Pay-As-You-Go Entity Resolution

Steven Euijong Whang, David Marmaros, Hector Garcia-Molina

Computer Science Department, Stanford University
353 Serra Mall, Stanford, CA 94305, USA
{swhang, marmaros, hector}@cs.stanford.edu

Abstract. Entity resolution (ER) is the problem of identifying which records in a database refer to the same entity. In practice, many applications need to resolve large data sets efficiently, but do not require the ER result to be exact. For example, people data from the Web may simply be too large to completely resolve with a reasonable amount of work. As another example, real-time applications may not be able to tolerate any ER processing that takes longer than a certain amount of time. This paper investigates how we can maximize the progress of ER with a limited amount of work using "hints," which give information on records that are likely to refer to the same real-world entity. A hint can be represented in various formats (e.g., a grouping of records based on their likelihood of matching), and ER can use this information as a guideline for which records to compare first. We introduce a family of techniques for constructing hints efficiently and techniques for using the hints to maximize the number of matching records identified using a limited amount of work. Using real data sets, we illustrate the potential gains of our pay-as-you-go approach compared to running ER without using hints.

1 Introduction

Entity resolution [7, 28, 14] (also known as record linkage or deduplication) is the process of identifying records that represent the same real-world entity. For example, two companies that merge may want to combine their customer records. In such a case, the same customer may be represented by multiple records, so these matching records must be identified and combined (into what we will call a cluster).

An ER process is often extremely expensive due to very large data sets and compute-intensive record comparisons. For example, collecting people profiles on social websites can yield hundreds of millions of records that need to be resolved. Comparing each pair of records to estimate their "similarity" can be expensive as many of their fields may need to be compared and substantial application logic must be invoked.

At the same time, it may be very important to run ER within a limited amount of time. For example, anti-terrorism applications may require almost real-time analysis (where streaming data is processed in small batches using operations like ER) to capture a suspect who is on the brink of escaping. Although the analysis may not be as complete as when the full data is available, the fast processing can increase the chance of the suspect being captured. As another example, a people search engine may have very limited time for resolution, in order to provide up-to-date results on celebrities, politicians, and other high profile people in the news.

In this paper we explore a pay-as-you-go approach to entity resolution, where we obtain partial results "gradually" as we perform resolution, so we can at least get some results faster. As we will see, the partial results may not identify all the records that correspond to the same real-world entity. Our goal will be to obtain as much of the overall result as possible, as quickly as possible.

Figure 1 is a simple cartoon sketch to illustrate our approach. The horizontal axis is the amount of work performed, say the number of record pairs that are compared (using the expensive application logic). The vertical axis shows the "quality" of the result, say the number of pairs that have been found to match (i.e., to represent the same entity). The bottom curve in the figure (running mostly along the horizontal axis) illustrates the behavior of a typical non-incremental ER algorithm: it only yields its final answer after it has done all the work. If we do not have time to wait to the end, we get no results. The center solid line represents a typical incremental ER algorithm that reports results as it goes along. This algorithm is preferable when we do not have time for the full resolution.

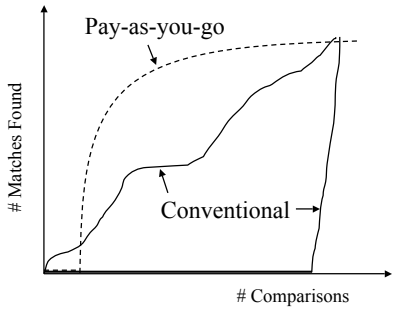


Fig. 1. Pay-as-you-go approach of ER

The dotted line in Figure 1 shows the type of algorithm we want to develop here: instead of comparing records in random order, it looks for matches in the "pairs that are most likely to match," hence it gets good quality results very fast. To identify the most profitable work to do early on, the algorithm performs some pre-analysis (the initial flat part of the curve). The pre-analysis yields what we call *hints* that are then used by the subsequent resolution phase to identify profitable work. If we have limited time, in our example say half of the time taken by the full resolution, our approach is able to give us a much better result than the traditional algorithms. Of course, in other cases our approach may be counterproductive (e.g., if the pre-analysis takes too long relative to the available time). Furthermore, not all ER approaches are amenable to the pay-as-you-go approach.

In this paper we address three important questions. First, how do we construct the hints? All schemes rely on an approximate and inexpensive way to compare records, e.g., two records are more likely to represent the same person if they have similar zip codes. However, there are several ways in which the hint can be encoded. For instance, a hint can be an ordered list of record pairs, sorted by likelihood of matching. A hint can also be an ordering of the records, that will lead to more profitable work in the resolution phase.

Second, how do we use the hints? The answer to this question depends on the ER strategy one is utilizing, and as stated earlier, some algorithms are not amenable to using hints. Since there are so many ER strategies available, clearly we cannot give a comprehensive answer to this second question, but we do illustrate the use of different types of hints in several representative instances.

Third, in what cases does pay-as-you-go pay off? Again, we cannot give a comprehensive answer but we do illustrate performance on several real scenarios and we identify the key factors that determine the desirability of pay-as-you-go.

It is important to note that our work is empirical by nature. Hints are heuristics. We will show they work well in representative cases, but they provide no formal guarantees. Also, our goal here is to provide a unifying framework for hints and to evaluate the potential gains. Certain types of hints have been used before (see Section 9), and we do not claim to cover all possible types of hints.

In summary, our contributions in this paper are as follows:

- We formalize pay-as-you-go ER where our goal is to improve the intermediate ER result. (Section 2) Our techniques build on top of blocking [20], which is a standard technique for scaling ER.
- We propose three types of hints:
  - *Sorted List of Record Pairs*: The most informative (but least compact) type of hint that contains a sorted list of record pairs (Section 3).
  - *Hierarchy of Partitions*: A moderately informative and compact type of hint that contains a series of partitions of records (Section 4).
  - *Sorted List of Records*: The most compact (but least informative) type of hint that contains a sorted list of records (Section 5).

For each hint type, we propose techniques for efficiently generating hints and investigate how ER algorithms can utilize hints to maximize the quality of ER while minimizing the number of record comparisons.
- We extend our approach to using multiple hints. (Section 6)
- We experimentally evaluate how applying hints can help ER do more work up front. (Section 8) We use actual comparison shopping data from Yahoo! Shopping and hotel information from Yahoo! Travel. Our results show scenarios where hints improve the ER processing to find the majority of matching records within a fraction of the total runtime.

2 Framework

In this section, we define our framework for pay-as-you-go ER. We first define a general model for entity resolution, and then we explain how pay-as-you-go fits in.

2.1 ER Model

An ER algorithm E takes as input a set of records R that describe real-world entities. The ER output is a partition of the input that groups together records that describe the same real-world entity. For example, the output $F = \{\{r_1, r_3\}, \{r_2\}, \{r_4, r_5, r_6\}\}$ indicates that records r_1 and r_3 represent one entity, r_2 by itself represents a different entity, and so on. Since sometimes we wish to run ER on the output of a previous resolution, we actually define the input as a partition. Initially, each record is in its own partition, e.g., $\{\{r_1\}$, $\{r_2\}$,$\{r_3\}$, $\{r_4\}$, $\{r_5\}$, $\{r_6\}\}$.

We denote the ER result of E on R at time t as $E(R)[t]$. In the above example, if E has grouped $\{r_1\}$ and $\{r_3\}$ after 5 seconds, then $E(R)[5] = \{\{r_1, r_3\}, \{r_2\}, \{r_4\}, \{r_5\}, \{r_6\}\}$. We denote the total runtime of $E(R)$ as $T(E, R)$. A quality metric M can be used to evaluate an ER result against the correct clustering of R. For example, suppose that M computes the fraction of clustered record pairs that are also clustered according to the correct ER answer. Then if $E(R) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and the correct clustering is $\{\{r_1, r_2\}, \{r_3\}$, $\{r_4\}\}$, $M(E(R)) = \frac{1}{3}$.

Most ER algorithms do their work by repeatedly comparing pairs of records to determine their semantic similarity or difference. Although ER algorithms use different strategies, the general principle is that if a pair of records appear "similar," then they are candidates for the same output partition. (We use the term *match* to refer to a pair that is similar enough to go in the same output partition. Details will vary by algorithm.) Since there are many potential records pairs to compare ($\frac{n\times(n-1)}{2}$ pairs for n records), most algorithms use some type of pruning strategy, where many pairs are ruled out based on a very coarse computation.

The most popular pruning strategy uses *blocking* or *indexing* [20, 12, 18, 8]. Input records are placed in blocks or canopies according to one or more of their fields, e.g., for product records, cameras are placed in one block, cell phones in another, and so on. LSH (locality sensitive hashing) [8] can also be used to place each record in one or more blocks. Then only pairs of records within the same block are compared. The number of record comparisons is substantially reduced, although of course matches may be missed. For instance, one store may call a camera-phone a cell phone while another may (mistakenly) call it a camera, so the two records from different stores will not be matched up even though they represent the same product.

Conceptually then we can think of blocking as defining a *set of candidate pairs* that will be carefully compared. The set may not be materialized, i.e., may only be implicitly defined. For instance, the placement of records in blocks defines the candidate set to be all pairs of records residing within a single block.

2.2 Pay-As-You-Go Model

With the pay-as-you-go model, we *conceptually* order the candidate pairs by the likelihood of a match. Then the ER algorithm performs its record comparisons considering first the more-likely-to-match pairs. The key of course is to determine the ordering of pairs very efficiently, even if the order is approximate.

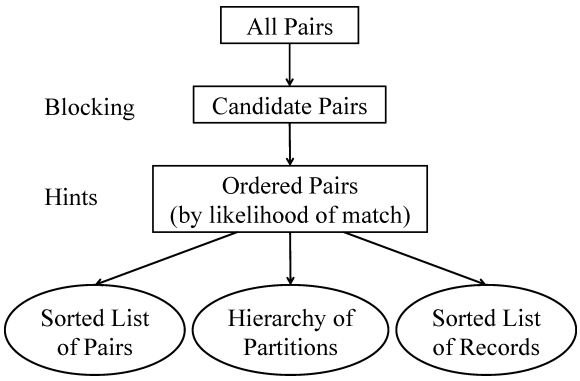


Fig. 2. Pay-As-You-Go ER Framework

To illustrate, say we have placed six records into two blocks: the first block contains records r_1, r_2, and r_3, while the second block contains r_4, r_5, and r_6. The implicit set of candidate pairs is $\{r_1 - r_2, r_1 - r_3, r_2 - r_3, r_4 - r_5 \ldots\}$. A traditional ER algorithm would then compare these pairs, probably by considering all pairs in the first block in some arbitrary order, and then the pairs in the second block. With pay-as-you-go, we instead first compare the most likely pair from either bucket, say $r_5 - r_6$. Then we compare the next most likely, say $r_2 - r_3$. However, if only one block at a time fits in memory, we may prefer to order each block independently. That is, we first compare the pairs in the first block by descending match likelihood, then we do the same for the second block. Either way, the goal is to discover matching pairs faster than by considering the candidate pairs in an arbitrary order. The ER algorithm can then incrementally construct an output partition that will more quickly approximate the final result. (As noted earlier, not all ER algorithms can be changed to compute the output incrementally and to consider candidate pairs by increasing match likelihood.)

More formally, we define a pay-as-you-go version of an ER algorithm as follows.

Definition 1. *Given a quality metric M, a pay-as-you-go algorithm E' of the ER algorithm E satisfies the following conditions.*

- Improved Early Quality*: For some given target time(s) $t_g < T(E, R)$, $M(E'(R)[t_g]) > M(E(R)[t_g])$. Target time t_g will typically be substantially smaller than $T(E, R)$ and represent the time at which early results are needed.*
- Same Eventual Quality*: $M(E'(R)[t]) = M(E(R)[t])$ for some time $t \geq T(E, R)$.*

The first condition captures our goal of producing higher-quality ER results upfront. The second condition guarantees that the pay-as-you-go algorithm will eventually produce an ER result that has the same quality as the ER result produced without hints. In comparison, blocking techniques may return an approximate ER result where the quality has decreased.

To efficiently generate candidate pairs in (approximate) order by match likelihood, we use an auxiliary data structure we call the *hints*. As illustrated in Figure 2, in this paper we discuss three types of hints. The most general form is a sorted list of record pairs, although as we will see, the list need not be fully materialized. A less general but more compact structure is a hierarchy where each level represents a partition of the records grouped by their likelihood of matching. A partition on a higher level is always coarser (see Definition 2) than a partition on a lower level of the hierarchy. The third structure is a sorted list of records (not pairs) where records that appear early in the list are more likely to match with each other than records far down the list.

