategies to distribute the pairwise comparison workload across nodes. In paper [37], 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 [15], a large scale data exploration system. Supervised meta-blocking [29] 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 [17] and [38]. Here, we will try to cover a few more recent studies with a connection to the setting in this paper. Paper [36] 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 [7, 35, 34, 33, 39, 14]. An alternative of using examples, is defining constraints for matching records, through declarative/interactive frameworks [10, 16]. Entity resolution is also studied in settings where the data is distributed across multiple nodes [5], and the goal is to reduce the bandwidth usage while maintaining a low execution time. Incremental ER is the focus in paper [19], where data updates can be handled efficiently and can also provide evidence to fix previous errors.

**Locality-Sensitive Hashing:** Many LSH variations have been studied since the concept of LSH has been proposed [20]. Here, we briefly discuss some of the variations that involve some notion of adaptivity; although quite different from the LSH adaptivity concept introduced in this paper. Multi-Probe LSH [24] 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 [27, 22], which trades time for space requirements, for nearest-neighbor search on Euclidean spaces. Bayesian [30] and sequential hypothesis testing LSH [11] use the hash values generated in the first stage of LSH, to efficiently verify if each two records in the same bucket of a hash table are indeed within the distance threshold or not. Paper [23] focuses on a specific locality-sensitive family of functions, the minwise hashing functions [9] for jaccard similarity: based on a theoretical framework, only a few bits are kept for each hash value, in order to reduce the space requirements and computational overhead. Paper [32] also focuses on a specific family of functions, 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 [21]: 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.

## 8. CONCLUSION

We proposed adaptive LSH, a novel approach for finding the records referring to the top-$k$ entities, in large datasets. 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 outcome is an accurately and drastically reduced, in terms of records, dataset: sophisticated, case-specific entity resolution algorithms can then be much more efficiently applied on the small dataset.

Our experiments involved different types of data: multi-field publication records, web articles, and images. We compared adaptive LSH to the widespread, for high dimensional data, LSH-blocking approach. The speedup ranges from 2x to 25x compared to the traditional LSH approach, while introducing only negligible errors due to the approach's probabilistic nature. Furthermore, we saw that even in datasets where the accuracy against the ground truth is not expected to be particularly high, we can increase the recall with a small penalty in performance. To verify that, in practice, adaptive LSH operates in a fundamentally different way than traditional LSH, we cherry-picked the best LSH variation, in each case. We found out that even compared to the hand-picked best LSH variation, adaptive LSH gives a substantial speedup, except for cases where a very large portion of the dataset refers to the top-1 entity and adaptive LSH is only slightly better than the best LSH variation.

## 9. 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. Hidden due to double-blind requirements.

[4] Top-k entity resolution with adaptive locality-sensitive hashing. *Technical report, http://bit.ly/2eOZOlS*.

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

[6] 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.

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

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

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros. Query adaptative locality sensitive hashing. In *International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, 2008.

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

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

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

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

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

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

[28] 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.

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

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

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

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

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

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

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

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

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

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

[39] 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 $h : R \rightarrow B$ 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, $h : R \rightarrow \{b_1, b_2\}$, 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, $h_1, h_2, \ldots h_w \in \mathcal{F}$, 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, $h_1, h_2, \ldots h_z \in \mathcal{F}$, 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 $h_1, h_2, \ldots h_w \in \mathcal{F}$, 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.

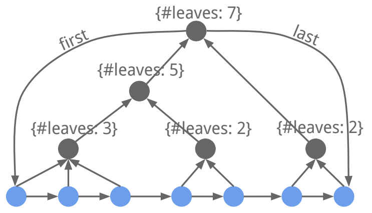


**Figure 18: Parent-pointer tree.**

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:

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

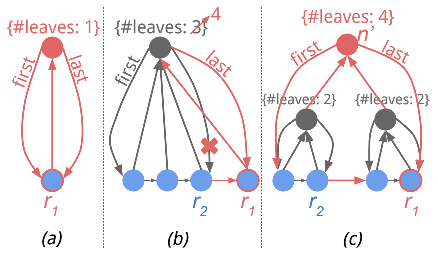


**Figure 19: Tree updates when adding a record $r_1$ to a hash table.**

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

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

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:

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

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

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

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:

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

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

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

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

## 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 2 *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:*

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

$$where \;\; 0 \leq d(f_a^{(i)}, f_b^{(i)}) \leq 1$$

*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,

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

An example where Theorem 2 applies is the family of minhash functions for the Jaccard distance. A more general version of Theorem 2 is stated in Theorem 3:

THEOREM 3 *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 2. □

## 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. OPTIMALITY ASSUMPTIONS

In this Appendix, we discuss when it could make sense for an algorithm *not* to follow the largest-first optimality assumptions, stated in Theorem 1, in Section 4.2.

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.