Note that a hint is not an interchangeable "module" than can simply be plugged into any ER algorithm. Each hint is a tool that may or may not be applicable for a given ER algorithm. In the following three sections we describe each hint type in more detail, and show how it can be used by some ER algorithms. For

simplicity we will focus on processing a single block of records (although as noted earlier a single hint could span multiple blocks). In Section 6, we discuss how to use multiple hints for resolving records.

3 Sorted List of Record Pairs

In this section we explore a hint that consists of a list of record pairs, ranked by the likelihood that the pairs match. We assume that the ER algorithm uses either a distance or a match function. The distance function $d(r, s)$ quantifies the differences between records r and s: the smaller the distance the more likely it is that r and s represent the same real-world entity. A match function $m(r, s)$ evaluates to true if it is deemed that r and s represent the same real-world entity. Note that a match function may use a distance function. For instance, the match function may be of the form "if $d(r, s) < T$ and other conditions then true," where T is a threshold.

We also assume the existence of an estimator function $e(r, s)$ that is much less expensive to compute than both $m(r, s)$ and $d(r, s)$. The value of $e(r, s)$ approximates the value of $d(r, s)$, and if the ER algorithm uses a match function, then the smaller the value of $e(r, s)$, the more likely it is that $m(r, s)$ evaluates to true.

Conceptually, our hint will be the list of all record pairs, ordered by increasing e value. In practice, the list may not be explicitly and fully generated. For instance, the list may be truncated after a fixed number of pairs, or after the estimates reach a given threshold. As we will see, another alternative is to generate the pairs "on demand": the ER algorithm can request the next pair on the list, at which point that pair is computed. As a result, we can avoid an $O(N^2)$ complexity for generating the hint.

We now discuss how to generate the pair-list hint, and then how an ER algorithm can use such a list.

3.1 Generation

We first discuss how we can generate pair-list hints using cheaper estimation techniques. We then discuss a more general technique that does not require application estimates.

Using Application Estimates In some cases, it is possible to construct an application-specific estimate function that is cheap to compute. For example, if the distance function computes the geographic distance between people records, we may estimate the distance using zip codes: if two records have the same zip code, we say they are close, else we say they are far. If the distance function computes and combines the similarity between many of the record's attributes, the estimate can only consider the similarity of one or two attributes, perhaps the most significant.

To generate the hint, we can compute $e(r, s)$ for all record pairs, and insert each pair and its estimate into a heap data structure, with the pair with smallest estimate at the top. After we have inserted all pairs, if we want the full list we can remove all pairs by increasing estimate. However, if we only want the top estimates, we can remove entries until we reach a threshold distance, a limited number of pairs, or until the ER algorithm stops requesting pairs from the hint.

In other cases, the estimates map into distances along a single dimension, in which case the amount of data in the heap can be reduced substantially. For example, say $e(r, s)$ is the difference in the price attribute of records. (Say that records that are close in price are likely to match.) In such a case, we can sort the records by price. Then, for each record, we enter into the heap its closest neighbor on the price dimension (and the corresponding price difference). To get the smallest estimate pair, we retrieve from the heap the record r with the closest neighbor. We immediately look for r's next closest neighbor (by consulting the sorted list) and re-insert r into the heap with that new estimate. The space requirement in this case is proportional to $|R|$, the number of records. On the other hand, if we store all pairs of records in the heap, the space requirement is order of $O(|R|^2)$.

Application Estimate Not Available In some cases, there may be no known inexpensive application specific estimate function $e(r, s)$. In such scenarios, we can actually construct a "generic but rough" estimate based on sampling. This technique may not always give good results, but as we show in Section 8, it can yield surprisingly good estimates in some cases.

The basic idea is to use the expensive function d to compute the distances for a small subset of record pairs, and then use the computed distances to estimate the rest of the distances. We do not assume the records to be in any space (e.g., Euclidean), so d does not have to compute an absolute distance. The main advantage of this sampling technique is its generality where we can estimate distances by only using the given distance function. Suppose we have a sample S, which is a subset of the set of records R. We first measure the actual distances between all the records within S and between records in S and records in $R-S$. Assuming that the sample size $|S|$ is significantly smaller than the total number of records $|R|$, the number of real distances measured is much smaller than the total number of pairwise distances. For example, if $|R|$ = 1000 and $|S| = 10$, then the fraction of real distances we compute is $\frac{\binom{10}{2}+990\times10}{\binom{1000}{2}} = \frac{9945}{499500} \approx 2\%$.

Given a fraction of the real distances, we can estimate the other distances. One possible scheme captures the distance between two records r and s as the sum of squares of the difference of $d(r, t)$ and $d(t, s)$ for each $t \in S$. Formally, the estimate $e(r, s) = \Sigma_{t\in S}(d(r, t) - d(t, s))^2$. The intuition is that, if r and s are very close, then they will be almost the same distance from any sample point t. For example, if $d(r, t_1) =$ 8, $d(r, t_2) = 10$, $d(t_1, s) = 5$, and $d(t_2, s) = 4$, then $e(r, s) = (8 - 5)^2 + (10 - 4)^2 = 45$. While 45 is not a "real" distance, we only need to compare the *relative* sizes of estimates of different record pairs to construct hints. The estimated distances among records within S and between records in S and $R - S$ must also be computed the same way as above. Our techniques resemble triangulation techniques where a point is located by measuring angles to it from known reference points.

The sample set may affect the quality of estimation. In the worst case, the sample can be $|S|$ duplicate records, and all estimates turn out to be the same for any pair of records. Hence it is desirable for the sample records to be evenly dispersed within R as much as possible. In practice, selecting a small random subset of $|S|$ records works reasonably well (see Section 8.5).

3.2 Use

The details on how to use a pair-list hint depend on the actual ER algorithm used. However, there are two general principles that can be employed:

- If there is flexibility on the order in which functions $m(r, s)$ or $d(r, s)$ are called, evaluate these functions first on r, s pairs that are higher in the pair-list. This approach will hopefully let the algorithm identify matching pairs (or pairs that are clustered together) earlier than if pairs are evaluated in random order.
- Do not call the d or m functions on pairs of records that are low on the pair-list, assuming instead that the pair is "far" (pick some large distance as default) or does not match.

Note that in some cases the ER algorithm with hints will return the same final answer (call it F') as the unmodified algorithm (call it F), but matches or clusters will be found faster. In other cases, the ER algorithm will return an answer F' that is different from the unmodified answer F, but hopefully F' will have a high quality compared to F.

We now illustrate how the Sorted Neighbor algorithm [12] (called SN) can benefit from a pair-list hint. Say a block contains the records $R = \{r_1, r_2, r_3\}$. The SN algorithm first sorts the records in R using a certain key assuming that closer records in the sorted list are more likely to match. For example, suppose that we sort the records in R by their names (which are not visible in this example) in alphabetical order to obtain the list [r_3, r_2, r_1]. The SN algorithm then slides a window of size w on the sorted record list and compares all the pairs of clusters that are inside the same window at any point. If the window size is 2 in our example, then we compare r_3 with r_2 and then r_2 with r_1, but not r_3 with r_1 because they are never in the same window. We thus produce pairs of records that match with each other. We can repeat this process using different keys (e.g., we could also sort the person records by their address values). While collecting all the pairs of records that match, we can perform a transitive closure on all the matching pairs of records to

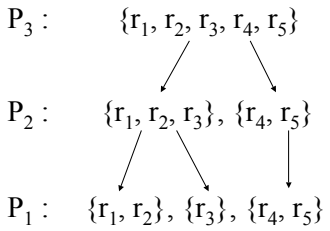


Fig. 3. A partition hierarchy hint for resolving R

produce a partition S of records. For example, if r_3 matches with r_2 and r_2 matches with r_1, then we merge r_1, r_2, r_3 together into the output $S = \{\{r_1, r_2, r_3\}\}$.

To use a pair list as a hint, we define the cheap distance function $e(r, s)$ to be the difference in rank between records according to the sorted list. That is, given two records r and s, $e(r, s) = |Rank(r) - Rank(s)|$ where $Rank(r)$ indicates the index of r in the sorted list of the records in R. Intuitively, the closer records are according to the sorted list, the more they are likely to match. In our example above, our sorted list is $[r_3, r_2, r_1]$, so $Rank(r_3) = 1$, $Rank(r_2) = 2$, and $Rank(r_1) = 3$. Hence, the distance between r_1 and r_2 is 1 while the distance between r_1 and r_3 is 2. The modified ER algorithm SN compares the records with the shortest estimated distances first, we are effectively comparing records within the smallest sliding window, and repeating the process of increasing the size of the window by 1 and comparing the records that are within the new sliding window, but have not been compared before. Notice that once the next shortest distance of records exceeds the window size w, we have done the exact same record comparisons as the SN algorithm. In addition, we can also stop comparing records in the middle of ER once we have exceeded the work limit W. For instance, if we set W to only allow one record comparison, then we only compare either $\langle r_3, r_2 \rangle$ or $\langle r_2, r_1 \rangle$ and terminate the ER algorithm.

4 Hierarchy of record partitions

In this section, we propose the partition hierarchy as a possible format for hints. A partition hierarchy gives information on likely matching records in the form of partitions with different levels of granularity where each partition represents a "possible world" of an ER result. The partition of the bottom-most level is the most fine-grained clustering of the input records. Higher partitions in the hierarchy are more coarse grained with larger clusters. That is, instead of storing arbitrary partitions, we require the partitions to have an order of granularity where coarser partitions are higher up in the hierarchy.

Definition 2. *A partition P is* coarser *than another partition P' (denoted as $P' \leq P$) when the following condition holds:*

- $\forall c' \in P', \exists c \in P$ *s.t.* $c' \subseteq c$

Figure 3 shows a hierarchy hint for the set of records $\{r_1, r_2, r_3, r_4, r_5\}$. Suppose that the most likely matching pairs of the records are $\langle r_1, r_2 \rangle$ and $\langle r_4, r_5 \rangle$. We can express this information as the bottom-level partition $\{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$ of the hierarchy. Among the clusters in the bottom level, suppose that $\{r_1, r_2\}$ is more likely to be the same entity as $\{r_3\}$ than $\{r_4, r_5\}$. The next level of the hint can then be a coarser partition of the bottom level partition where the clusters $\{r_1, r_2\}$ and $\{r_3\}$ from the bottom-level partition have merged.

We now formally define a partition hierarchy hint.

Definition 3. *A valid partition hierarchy hint H with L levels is a list of partitions $P_1, \ldots, P_L$ of R where $P_j \leq P_{j+1}$ for any $1 \leq j < L$.*

For example, Figure 3 is a valid partition hierarchy hint where $P_1 = \{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}$ (i.e., $P_1 \leq P_2$). However, if P_2 were $\{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4, r_5\}\}$, then H would not be valid because $P_1 \not\leq P_2$.

Within the hierarchy of a partition hierarchy hint, a cluster c in a higher level is connected to the clusters in the lower level that were combined to construct c. We call these clusters the children of c.

Definition 4. *The* children *of a cluster c (denoted as $c.ch$) in the ith level ($i > 1$) of a partition hierarchy hint H is the largest set of clusters S in the $(i-1)$st level of H such that $\forall c' \in S, c' \leq c$.*

For example, in Figure 3, the children of cluster $\{r_1, r_2, r_3\}$ in P_2 is the set $\{\{r_1, r_2\}, \{r_3\}\}$, and the children of cluster $\{r_4, r_5\}$ in P_2 is the set $\{\{r_4, r_5\}\}$.

A significant advantage of the partition hierarchy structure is that the storage space is linear in the number of records regardless of the height L. A compact way to store the information of a partition hierarchy is to keep track of the clusters splitting into their children in lower levels. For example, in Figure 3, there are two cluster splits: one that splits the cluster $\{r_1, r_2, r_3, r_4, r_5\}$ in P_3 into $\{r_1, r_2, r_3\}$ and $\{r_4, r_5\}$ and another that splits $\{r_1, r_2, r_3\}$ in P_2 into $\{r_1, r_2\}$ and $\{r_3\}$. Hence we only need to save the information of two cluster splits. Since a partition hierarchy can have at most $|R| - 1$ splits, the maximum space required to store the splits information is linear in the number of records.

4.1 Generation

We propose various methods for efficiently constructing a partition hierarchy. In the following section, we construct hints based on sorted records, which are application estimates. Next, we discuss how partition hierarchies can also be generated using hash functions (which are also application estimates) and inexpensive distance functions (which are not application estimates).

Using Sorted Records We explore how a partition hierarchy can be generated when the estimated distances between records can map into distances along a single dimension according to a certain attribute key.

Algorithm 1 shows how we can construct a partition hierarchy hint H using different thresholds $T_1, \ldots, T_L$ for partitioning records based on their key value distances. (The thresholds values are pre-specified based on the number of levels L in H.) For example, say we have a list of three records [*Bob*, *Bobby*, *Bobji*] (the records are represented and sorted by their names). Suppose that we set two thresholds $T_1 = 1$ and $T_2 = 2$, and use edit distance (i.e., the number of character inserts and deletes required to convert one string to another) for measuring the key distance between records. Algorithm 1 first reads *Bob* and adds it into a new cluster both for P_1 and P_2 (Step 9). Then we read *Bobby* and compare it with the previous record *Bob* (Step 6). The edit distance between *Bob* and *Bobby* is 2. Since this value is larger than T_1, we create a new cluster in P_1 and add *Bobby* (Step 9). Since the edit distance does not exceed T_2, we add *Bobby* into the first cluster in P_2 (Step 7). For the last record *Bobji*, the edit distance with the previous record *Bobby* is 4, which exceeds both thresholds. As a result, a new cluster with *Bobji* is created for both P_1 and P_2. The resulting hint thus contains two partitions: $P_1 = \{\{Bob\}, \{Bobby\}, \{Bobji\}\}$ and $P_2 = \{\{Bob, Bobby\}, \{Bobji\}\}$.

The following result shows the correctness of Algorithm 1. Proofs for this result and subsequent ones can be found in Appendix A.1.

Proposition 1. *Algorithm 1 returns a valid hint.*

Given that the input *Sorted* is already sorted, Algorithm 1 runs in $O(L \times |R|)$ time by iterating all records in *Sorted* and, for each record, iterating through all thresholds.

ALGORITHM 1: Generating a partition hierarchy hint from sorted records

1: **Input:** a list of sorted records $Sorted = [r_1, r_2, \ldots]$ and a list of thresholds $T = [T_1, \ldots, T_L]$
2: **Output:** a hint $H = \{P_1, \ldots, P_L\}$
3: Initialize partitions $P_1, \ldots, P_L$
4: **for** $r \in Sorted$ **do**
5: **for** $T_j \in T$ **do**
6: **if** $r.prev.exists() \wedge KeyDistance(r.key, r.prev.key) \leq T_j$ **then**
7: Add r into the newest cluster in P_j
8: **else**
9: Create new cluster in P_j containing r
10: **return** $\{P_1, \ldots, P_L\}$

Using Hash Functions We can also generate a partition hierarchy based on hash functions with different probabilities of collision. For example, minhash signatures [13] can be used to estimate set similarity. Or if an attribute of records contains categorical values, then each record can be hashed as its category. To generate L partitions for a hint, we can use a family of hash functions $H_1, \ldots, H_L$ where for any $1 \leq i < L$, H_i has a lower probability of collision than H_{i+1} and any collision that occurs in H_i also occurs in H_{i+1}. The algorithm for constructing the hint is similar to Algorithm 1, except that records are now assigned to clusters based on their hash values. For example, suppose that we have a set of three records {*Bobbie*, *Bobby*, *Bobji*}. Given H_1 that uses the first four characters of a name as a record's hash value while H_2 uses the first three characters, then $P_1 = \{\{Bobbie, Bobby\}, \{Bobji\}\}$ while $P_2 = \{\{Bobbie, Bobby, Bobji\}\}$. The complexity of the algorithm is $O(L \times |R|)$ because for each level, we iterate all the records and assign each record to its bucket in each level.

Using Distance Estimation Functions We can also use an inexpensive distance estimator function $e(r, s)$ to generate a partition hierarchy. The $e(r, s)$ function can be application specific or generated by a sampling technique (see Section 3.1).

Algorithm 2 shows how we can construct a partition hierarchy hint given the distance estimates. For each level L_j in H, we can use a union-find algorithm [23] to generate a transitive closure of records that have estimated distances less than a given threshold T_j. For example, suppose we have three records r_1, r_2, r_3 whose estimated distances are set as $e(r_1, r_2) = 1$, $e(r_1, r_3) = 2$, $e(r_2, r_3) = 3$. Also, suppose that we set $T_1 = 1$ and $T_2 = 2$. Algorithm 2 first initializes all partitions $P_1, \ldots, P_L$ into empty sets (Step 3). For the first pair $\langle r_1, r_2 \rangle$, we compare its estimated distance 1 with $T_1 = 1$ (Step 6). Since r_1 and r_2 are close enough, we connect r_1 and r_2 in P_1 (Step 7). Next, we compare the estimated distance 1 with $T_2 = 2$. Again, r_1 and r_2 are connected in P_2. We then read the next pair of records $\langle r_1, r_3 \rangle$. Since the estimated distance is 2, r_1 and r_3 are connected in P_2, but not in P_1. For the last pair $\langle r_2, r_3 \rangle$, the estimated distance 3 exceeds both thresholds. As a result, the resulting hint contains two partitions: $P_1 = \{\{r_1, r_2\}, \{r_3\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}\}$. In Step 7, one can use a more sophisticated clustering algorithm (instead of a transitive closure) provided that the clustering results $\{P_1, \ldots, P_L\}$ satisfy Definition 3.

Notice that when estimating the distances between records, we do not have to actually store the estimates for each pair of records, which would require a space quadratic in the number of records. Instead, we can construct the partition hierarchy *while* generating the estimates. Hence, the space complexity of construction based on sampling is $O(L \times |R|)$. The time complexity for constructing the partition hierarchy is $O(|R|^2 + L \times C(|R|))$ where $|R|^2$ is needed for the sampling and $C(|R|)$ is the complexity of the clustering algorithm used to generate the partitions of R in the hierarchy.

4.2 Use

Given a partition hierarchy, the next question is how an ER algorithm can actually exploit this information to maximize the ER quality with a limited amount of work. We assume the ER algorithm is given based

ALGORITHM 2: Generating a partition hierarchy hint from pairwise distance estimates

1: **Input:** a list of pairs with their estimated distances $Pairs = [\langle r_1, s_1\rangle, \langle r_2, s_2\rangle, \ldots]$, and a list of thresholds $T = [T_1, \ldots, T_L]$
2: **Output:** a hint $H = \{P_1, \ldots, P_L\}$
3: Initialize partitions $P_1, \ldots, P_L$
4: **for** $\langle r, s\rangle \in Pairs$ **do**
5: **for** $T_j \in T$ **do**
6: **if** $e(r, s) \leq T_j$ **then**
7: TransitiveClosure($P_j, \langle r, s\rangle$)
8: **return** $\{P_1, \ldots, P_L\}$

on what works best for the application or what developers have experience with. In general, there are two principles that can be employed to use a partition hierarchy:

- If there is flexibility on the order of which records are resolved, compare the records that are in the same cluster in the bottom-most level of the hierarchy hint.
- If there is more time, start comparing records in the same cluster in higher levels of the hierarchy hint.

Algorithm 3 shows how a partition hierarchy hint can be used by an ER algorithm. Given a set of records R, an ER algorithm E, a partition hierarchy hint H, and a work limit W, we intuitively resolve the records in the bottom-level clusters first and progressively resolve more records in higher-level clusters in the hierarchy until there are no more records to resolve or the amount of work done exceeds W (e.g., the number of record comparisons should not exceed 1 million).

We illustrate Algorithm 3 using the Single-link Hierarchical Clustering algorithm [9, 17] (which we call HC_S). The HC_S algorithm merges the closest pair of clusters (i.e., the two clusters that have the smallest distance) into a single cluster until the smallest distance among all pairs of clusters exceeds a certain threshold T. The distance between two records is measured using a commutative distance function D that returns a non-negative distance between two records. When measuring the distance between two clusters, the algorithm takes the smallest possible distance between records within the two clusters. Now suppose we have $R = \{r_1, r_2, r_3\}$ (which can also be viewed as a list of three singleton clusters) where the pairwise distances are $D(r_1, r_2) = 2$, $D(r_2, r_3) = 4$, and $D(r_1, r_3) = 5$ with a given threshold $T = 2$. The HC_S algorithm first merges r_1 and r_2, which are the closest records and have a distance smaller or equal to T, into $\{r_1, r_2\}$. The cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the minimum of $D(r_1, r_3)$ and $D(r_2, r_3)$, which is 4. Since the distance exceeds T, $\{r_1, r_2\}$ and $\{r_3\}$ do not merge, and the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$.

We can use Algorithm 3 to run the HC_S algorithm with a hint that is a partition hierarchy. Continuing our example above where $R = \{r_1, r_2, r_3\}$, suppose that we are given the hint $P_1 = \{\{r_1, r_2\}, \{r_3\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}\}$. Also say that W is set to three record comparisons. According to Algorithm 3, we first resolve the clusters in P_1 of the hint. Thus we compare r_1 with r_2 by invoking $Resolve(E, \{r_1, r_2\}, h)$ in Step 8. Since r_1 and r_2 match, F becomes $\{\{r_1, r_2\}, \{r_3\}\}$. We also store the ER results of $\{r_1, r_2\}$ and $\{r_3\}$ in h. Next, we start resolving records in the cluster $\{r_1, r_2, r_3\}$ in P_2. When resolving $\{r_1, r_2, r_3\}$, we first subtract from F the clusters that are subsets of $\{r_1, r_2, r_3\}$, leaving us with $F = \{\}$ (Step 7). We now run $Resolve(E, \{r_1, r_2, r_3\}, h)$ in Step 8. Again, only r_1 and r_2 match and we union F with $\{\{r_1, r_2\}, \{r_3\}\}$ (Step 9). Assuming *Resolve* used at least two more record comparisons to resolve $\{r_1, r_2, r_3\}$, the total work is larger or equal to the work limit W, and we return the ER result $F = \{\{r_1, r_2\}, \{r_3\}\}$ (Step 12), which is the correct answer. Notice that, if W was set to 1 instead of 3, the same ER result $F = \{\{r_1, r_2\}, \{r_3\}\}$ would have been returned using only one record comparison.

Proposition 2. *Given a valid ER algorithm E, Algorithm 3 returns a correct ER result when $P_L = \{R\}$ and W is unlimited.*

The complexity of Algorithm 3 is at least the complexity of the ER algorithm E because we can always use a hierarchy with one level having $\{R\}$ as its partition. The actual efficiency of the algorithm largely

ALGORITHM 3: Using a partition hierarchy in ER

```
1: Input: a set of records R, an ER algorithm E, a hint H = {P_1, ..., P_L}, and a work limit W
2: Output: an intermediate ER result F of E(R)
3: F ← ∅, h ← ∅
4: for i = 1 ... L do
5:    for c ∈ P_i do
6:       for child ∈ c.ch do
7:          F ← F − {clus | clus ∈ F ∧ clus ⊆ child}
8:       h(c) ← Resolve(E, c, h)
9:       F ← F ∪ h(c)
10:      if total work ≥ W then
11:         return F
12: return F
```

depends on the implementation of $Resolve(E, c, h)$ in Step 8. In the worst case, E can simply ignore the information of resolved records h and run $E(c)$ from scratch. However, an ER algorithm can exploit the information in h to produce $Resolve(E, c, h)$ more efficiently. For example, if $c = \{r_1, r_2, r_3\}$ and we know by h that r_1 and r_2 are the same entity. Then the ER algorithm can avoid a redundant record comparison between r_1 and r_2.

4.3 General Incremental Property

We explore an interesting property of an ER algorithm (called "general incremental") that, when satisfied, can enable efficient computation given information of previously resolved records. That is, given an input set of records R and ER results of previously resolved records, we would like $E(R)$ to run faster than resolving R from scratch. An ER algorithm is incremental [14] if it can resolve one record at a time. We use a more generalized version of the incremental property [26] for our ER model where subsets of R can be resolved in any order.

In order to precisely define the general incremental property, we need to formalize the ER definition in Section 2.1 further. First, we assume that an ER algorithm receives a partition of R (called R_p) and returns a new partition of R. This view does not change our original ER model (where ER partitions a set of records) because a set of records $R = \{r_1, \ldots, r_n\}$ can also be viewed as a set of singleton clusters $R_p = \{\{r_1\}, \ldots, \{r_n\}\}$. We denote all the possible partitions that can be produced by the ER algorithm E as $\bar{E}(R_p)$, which is a set of partitions of R. That is, we assume that ER is non-deterministic in a sense that different partitions of R may be produced depending on the order of records processed or by some random factor (e.g., the ER algorithm could be a random algorithm). Hence, $E(R_p)$ is always one of the partitions in $\bar{E}(R_p)$. For example, given $R_p = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, $\bar{E}(R_p)$ could be $\{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$ while $E(R_p) = \{\{r_1, r_2\}, \{r_3\}\}$.

Definition 5. *An ER algorithm is* generally incremental *[26] if for any four partitions P_1, P_2, F_1, and F_2 such that*

- *$P_1 \subseteq P_2$ and*
- *$F_1 \in \bar{E}(P_1)$ and*
- *$F_2 \in \bar{E}(F_1 \cup (P_2 - P_1))$*

then $F_2 \in \bar{E}(P_2)$.

For example, suppose we have $P_1 = \{\{r_1\}, \{r_2\}\}$, $P_2 = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and $F_1 = \{\{r_1, r_2\}\}$. That is, we have already resolved P_1 into the result F_1. We can then add to F_1 the remaining cluster $\{r_3\}$, and resolve all the clusters together (i.e., we run $E(\{\{r_1, r_2\}, \{r_3\}\})$). The result is as if we had resolved everything from scratch (i.e., from P_2). Presumably, the former way (incremental) will be more efficient than the latter by exploiting the information on records have already been resolved.

ALGORITHM 4: Efficient *Resolve* function using information on previously resolved records

1: **Input:** an ER algorithm E, a set c of records to resolve and a hash table h containing ER results of sets of records
2: **Output:** $F \in \bar{E}(\{\{r\}|r \in c\})$
3: $R_p \leftarrow \emptyset$
4: **for** $s \in c.ch$ **do**
5: $\quad R_p \leftarrow R_p \cup h(s)$
6: **if** $R_p = \emptyset$ **then**
7: $\quad R_p \leftarrow \{\{r\}|r \in c\}$
8: **return** $E(R_p)$

Proposition 3. *Suppose we have a partition $S = \{s_1, \ldots, s_n\}$ of R_p. That is, $\bigcup s_i = R_p$ and for any $1 \leq i, j \leq n$ where $i \neq j$, $s_i \cap s_j = \emptyset$. Then given a general incremental ER algorithm E, $F = E(\bigcup_{i=1\ldots n} E(s_i)) \in \bar{E}(R_p)$.*

We now propose Algorithm 4 that runs ER on previously resolved records and can be used as the *Resolve* function in Algorithm 3. For example, suppose that $c = \{r_1, r_2, r_3, r_4, r_5\}$ and c's children $c.ch = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}$. Also say that $h(\{r_1, r_2, r_3\}) = \{\{r_1, r_3\}, \{r_2\}\}$, and $h(\{r_4, r_5\}) = \{\{r_4, r_5\}\}$. We thus construct R_p as $\{\{r_1, r_3\}, \{r_2\}, \{r_4, r_5\}\}$ in Steps 4–5. Alternatively, if h did not contain any ER result, then at Step 7 R_p would have been set to the singleton partition of c, i.e., $\{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_5\}\}$. The algorithm then returns $E(R_p)$. In the former case where h does contain ER results of previously resolved records, Algorithm 4 is presumably faster than simply running ER from the singleton partition of c by avoiding redundant record comparisons.

The following result shows the correctness of Algorithm 4.

Proposition 4. *If E is general incremental (satisfying Definition 5), Algorithm 4 correctly returns an ER result $F \in \bar{E}(\{\{r\}|r \in c\})$.*

We now show that the HC_S algorithm is general incremental and can thus be used in Algorithm 4.

Proposition 5. *The HC_S algorithm is general incremental.*

5 Ordered List of Records

We now propose an ordered list of records as a format for hints. In comparison to a partition hierarchy, a list of records tries to maximize the number of matching records identified when the list is resolved sequentially. Two significant advantages are that the ER algorithm itself does not have to change in order to exploit the information in a record list and that there is no required storage space for the hint. On the downside, finding the right ordering of records in order to guide the ER algorithm to find matching records as much as possible is a non-trivial task where the best solution depends on the ER algorithm itself. We propose general techniques for constructing record lists either from a partition hierarchy or from sampling. We then discuss how a record list can be used by ER algorithms.

5.1 Generation

We propose methods for efficiently constructing a list of records. The following section uses a partition hierarchy for generation. We also discuss how record lists can be generated using distance estimation functions.

Using Partition Hierarchies We propose a technique for generating record lists based on a partition hierarchy. Assuming that an ER algorithm resolves records in the input list from left to right, a desirable feature of a record list is to order the records such that the ER algorithm can minimize the number of fully identified entities at any point of time. A fully identified entity is one where the ER algorithm has found all the matching records for that entity. For example, given a record list $[r_1, r_2, r_3]$ where r_1 refers to the same entity as r_2, an ER algorithm fully identifies the entity for $\{r_1, r_2\}$ after resolving the first two records and fully identifies the entity for $\{r_3\}$ after resolving the last record. Another input list could be $[r_3, r_1, r_2]$ where one entity (i.e., $\{r_3\}$) is already identified after resolving the first record in the list. The first list is better as a record list in a sense that the only record match between r_1 and r_2 was found early on. The second list is worse because $\{r_3\}$ was fully identified early on, and the comparison between r_1 and r_3 was unnecessary and could have been done after matching r_1 and r_2. That is, if we are only able to do one record comparison, then we will find the correct answer when using the record list $[r_1, r_2, r_3]$ and not when using the list $[r_3, r_1, r_2]$.

In general, we want to minimize the entities that are fully identified because they generate unnecessary comparisons with newer records resolved. We will later capture this idea by minimizing the expected number of fully-identified entities when the record list is resolved sequentially from left to right. While we can use other orderings for generating a record list hint, our generation focuses on ER algorithms that follow the guideline in Section 5.2 where records in the front of the list are compared first.

Given a partition hierarchy H with L levels, we assume each of the partitions $P_1, \ldots, P_L$ are equally likely to be the ER answer. That is, each partition has the same chance of being the correct ER result of R and is thus a possible world of the records resolved. Suppose that we resolve a subset S of R. For each partition P_j, we estimate the number of clusters that are fully identified by resolving S as $nEntities_j(S) = \Sigma_{c \in P_j} \frac{|c \cap S|}{|c|}$. Since each partition is equally likely to be the answer, we define the overall estimate of the number of entities fully identified by resolving S as $\Sigma_{j=1\ldots L}(\frac{1}{L} \times nEntities_j(S))$.

For example, suppose that the partition hierarchy H_1 has 3 levels where $P_1 = \{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$, $P_2 = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}$, and $P_3 = \{\{r_1, r_2, r_3, r_4, r_5\}\}$. Each partition is equally likely to be the ER answer. Suppose that we resolve the set of records $S = \{r_1, r_2, r_4, r_5\}$, which is a subset of R. Then according to our definition of $nEntities$, the estimated number of entities identified in P_1 is $\frac{2}{2} + \frac{2}{2} = 2$ because all records in $\{r_1, r_2\}$ and $\{r_4, r_5\}$ have been resolved. For P_2, the estimation is $\frac{2}{3} + \frac{2}{2} = \frac{5}{3}$ because 2 out of 3 records in $\{r_1, r_2, r_3\}$ and all records in $\{r_4, r_5\}$ have been resolved. For P_3, the estimation is $\frac{4}{5}$. Our overall estimate for the actual number of entities identified $nEntities(S)$ is thus $\frac{1}{3} \times (2 + \frac{5}{3} + \frac{4}{5}) = \frac{67}{45}$, i.e., about 1.5 entities identified.

One could extend our model by allowing each partition to have its own probability of being the ER answer. That is, for each possible world P_i, we add a probability w_i indicating the confidence we have on that possible world. Given that the sum of the weights is 1, the estimated number of fully identified entities for the set S resolved would be $\Sigma_{j=1\ldots L}(w_j \times nEntities_j(S))$. While we have considered the extension, we have chosen the current simple scheme for two reasons. First, setting the probabilities for each partition is difficult in practice. Second, the simple scheme usually performs as well as any other scheme using different probabilities (see Section 8.6).

We now define an optimal record list. Intuitively, we would like to minimize the number of entities fully identified at any point in time given that the ER algorithm resolves the records in the input list from left to right. We define a prefix set of a list to be the set of records from the beginning of the list. For example, the prefix sets of the list $[r_1, r_2]$ are $\{\}$, $\{r_1\}$, and $\{r_1, r_2\}$.

Definition 6. *A record list H of R is* optimal *if any prefix set P of H has a minimum value of $\Sigma_{j=1\ldots L}(\frac{1}{L} \times nEntities_j(P))$ among all subsets of R with size $|P|$.*

Interestingly, we can always derive a record list that is optimal according to Definition 6. A key observation is that $nEntities\ (S) = \Sigma_{r \in S}\ nEntities(\{r\})$, which says that the expected number of entities identified by a set S is the sum of the expected numbers of entities identified by the records in S. For example, according to H_1 (defined above), $nEntities(\{r_1, r_3\}) = \frac{1}{L} \times (nEntities_1(\{r_1, r_3\}) + nEntities_2(\{r_1, r_3\}) + nEntities_3(\{r_1, r_3\})) = \frac{1}{L} \times ((\frac{1}{2} + \frac{1}{1}) + \frac{2}{3} + \frac{2}{5}) = \frac{1}{3} \times (\frac{1}{2} + \frac{1}{3} + \frac{1}{5}) + \frac{1}{3} \times (\frac{1}{1} + \frac{1}{3} + \frac{1}{5}) = nEntities(\{r_1\}) + nEntities(\{r_3\})$. Hence, by simply sorting the records in R by their $nEntities$ values in increasing order

ALGORITHM 5: Generating an optimal record list from a partition hierarchy

1: **Input:** the set of records R, a hint $H = \{P_1, \ldots, P_L\}$
2: **Output:** an optimal record list H'
3: **for** $r \in R$ **do**
4: $\quad nEntities(\{r\}) = 0$
5: **for** $i = 1 \ldots L$ **do**
6: $\quad$ **for** $c \in P_i$ **do**
7: $\quad\quad$ **for** $r \in c$ **do**
8: $\quad\quad\quad nEntities(\{r\}) \leftarrow nEntities(\{r\}) + \frac{1}{L} \times \frac{1}{|c|}$
9: $H' \leftarrow$ Sorted records in R by their $nEntities$ values in increasing order
10: **return** H'

(remember, we want to *minimize* the number of fully-resolved records), we can derive a record list where any prefix set has an optimal $nEntities$ value.

Algorithm 5 derives an optimal record list according to Definition 6. Using Steps 5–8 we compute the estimated number of entities for each record. According to H_1 above, record r_3 has a $nEntities$ value of $\frac{1}{3} \times (\frac{1}{1} + \frac{1}{3} + \frac{1}{5}) = \frac{46}{90}$. Similarly, r_1 and r_2 each have a value of $\frac{1}{3} \times (\frac{1}{2} + \frac{1}{3} + \frac{1}{5}) = \frac{31}{90}$. Finally, records r_4 and r_5 each have a value of $\frac{1}{3} \times (\frac{1}{2} + \frac{1}{2} + \frac{1}{5}) = \frac{36}{90}$ (the fractions were not reduced for easy comparison). We now sort the records by their $nEntities$ values (Step 9) in increasing order. In our example, we can produce the record list $H' = [r_1, r_2, r_4, r_5, r_3]$. (Records with the same $nEntities$ value can swap positions within the list.) As a simple verification, no prefix set of size 2 has a $nEntities$ value smaller than that of $\{r_1, r_2\}$, which is $\frac{31}{90} + \frac{31}{90} = \frac{62}{90}$.

We now show that Algorithm 5 returns an optimal record list. The proof is given in Appendix A.2.

Proposition 6. *Algorithm 5 returns an optimal record list.*

The complexity of Algorithm 5 is $O(|R| \times (L + log(|R|)))$ because of the loop in Steps 5–8 and the sorting of records in Step 9. In special cases, however, the sorting can be done in linear time. For example, if there is only one level P in the hierarchy, then for a record r, $nEntities(\{r\}) = \Sigma_{c \in P} \frac{|\{r\} \cap c|}{|c|} = \frac{1}{|c_r|}$ where c_r is the cluster in P containing r. Hence, we can sort the records in R by the sizes of clusters that contain them in decreasing order. If we are given a maximum cluster size constant $MaxSize$ of P, we can create $MaxSize$ buckets and assign each record r to the bucket with the index $|c_r|$. We can then generate an optimal record list by iterating through all the buckets. The sorting can thus be done in $O(|R|)$ time. Using our example above, suppose that $P = \{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$. Then one optimal record list is $[r_1, r_2, r_4, r_5, r_3]$ because the cluster in P containing r_3 has a size of 1 while the cluster sizes for the other four records are all 2. Here, we can sort the five records using two buckets (i.e., $MaxSize = 2$).

Obviously, there are other ways to generate record lists using a partition hierarchy. For example, one could simply return the records in one of the partitions of the partition hierarchy. Our approach is a general way to produce record lists and has theoretical guarantees of minimizing the expected number of entities identified at any point in time.

Using Distance Estimation Functions We can also use an inexpensive distance estimator function $e(r, s)$ (application specific or sampling based) to generate a record list. Algorithm 6 shows how we can generate a list that resembles the given pair list. For example, given the pair list $[\langle r_1, r_2\rangle, \langle r_1, r_3\rangle, \langle r_2, r_3\rangle]$, we first read the pair $\langle r_1, r_2\rangle$ and append the records r_1 and r_2 to H. For the next pair $\langle r_1, r_3\rangle$, we only need to append r_3 to H because r_1 already exists in H. Hence, we generate the record list $H = [r_1, r_2, r_3]$. Of course, the record list H does not necessarily preserve all the information in the pair list. While some information cannot avoid being lost, we make the best effort to place the most likely matching records up front in the record list.

Unlike Algorithm 2 where a partition hierarchy hint can be constructed without sorting the list of pairs, a record list requires the list of pairs to be sorted by the estimated distances of the pairs. Hence, the complexity of Algorithm 2 is $O(|R|^2 \times log(|R|))$ where R is the set of input records. Moreover, a space complexity of

ALGORITHM 6: Generating a record list based on a list of pairs

```
1: Input: a list of pairs with their estimated distances L = [⟨r1, s1⟩, ⟨r2, s2⟩, ...]
2: Output: a record list H
3: H ← []
4: Sort L by estimated distances in increasing order
5: for ⟨r, s⟩ ∈ L do
6:    if r ∉ H then
7:       H ← H + r
8:    if s ∉ H then
9:       H ← H + s
10: return H
```

$O(|R|^2)$ is required to store the estimated distances of all record pairs. In the case where there is limited time or space, we can approximate the result of Algorithm 2 by only retaining the top-k closest pairs where k is a parameter reflecting the limited time or space. A heap structure can be used to store the top-k pairs while estimating the pairwise distances in the sampling scheme. We then consider all the other pairs to be infinitely distanced. The time complexity of Algorithm 2 is then $O(k \times log(k))$ while the space complexity $O(k)$. In Section 8.3, we experimentally show that limiting k can significantly improve the time and space requirements with almost no decrease in quality.

5.2 Use

A record list can be applied to any ER algorithm that accepts as input a record list. A key advantage of using record lists is that the ER algorithm itself does not have to change. The following principle can be employed to benefit from a record-list hint:

– If there is flexibility in the order of which records are resolved, resolve the records in the front of the list first.

Again, our goal is to help the ER algorithm with hints to efficiently return an answer F' that has high precision and recall relative to the unmodified answer F.

As an example, we consider hierarchical clustering based on a Boolean comparison rule [3] (called HC_B), which can benefit from record lists. The HC_B algorithm combines matching pairs of clusters in any order until no clusters match with each other. The comparison of two clusters can be done using an arbitrary function that receives two clusters and returns true or false, using the Boolean comparison function B to compare pairs of records. For example, suppose we have $R = \{r_1, r_2, r_3\}$ (which can also be viewed as a list of three singleton clusters) and the comparison function B where $B(r_1, r_2)$ = true, $B(r_2, r_3)$ = true, but $B(r_1, r_3)$ = false. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs (e.g., a record r_2 has an ID of 2) from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Depending on the order of clusters compared, the HC_B algorithm can merge $\{r_1\}$ and $\{r_2\}$ first, or $\{r_2\}$ and $\{r_3\}$ first. In the first case, the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$ (because the clusters $\{r_1\}$ and $\{r_2\}$ match, but $\{r_1, r_2\}$ and $\{r_3\}$ do not match) while in the second case, the ER result is $\{\{r_1, r_2, r_3\}\}$ (the clusters $\{r_2\}$ and $\{r_3\}$ match, and then $\{r_1\}$ and $\{r_2, r_3\}$ match). Now given a record list $[r_1, r_2, r_3]$ (the ordering is arbitrary and is set to illustrate the behavior of HC_B), the HC_B algorithm first compares r_1 and r_2. If we set the work limit W to one record comparison, then HC_B will terminate returning $\{\{r_1, r_2\}, \{r_3\}\}$.

6 Using Multiple Hints

Until now, we have assumed that one hint is generated per block. However, depending on the type of hint and the number of attributes used to generate the hint, we may have to generate multiple hints in order to

accurately capture the order information of record pairs. For example, suppose that we are resolving people records and there are two ways to order the pairs of records: by their name or address similarities. If we are generating a list of pairs hint, then we could choose one of two attributes – name or address – and use it to sort the pairs into one hint. Another option is to combine the name and address similarity into one similarity (e.g., by taking a weighted sum of the values) and generate one hint. Finally, we can generate two separate hints for the two attributes. In this section, we assume that multiple hints of the same type are generated (corresponding to the last case in the above example) and discuss three options on how to exploit them while resolving records.

The first straightforward option is to repeatedly resolve the block of records for each hint and combine the results. For example, we could resolve the people with the closest names first and then resolve those with the closest addresses first and union the matching records. While this method is easy to apply to any type of hint, the overall runtime of resolving records may slow down due to redundant record comparisons for different hints.

Another option is to merge the multiple orderings into one ordering and then resolve the records once using this new combined hint. For example, given two sorted lists of records, we can merge the lists by sorting the records according to their sum of ranks in the two sorted lists. As another example, if we are combining two partition hierarchies, we could combine each level by performing a meet operation on the corresponding partitions. While combining hints may result in a loss of information of the ordering of pairs, the main advantage is that there is only one hint to use when resolving records and thus no redundant record comparisons.

The final option is to exploit the multiple hints simultaneously without combining them into one hint. For example, if there are two sorted lists of record pairs, then we can start reading the first pairs of records from both hints. If any record pair from one hint has already been read from the other hint, then we can read the next pair of records. While this option has the potential to fully exploit the ordering information of all the hints, deciding how exactly we can exploit the multiple hints is not obvious.

7 Determining which Hint to Use

As mentioned in Section 2.2, an ER algorithm may only be compatible with some types of hints (or with none at all), depending on the data structures and processing used. In this section we provide some hint selection guidelines and then illustrate how the guidelines apply to the ER algorithms we have already introduced.

If the ER algorithm compares pairs of records, and there is an estimator function e that is cheaper than the distance function d, a pair-list hint may be useful. If there is no estimator function e, then sampling techniques can be used to estimate the other distances. Next, if the ER algorithm clusters records based on their relative distances, then a hierarchy hint could be useful for focusing on the relatively closer records first. Finally, if the ER algorithm performs a sequential scan of records when resolving them, a record list hint may help compare the records that are more likely to match first.

Figure 4 summarizes our three hint types and the techniques used to generated them (see Section 8.1 for details). The figure also shows the ER algorithms we used in Sections 3 through 5 to illustrate each hint type. Although we could use a hierarchy hint or a record-list hint for the SN algorithm, the pair-list hint can be used most naturally because SN basically compares pairs of records that are likely to match in a given order. We use a partition hierarchy hint for the HC_S algorithm because HC_S can naturally resolve subsets of R with the guidance of the partition hint. While HC_S can also use a record list as a hint, the record list is designed to work better for ER algorithms that resolve records sequentially. For the HC_B algorithm we use a record lists hint because HC_B sequentially resolves its records. The HC_B algorithm could also use a partition hierarchy as its hint. However, we would have to modify HC_B and thus change its efficient algorithm for comparing records.

Hint	Generated from	ER algorithm
Pair list (PL)	Cheap dist. functions Sampling	SN
Hierarchy (H)	Sorted records Hash functions Sampling	HC_S
Record list (RL)	Partition hierarchy Complete sampling Partial sampling	HC_B

Fig. 4. Hints to generate and ER algorithms to run

8 Experimental Results

In this section, we evaluate pay-as-you-go ER on real data sets and show how creating and using hints can improve the ER quality given a limit on how much work can be done. We assume that blocking [20] is used (see Section 2), as it is in most ER applications with massive data. With blocking, the input records are divided into separate blocks using one or more key fields. For instance, if we are resolving products, we can partition them by category (books, movies, electronics, etc). Then the records within one block are resolved independently from the other blocks. This approach lowers accuracy because records in separate blocks are not compared, but makes resolution feasible. (See [18, 27] for more sophisticated approaches.) From our point of view, the use of blocking means that we can read a full block (which can still span many disk blocks) into memory, perform resolution using hints, and then move on to the next block. In our experiments we thus evaluate the cost of resolving a single block, except for Section 8.8 where we perform scalability experiments by resolving multiple blocks. Keep in mind that these costs should be multiplied by the number of blocks. Finally in our experiments, we generate *one hint* for each block. Our approach can easily be extended to multiple hints using the techniques described in Section 6.

We start by describing our experimental setting in Section 8.1. In Section 8.2, we show how using hints can improve the ER quality with limited amounts of work. In Section 8.3, we investigate the CPU time and space overhead for creating hints and discuss the tradeoffs between the overhead and benefit of using hints. In Section 8.4 we investigate the right number of levels in a partition hierarchy hint. In Section 8.5, we explore the impact of the sample size on the accuracy of hints using sampling techniques. In Section 8.6, we experiment on record lists generated from partition hierarchies using an extended model where partitions in the hierarchy now have different confidence values. In Section 8.7, we discuss how partition hierarchy hints can still enhance ER algorithms that are not incremental. In Section 8.8, we show how hints can be used to enhance ER in practical scenarios where the datasets can be very large. Finally, in Section 8.9, we show how our hint generation algorithms scale.

8.1 Experimental setting

In this section, we describe the settings used for our experiments. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.

Quality Metric Since ER results may now be incomplete, it is important to measure the quality of an intermediate ER result. We compare an intermediate result with a "Gold Standard," which is the result of running ER on the same dataset to the end. Notice that we are *not* measuring the correctness of the ER algorithm itself, but instead determining how "close" the intermediate results are to the exhaustive result. Since ER results are partitions of the input set of records, we consider all the input records in the same output cluster to be identical. For instance, if records r and s are clustered into $\{r, s\}$ and then clustered with t, all three records r, s, t are considered to be the same (i.e., to match).

Suppose that the Gold Standard G contains the record pairs that match for the exhaustive solution while set S contains the matching pairs for the intermediate result. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall

Re is $\frac{|G \cap S|}{|G|}$. If the precision Pr is always 1 (i.e., the incremental algorithm always reports true matches), we use Re, the fraction of matching pairs found, as our quality metric. Otherwise, we can use the F_1 metric, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, as the quality metric (for more general ER metrics, see [19]).

We will use recall as our metric for all the experiments sections (except for Section 8.7) because the precision is always 1 for any intermediate ER result.

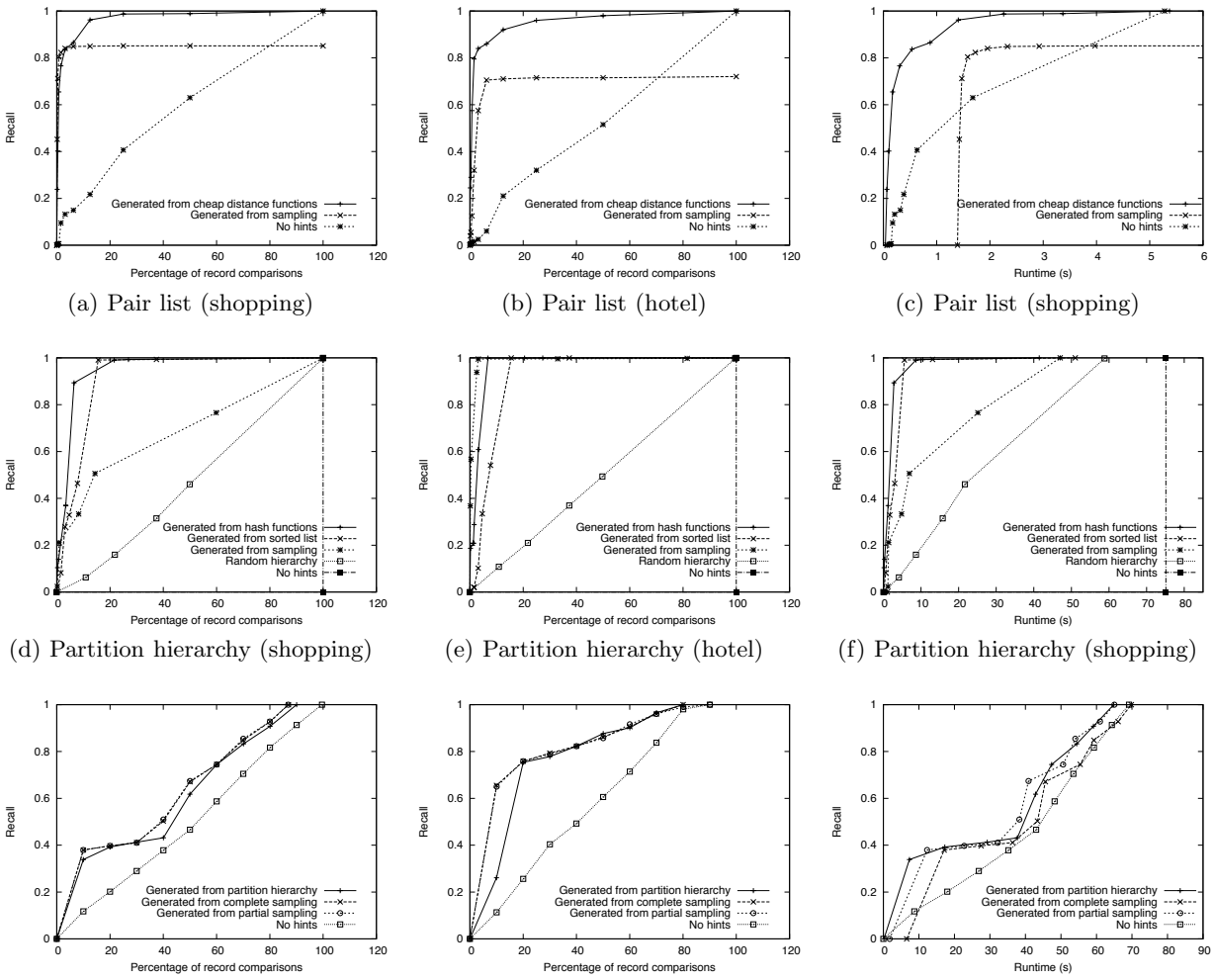
Real Data The comparison shopping dataset we use was provided by Yahoo! Shopping and contains millions of records that arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains attributes including the title, price, and category of an item. We experimented on a random subset of 3,000 shopping records that had the string "iPod" in their titles and 2 million shopping records. When scaling ER on 2 million shopping records (see Section 8.8), the average block size was 124 records while the maximum block size was 6,082 records. Hence, the random subset of 3,000 shopping records can be considered as one (relatively large) block. We also experimented on a hotel dataset provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. We experimented on a random subset of 3,000 hotel records located in the United States. Each hotel record contains attributes including the name, address, city, state, zip code, latitude, longitude, and phone number of a hotel. Again, the 3,000 hotel records can be considered as one block. While the 3K shopping and hotel datasets fit in memory, the 2 million shopping dataset did not fit in memory and had to be stored on disk.

Hints and ER Algorithms For our experiments we use the three ER algorithms used to illustrate our hints (and summarized earlier in Figure 4). In this sub-section we provide some implementation details for the ER algorithms used.

The SN algorithm uses a Boolean match function for comparing two records. When comparing shopping records, the Boolean match function B compares the titles, prices, and categories. When comparing hotel records, B compares the states, cities, zip codes, and the names of the two hotels. We generate a pair list using cheap distance functions or from sampling. When generating record lists using cheap distance functions, we used the estimate function $e(r, s) = |Rank(r) - Rank(s)|$ using the title (name) attributes of shopping (hotel) records as the sort key. When generating pair lists using sampling, we only computed and stored the top-$((w-1) \times |R| - \frac{w \times (w-1)}{2})$ closest pairs (i.e., the number of record pairs that would be compared by SN given the window size w) to limit the time and space overhead. We also used a random sample of 10 records.

The HC_S algorithm uses a distance function for comparing two records. When the comparing shopping records, the distance function D measures the Jaro distance [28] between the titles of two records. For the hotel records, D measures the Jaro distance of the names of two records. We generate partition hierarchies in three ways: using sorted records, hash functions, and sampling. By default, we set the number of levels of a partition hierarchy to 5. While increasing the number of levels helps us find more matching records early on, the benefits diminish from a certain point (see Section 8.4). The partition hierarchies based on sorted lists were balanced binary trees with the highest level containing a single cluster with all input records. The partition hierarchies based on hash functions used the prefixes of titles (names) as the hash values of shopping (hotel) records. When generating partition hierarchies using sampling, we clustered records with similar titles (for the shopping dataset) or names (for the hotel dataset) using several string comparison thresholds. We randomly selected 10 records for our samples. (In Section 8.5, we show that small sample sizes are sufficient for reasonable results.) A partition hierarchy is suitable for the HC_S algorithm because the hint suggests sets of records to resolve first, and the HC_S algorithm can easily resolve subsets of records at a time.

The HC_B algorithm uses a Boolean match function for comparing two records. When comparing shopping records, the Boolean match function B compares the titles, prices, and categories. When comparing hotel records, B compares the states, cities, zip codes, and the names of the two hotels. We generate a record list from a partition hierarchy (generated with hash functions) and from sampling. When generating a partition hierarchy used for constructing a record list, we used minhash signatures [13] generated from titles (names)



(a) Pair list (shopping) (b) Pair list (hotel) (c) Pair list (shopping)

(d) Partition hierarchy (shopping) (e) Partition hierarchy (hotel) (f) Partition hierarchy (shopping)

(g) Record list (shopping) (h) Record list (hotel) (i) Record list (shopping)

Fig. 5. Recall of ER algorithms using hints against work or runtime, 3K shopping/hotel records

as the hash values of shopping (hotel) records. When generating record lists using sampling, we tested two schemes. For the complete sampling scheme, we computed and stored all the estimate distances of pairs (i.e., $\frac{|R|\times(|R|-1)}{2}$ pairs) and generated a record list. For the partial sampling scheme, we only computed and stored the top-$(5 \times|R|)$ closest pairs to limit the time and space overhead. In both schemes, we used a random sample of 10 records.

In all our algorithms, we avoid expensive comparisons when possible by comparing in phases. For example, when comparing two shopping records, we compare the category, price, and title attributes, in that order. If the categories do not match, we avoid comparing the prices and titles. If the categories match, but not the prices, we avoid comparing the titles. This way, we can avoid many expensive title string comparisons. When experimenting on large datasets, we use various blocking techniques (see Section 8.8) to further scale ER. While more optimizations can be used on the base ER algorithms, our focus is to show the *relative* benefits of using hints compared to when they are not used.

8.2 Hint Benefit

In this section, we explore the benefits of using hints by measuring the recall values for various ER algorithms using different hints. Figure 5(a) shows how a pair list can help the SN algorithm compare the most likely matching record pairs for 3,000 shopping records. We experimented on the SN algorithm using two types of hints. Recall that the SN algorithm first sorts the records by a certain key. In our implementation, we sorted the records by their titles and then slid a window of size 100, comparing only the record pairs within the same window. The first hint we used was to order the pairs of records according to their difference in rank according to the sorted list. That is, the difference in rank was considered the distance between two records. The second hint we used estimated the pairwise distance between the records using the sampling technique (see Section 3.1) and compared the records with the closest estimated distance first. In our experiments, we set the sample size to 10 records. (In Section 8.5, we show that even a sample this small produces reasonable results.) Notice that when using the sampling technique, the SN algorithm does not use a sliding window on a sorted list of the records, but simply compares the pairs of records as dictated by the pair list.

As more records are compared using the match function B, the quality of SN using hints rapidly increases. For example, the quality of SN using a pair list generated from cheap distance functions achieves 0.96 recall with only 12.5% of the record comparisons required when running SN without hints. The quality of SN using the sampling technique achieves 0.8 recall with 0.78% of the entire work. While the sampling techniques gives a high recall early on, it does not give 1.0 recall even after performing as many comparisons as the SN algorithm without hints. The reason is that there are still matching record pairs that would have been found by SN without hints, but are further down the pair list and will eventually be compared if more pairs are compared (recall that the SN algorithm only compares a small fraction of the total record pairs using a sliding window). In Section 8.5, however, we show that the sampling technique is actually very good at finding all matching pairs that are not necessarily within the same window. Finally, the recall of SN without hints increases linearly with more record comparisons.

Figure 5(d) shows how a partition hierarchy can help the HC_S algorithm to quickly identify matching records for 3,000 shopping records. The bottom-right plot (in Figure 5(d)) shows the progress of the original HC_S algorithm where records are clustered only after all pairs of base records are compared. Notice that the clustering of records does not involve record comparisons, which is why the original HC_S algorithm has a jump in recall from 0 to 1 when 100% of the record comparisons are done. The actual runtime for the second clustering step is very small (0.004s). The random hierarchy plot shows how a randomized partition hierarchy helps the ER quality. Here, the records are clustered in a random fashion without any similarity comparisons. As a result, the plot shows a linear increase of recall as the number of record comparisons increases. The other three plots use partition hierarchies generated from a sorted list, hash functions, and sampling. Among them, a partition hierarchy based on sampling gives the slowest increase in recall where we get 0.51 recall with 14% of the comparisons HC_S uses without hints. The main reason for the relatively low recall is that the partitions in the hierarchy were highly skewed where some clusters in a partition were very large. As a result, the partitions in the hierarchy were not "pinpointing" the likely matching records. Moreover, setting the thresholds for creating the partitions was not a trivial task, making this approach relatively difficult to use. When using a partition hierarchy hint generated from a sorted list, we achieve 0.99 recall with 16% of the total comparisons of HC_S without hints. Finally, when using a hint generated using hash functions, we achieve a similar result of 0.89 recall using 6.5% of the total comparisons.

Figure 5(g) shows how record lists can help the HC_B algorithm to identify matching records early without modifying the ER algorithm itself. Again, we experimented on 3,000 shopping records. When using a record list generated from a partition hierarchy, we obtain 0.61 recall with 50% of the comparisons used by HC_B without hints. Record lists generated from complete or partial sampling give similar results where we obtain 0.67 recall with 50% of the total comparisons. In contrast, the HC_B algorithm without hints obtains 0.47 recall for 50% of its comparisons. While the complete and partial sampling schemes produce near-identical recall results against the number of record comparisons done, we will see in Section 8.3 that the partial sampling scheme outperforms the complete sampling scheme in recall against the actual ER runtime. Although the record list does not generally improve HC_B as much as partition hierarchies improve HC_S, the main advantage is that all these benefits were achieved without modifying the HC_B algorithm itself.

Hint	Generation	Time Overhead		Space Overhead (Const/Use)	
		Sho3K	**Ho3K**	**Sho3K**	**Ho3K**
Pairs List	Cheap dist. functions	0.005	0.19	22 / 22	7.8 / 7.8
	Sampling	0.16	3.56	22 / 22	7.8 / 7.8
Hierarchy	Sorted records	4E-4	2E-4	0.07 / 0.07	0.02 / 0.02
	Hash functions	1E-4	1E-4	0.08 / 0.08	0.03 / 0.03
	Sampling	0.02	0.01	0.08 / 0.08	0.03 / 0.03
Record List	Partition hierarchy	7E-4	0.01	0.08 / 0	0.03 / 0
	Complete sampling	0.09	1.07	349 / 0	119 / 0
	Partial sampling	0.02	0.31	1.15 / 0	0.4 / 0

Fig. 6. Time and space hint construction overhead depending on the type of hint, 3K shopping/hotel records

Figures 5(b), 5(e), and 5(h) show the hint results when resolving 3,000 hotel records. Unlike the shopping dataset where multiple records can match, the records in the hotel datasets mostly come from two data sources that do not have duplicates within themselves, so relatively few clusters have a size larger than 2. The hotel results show that a partition hierarchy based on sampling or any record list performs better on hotel data than when they are used on shopping data. Figures 5(c), 5(f), and 5(i) show the recall values of ER algorithms against runtime and will be explained in Section 8.3.

8.3 Hint Overhead

In this section we explore the CPU and memory space overhead of using hints. We first explore the time and space overhead of constructing and using hints. We then show the tradeoffs between the overhead and benefit of using hints from various perspectives.

Time and Space Overhead The time overhead of a hint consists of the time to construct the hint and the time to use the hint. While we will measure the construction time for hints, the time overhead of using the hints themselves is not significant. The usage time overhead for accessing a pair list is a simple iteration of the pairs in the list. The usage time overhead for accessing a partition hierarchy is an iteration of the clusters from the bottom partition to top. There is no time overhead for using a record list because we simply reorder the input list of records.

The "Time Overhead" column in Figure 6 shows the construction time overhead for each type of hint in Figure 4 (we explain the space overhead later). The sub-column head Sho3K means 3,000 shopping records while the sub-column head Ho3K means 3,000 hotel records. Each construction time overhead was produced by dividing the construction time of a hint by the CPU time for running the ER algorithm without using any hints. For example, the construction time for a partition hierarchy based on hash functions using 3,000 shopping records is 0.0001x the time for running the HC_S algorithm without hints.

The overhead for constructing pair lists based on cheap functions depends on the number of pairs compared (which depends on the window size w). The larger the window size, the larger the construction time overhead. The overhead for constructing pair lists based on sampling is more expensive because all record pairs are compared before taking the top matching pairs. The time overhead for resolving 3,000 hotel records is 3.56x, which means that the time to construct the hint takes longer than running the ER algorithm itself. In this case, it is better to simply run the ER algorithm. The overhead for constructing partition hierarchies based on sorting or hashing is very small compared to running the HC_S algorithm because the record comparisons in HC_S are relatively expensive. Even if sampling is used (which requires a runtime quadratic in the number of input records), the construction time overhead is 0.02x for shopping records because the cost for estimating distances is much cheaper than computing the real distances. The overhead for constructing a record list from a partition hierarchy is relatively small compared to running the HC_B algorithm because, again, the record comparisons in HC_B are relatively expensive. However, when constructing a record list with complete sampling, the time overhead for HC_S resolving 3,000 hotel records is 1.07x. The partial sampling

scheme significantly improves the complete sampling scheme where the time overhead for the same hint and data is 0.31x. Note that this improvement comes with almost no penalty in recall (see Figure 5(h)).

The "Space Overhead (Const/Use)" column in Figure 6 shows the space overhead for each type of hint. The space overhead of a hint consists of the memory space needed for constructing the hint and the memory space needed to use the hint while running ER. Both of these costs can be significant and will be explored. The words "Const" and "Use" indicate the construction space overhead and usage-space overhead, respectively. The construction space overhead of a hint was computed by dividing the memory space needed for creating the hint by the memory space needed to store the input record list. The usage-space overhead of a hint was computed by dividing the memory space needed for storing the constructed hint by the memory space of the input record list. For example, the construction space overhead of a record list based on a partition hierarchy is 0.08x the space needed to store 3,000 shopping records while the space needed to store and use that hint (i.e., the usage overhead) is 0. Note that the space overhead is dependent on the size of the input records (i.e., if the records are larger, then the space overhead will decrease).

The space overhead for pair lists is proportional to the number of record pairs stored (which depends on the window size w). While the current space overhead for shopping records is 22x, one could reduce the window size to reduce the overhead if necessary. (Of course, reducing the number of pairs stored comes at a price of reducing the recall of SN.) The space overhead is same regardless of the how the list was made because the sampling technique store exactly the same number of record pairs as when using cheap functions. The space overhead for partition hierarchies based on sorted records and hash functions is reasonably small (0.07–0.08x for shopping records) because the hierarchy size is linear in the number of records. A partition hierarchy based on sampling has a reasonable construction space overhead (0.08x for shopping records) because we do not actually store the pairwise distance estimates computed by the sampling technique. The record-list hint based on a hierarchy hint has a construction space overhead of 0.08x because the partition hierarchy hint was based on hash functions. The record-list hint based on complete sampling has a large construction space overhead (349x for shopping records) because of the quadratic space required. This result is the largest space overhead a sampling scheme can have where all distance estimates between records are sorted and stored. The partial sampling scheme, however, shows a much lower and reasonable space overhead (1.15x for shopping records). We achieve this significant improvement with near-identical recall results (see Figures 5(g) and 5(h)). Finally, both record-list hints do not have usage-space overhead.

Tradeoff between Time Overhead and Benefit We now observe how the construction time overhead of a hint actually affects the overall runtime of ER. We experiment on 3,000 shopping records. Figures 5(c), 5(f), and 5(i) show the recall values of ER results as a function of the ER runtime. The plots do not differ significantly from Figures 5(a), 5(d), and 5(g), respectively. While the construction time overhead are reflected in the plots, only the plots for using pair lists based on sampling and record lists based on complete and partial sampling show visible construction time overhead. When using pair lists based on sampling, it takes 1.45 seconds for SN to perform better than SN without hints. We also observe that the runtime needed to cluster records by HC_S after the pairwise distances is negligible (0.004s) compared to the total ER runtime. The results show that hints can benefit ER in runtime even with the construction time overhead.

We demonstrate how hints are helpful in finding "most" of the matching record pairs efficiently. Figure 7 shows how efficient hints are when obtaining 0.8 recall using 3,000 shopping records. For each hint type, we measure its construction time overhead (x-value). We then measure the time for the ER algorithm using the hint to achieve 0.8 recall divided by the ER runtime without hints (y-value). For example, a partition hierarchy generated from sampling for SN takes 0.01x the time to run HC_S without hints (x-value). Also, the time for HC_S with this hint to get 0.8 recall takes 0.37x the time to run HC_S without hints (y-value). Hence, the total runtime of SN using the hierarchy is 0.38x the runtime for SN without hints. Notice that in the case where the sum of the x and y values of a point is 1 (i.e., if the point is on the diagonal line $X + Y = 1$), then running ER with hints to obtain 0.8 recall takes the same time as running ER fully without hints. Hence, a hint is useful when its point is below the diagonal line. Our results show that all hints have points below the diagonal line, which means that our hints can efficiently identify 80% of the matching record pairs.

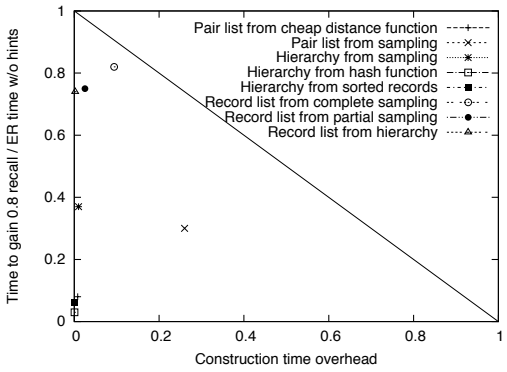


Fig. 7. Construction time versus time to obtain 0.8 recall, 3K shopping records

Figure 8 shows how the construction time of a hint can affect the point when using a hint starts to help. At one extreme, if there is no construction time, then hints can improve ER progress within a short time. On the other hand, if the construction time is very large, it may take many record comparisons until the overhead starts to pay off. For each hint type, we vary the construction time and convert it into number of record comparisons performed. For example, suppose HC_S does Z record comparisons without using hints. Then we can set the construction time of a partition hierarchy based on sampling to be equivalent to, say, 35% of Z. For each construction time, we also derive the number of record comparisons when ER using hints starts to achieve higher recall than ER without hints. When using a partition hierarchy based on sampling it takes about 46% of Z comparisons for the overhead of constructing the hint to pay off (see the right-most black-circle in Figure 8). The "ideal" plot would be exactly the $Y = X$ plot where no matter the overhead of constructing the hint, we immediately start benefitting by using the hints. For each hint, there is a point where the hint can no longer benefit ER with larger construction times. For example, if the construction overhead for a partition hierarchy based on sampling exceeds 35% of Z (i.e., if we go beyond the right-most black-circle), then the HC_B algorithm using the hint can never perform better than HC_B without hints.

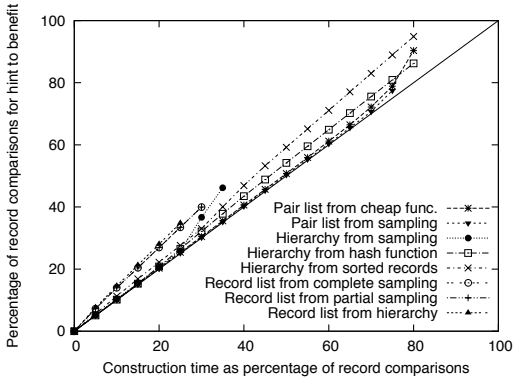


Fig. 8. Construction time impact on hint payoff point, 3K shopping records

8.4 Choosing the Number of Levels

Figure 9 shows the impact of the number of levels in the recall achieved by a given number of record comparisons. We resolved 3,000 shopping records using the HC_S algorithm using a hierarchy hint generated

from the records sorted by their titles. Each hierarchy has $\{R\}$ as its highest-level partition and was a binary tree where each cluster $\{r_1, \ldots, r_n\}$ had exactly two children $\{r_1, \ldots, r_{\frac{n}{2}}\}$ and $\{r_{\frac{n}{2}+1}, \ldots, r_n\}$. As a result, the more levels there are in the hint (increasing from 3 to 9), the steeper the recall curve becomes. For example, while using a hint with 3 levels gives a 65% recall for 25% of the total comparisons done by HC_S without hints, using a hint with 7 levels gives a 98% recall with only 15% of the total comparisons. Starting from 7 levels, however, the recall improvement becomes negligible. When the number of levels increase from 3 to 9, the hint construction time overhead ranges from 1.4E-4x to 5.3E-4x and the space overhead 0.07x to 0.08x. Hence, the time and construction space overhead do not significantly change with varying numbers of levels.

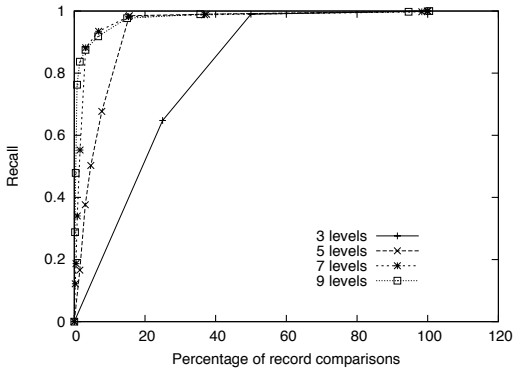


Fig. 9. Number of levels impact on recall, 3K shopping records

8.5 Sampling Performance

Figure 10 shows how the sample size affects the sampling scheme. We resolved 3,000 shopping records using a pair list as a hint where we simply compared pairs of records using a Boolean match function B following the order in the hint and performed a transitive closure at the end. The sample sizes ranged from 2 to 1000 and were chosen randomly from the input set of records. We also ran the naïve method where records were compared in a random order. The ER result was compared with the entire result of comparing all pairs of records using B and performing a transitive closure at the end. As a result, even a sample size of 2 produced a result significantly better than the random comparisons result and close to the result using a sample size of 1,000. Notice that the sample size = 2 plot sometimes performs better than the sample size = 10 plot. This result implies that simply having a larger sample size does not guarantee strictly better ER results. In summary, our sampling results show that small sample sizes suffice for near-optimal results.

8.6 Using Weights on Partition Hierarchy Levels

We consider the scenario where the HC_B algorithm uses a record list hint generated from a partition hierarchy. In Section 5.1, we discussed an extension of the partition hierarchy model where each partition can have a weight (or confidence value) associated with it. In this section, we vary the weights when generating record lists and see the impact the weights have on the quality of HC_B. We can use Algorithm 5 to generate an optimal record list by replacing Step 8 with the line "$nEntities(\{r\}) \leftarrow nEntities(\{r\}) + w_i \times \frac{1}{|c|}$."

Figure 11 shows for each combination of weights, the final recall of running HC_B on 3,000 shopping records where the work limit is set as half the number of comparisons HC_B would have performed without hints. For example, if $w_1 = 1$ and all the other weights are 0, we only use the bottom-most partition of the hierarchy for generating the record list. In Figure 11, the recall values range from 0.431 to 0.634. Not

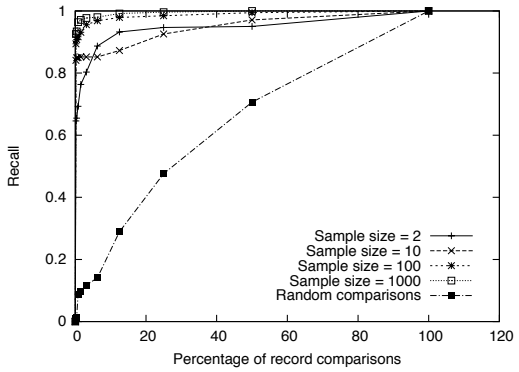


Fig. 10. List of pairs using sampling, 3K shopping records

surprisingly, the minimum recall occurs when $w_5 = 1$, which means that only the highest-level partition was used to generate the record list. Any other weight assignment gives better results than setting $w_5 = 1$. The recall when all weights have equal values (i.e., each weight is $\frac{1}{5}$) is 0.618, which is not significantly lower than the highest recall possible. We conclude that using equal weights is a reasonable strategy with the benefit that one does not have to fine-tune the weights of the hierarchy.

Weights					**Recall**	**Weights**					**Recall**
w_1	w_2	w_3	w_4	w_5		w_1	w_2	w_3	w_4	w_5	
1	0	0	0	0	0.623	0	0	0	0	1	0.431
0	1	0	0	0	0.598	1/2	0	0	0	1/2	0.623
1/2	1/2	0	0	0	0.620	0	1/2	0	0	1/2	0.598
0	0	1	0	0	0.535	1/3	1/3	0	0	1/3	0.620
1/2	0	1/2	0	0	0.634	0	0	1/2	0	1/2	0.535
0	1/2	1/2	0	0	0.591	1/3	0	1/3	0	1/3	0.634
1/3	1/3	1/3	0	0	0.621	0	1/3	1/3	0	1/3	0.591
0	0	0	1	0	0.491	1/4	1/4	1/4	0	1/4	0.621
1/2	0	0	1/2	0	0.620	0	0	0	1/2	1/2	0.491
0	1/2	0	1/2	0	0.591	1/3	0	0	1/3	1/3	0.620
1/3	1/3	0	1/3	0	0.620	0	1/3	0	1/3	1/3	0.591
0	0	1/2	1/2	0	0.529	1/4	1/4	0	1/4	1/4	0.620
1/3	0	1/3	1/3	0	0.620	0	0	1/3	1/3	1/3	0.529
0	1/3	1/3	1/3	0	0.585	1/4	0	1/4	1/4	1/4	0.620
1/4	1/4	1/4	1/4	0	0.618	1/5	1/5	1/5	1/5	1/5	0.618

Fig. 11. Weights impact on accuracy, 3K shopping records

8.7 Non-incremental ER algorithms

We now experiment with ER algorithms that do not satisfy the general incremental property (see Definition 5). A non-incremental ER algorithm is not guaranteed to return a correct ER result when using Algorithm 4 for resolving clusters (see below). In this section, we experiment on the complete-link hierarchical clustering algorithm (called HC_C), which is identical to the HC_S algorithm (see Section 4.2) except that the distance between two clusters is defined by the maximum pairwise distance between their records. To see how using a partition hint can incorrectly alter the ER result of HC_C, suppose that we have $R = \{r_1, r_2, r_3\}$ where the pairwise distances are $D(r_1, r_2) = 2$, $D(r_2, r_3) = 2$, and $D(r_1, r_3) = 5$ with a given threshold T

= 2. The ER result would be $\{\{r_1, r_2\}, \{r_3\}\}$ because $\{r_1\}$ and $\{r_2\}$ match while $\{r_1, r_2\}$ and $\{r_3\}$ do not (having a distance of 5). However, if the partition hierarchy hint contains one partition $\{\{r_1\}, \{r_2, r_3\}\}$, then HC_C will resolve $\{r_2, r_3\}$ first and will merge r_2 and r_3. Since $\{r_1\}$ and $\{r_2, r_3\}$ do not match, the ER result is $\{\{r_1\}, \{r_2, r_3\}\}$. However, this result can never occur when running $E(R)$ without hints. We measure the accuracy of an intermediate ER result using the F_1 measure defined in Section 2.1. Our experiments were done using 3,000 shopping records.

Figure 12 shows the accuracy results of running HC_C against the number of record comparisons performed. Among all schemes, using a partition hierarchy hint generated from a hash function produces an ER result with 0.98 accuracy using only 6% of the record comparisons required for a naïve approach without hints. The experiments show that partition hierarchy hints can produce highly-accurate ER results with few record comparisons even if the ER algorithms are not general incremental.

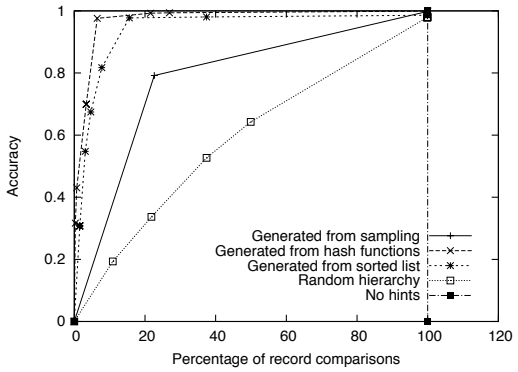


Fig. 12. Non-incremental algorithm accuracy, 3K shopping records

8.8 Early Termination on Large Datasets

We now scale our techniques on 0.5–2 million shopping records. Since the records do not fit in memory, we used blocking techniques as described in the beginning of Section 8. We used minhash signatures [13] for distributing the records into blocks. For the shopping dataset, we extracted 3-grams from the titles of records. We then generated a minhash signature for each records, which is an array of integers where each integer is generated by applying a random hash function to the 3-gram set of the record.

While hints can help maximize the ER quality, it is not obvious exactly when to stop ER without knowledge on how many more matching records need to be identified. We compare three possible schemes on when to terminate ER:

- *No Limit*: We simply run ER without hints to the end.
- *Popcorn Scheme (Limit Rate)*: We stop when the rate of newly found matching pairs drops below a threshold. The analogy is making popcorn where we stop cooking when the frequency of pops drops below a certain level.
- *TV Dinner Scheme (Limit Computation)*: We limit the number of record comparisons based on the number of records to be resolved. The analogy is heating a TV Dinner in a microwave oven for a fixed amount of time as specified by the cooking instructions.

We used the HC_S algorithm and partition hierarchy hints generated from sorted lists. The first Popcorn scheme is useful when we want to maximize recall and yet minimize the runtime as much as possible. In our implementation, we terminate ER when the rate of finding new matching pairs among all record pairs compared drops below 1%. For example, for the next 200 record pairs compared, if fewer than 2 pairs

Scheme	Runtime(hrs)			Recall		
	0.5M	1M	2M	0.5M	1M	2M
No Limit	0.4	1.8	15	1.0	1.0	1.0
Popcorn	0.12	0.37	2.5	0.98	0.98	0.99
TV Dinner	0.08	0.26	1.3	0.6	0.6	0.6

Fig. 13. Runtime and recall for different schemes, 2M shopping records

matched, then we terminated HC_S. The rate was checked after each level iteration in the hierarchy. The second TV Dinner scheme is useful when there is only a given amount of time for the application to run. In our experiments, we set the computation limit to be 10% of the total number of record pairs in the current set of records to be resolved. For example, when resolving a cluster of size 20, we ran at most about $\frac{1}{10} \times \frac{20 \times 19}{2}$ = 19 record comparisons.

Figure 13 shows how the two schemes perform compared to when ER runs without hints. We measured the entire ER runtimes including the IO costs for reading and writing blocks on disk. However, the bottleneck for the entire ER process was the CPU time to resolve the blocks in memory. While the Popcorn scheme tends to give better recall, it does not guarantee termination within a given amount of time. On the other hand, while the TV Dinner scheme has the advantage of having a predictable runtime, it may not always give the best recall results. The runtime improvements (at most 11.5x) are not as high as what we observed in the 3,000 shopping dataset results. (According to Figure 5(f), we can obtain 0.99 recall about 18x faster than running ER without hints using partition hierarchies generated from sorted lists on 3,000 shopping records.) The reason is that in our scenario many blocks were not large enough for hints to help as much (i.e., the overhead of constructing hints did not pay off as much), so the average benefit of using hints was relatively low. Nevertheless, using hints can still significantly improve the runtime of ER on large datasets (by 3.3–11.5x) while still obtaining high recall.

8.9 Scalability of Generating Hints

Table 14 shows the scalability results for generating hints. The construction times for hints scale well with the exception of generating a record list using complete sampling (for 2M records, the memory overflowed). However, by using partial sampling instead, we can obtain scalability with minimal loss in quality (see Section 8.2).

Hint	Generation	0.5M	1M	2M
PL	Cheap dist. fns	4	9	20
	Sampling	47	120	309
H	Sorted records	3	6	12
	Hash functions	8	11	16
	Sampling	40	130	379
RL	Par. hierarchy	11	19	31
	Com. sampling	291	1256	OOM
	Par. sampling	53	158	597

Fig. 14. Hint generation time (secs), 2M shopping records

9 Related Work

Entity Resolution has been studied under various names including record linkage [21], merge/purge [12], deduplication [22], reference reconciliation [6], object identification [24], and others (see [7, 28] for recent

surveys). Most of the ER works have focused on optimizing the overall runtime. In contrast, our approach takes a pay-as-you-go approach that optimizes the *intermediate results* of ER. Our approach is useful when either the data set is too large to resolve within a reasonable amount of time, or when there is a work/runtime limit on resolving even just a few records.

Blocking techniques [18, 2, 12] focus on improving the overall runtime of ER where the records are divided into possibly overlapping blocks, and the blocks are resolved one at a time. Locality sensitive hashing [8] is a method for performing probabilistic dimension reduction of high-dimensional data and can also be used as a blocking technique. A number of works [1, 5] propose efficient similarity joins. Our pay-as-you-go techniques improve blocking by also exploiting the ordering of record pairs according to their likelihood of matching to produce the best intermediate ER results.

A number of ER works [6, 25] implicitly use hints by comparing record pairs in the order of their similarity. More recently, a framework for clustering records based on similarity join results [11, 10] has been proposed. Here, the duplication detection framework consists of two stages: an efficient similarity join, which returns similarities between likely matching records, and a clustering stage where records are clustered based on the given similarities. While these systems may already use hints, we believe our work is the first to explicitly identify and study a wide range of hints that yield results early. Given the explosion of data around us, we believe that many future ER systems will benefit from early termination techniques that try to make the maximum progress possible using limited time and resources.

Another line of works propose similarity search [30, 4, 29] techniques where indices or blocking criteria are used for quickly finding the records that are likely to match with a single record. In contrast, our work focuses on resolving all the records (instead of just one) by using hints, which provide information on the best record pairs that are more likely to match.

There has been a recent surge of works on pay-as-you-go information integration [16, 15] on large scale data. Our works are in the same spirit of these works where we incrementally resolve records given the limited amount of time and resources we have. Our work focuses on the ER domain and improves existing ER algorithms to produce results in a pay-as-you-go fashion using hints.

10 Conclusion

We have proposed a pay-as-you-go approach for Entity Resolution (ER) where given a limit in resources (e.g., work, runtime) we attempt to make the maximum progress possible. We introduce the novel concept of hints, which can guide an ER algorithm to focus on resolving the more likely matching records first. Our techniques are effective when there are either too many records to resolve within a reasonable amount of time or when there is a time limit (e.g., real-time systems). We proposed three types of hints that are compatible with different ER algorithms: a sorted list of record pairs, a hierarchy of record partitions, and an ordered list of records. We have also proposed various methods for ER algorithms to use these hints. Our experimental results evaluated the overhead of constructing hints as well as the runtime benefits for using hints. We considered a variety of ER algorithms and two real-world data sets. The results suggest that the benefits of using hints can be well worth the overhead required for constructing and using hints. We believe our work is one of the first to define pay-as-you-go ER and explicitly propose hints as a general technique for fast ER. Many interesting problems remain to be solved, including a more formal analysis of different types of hints and a general guidance for constructing and updating the "best" hint for any given ER algorithm.

APPENDIX

A Proofs

A.1 Hierarchy of record partitions

PROPOSITION 1 *Algorithm 1 returns a valid hint.*

Proof. A higher level partition P is always coarser than a lower level partition P' because a higher threshold is used to split records in the sorted list. Hence, $P' \leq P$. Since higher level partitions is always coarser than lower level partitions, H is a valid hint according to Definition 3.

PROPOSITION 2 *Given a valid ER algorithm E, Algorithm 3 returns a correct ER result when $P_L = \{R\}$ and W is unlimited.*

Proof. Given that W is never satisfied, E can always run to the end. Since $P_L = \{R\}$, the final result $F = E(R_p)$, which is the correct result by definition.

PROPOSITION 3 *Suppose we have a partition $S = \{s_1, \ldots, s_n\}$ of R_p. That is, $\bigcup s_i = R_p$ and for any $1 \leq i, j \leq n$ where $i \neq j$, $s_i \cap s_j = \emptyset$. Then given a general incremental ER algorithm E, $F = E(\bigcup_{i=1\ldots n} E(s_i)) \in \bar{E}(R_p)$.*

Proof. We show that, if E satisfies the general incremental property, then $E(\bigcup_{i=1\ldots n} E(s_i)) \in \bar{E}(R_p)$. We define the following two notations: $\alpha(k) = \bigcup_{s \in R_p - \{s_1,\ldots,s_k\}} E(s)$ and $\beta(k) = \bigcup_{s \in \{s_1,\ldots,s_k\}} s$. To prove that $F = E(\alpha(0)) \in \bar{E}(R_p) = \bar{E}(\beta(|S|))$, we prove the more general statement that $F \in \bar{E}(\alpha(k) \cup \beta(k))$ for $k \in \{0,\ldots, |S|\}$. Clearly, if our general statement holds, we can show that $F \in \bar{E}(\beta(|S|)) = \bar{E}(R_p)$ by setting $k = |S|$.

Base case: We set $k = 0$. Then $F = E(\alpha(0)) \in \bar{E}(\alpha(0)) = \bar{E}(\alpha(0) \cup \beta(0))$.

Induction: Suppose that our statement holds for $k = n$, i.e., $F = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n))$. We want to show that the same expression holds for $k = n + 1$ where $n + 1 \leq |S|$. We use the general incremental property by setting $P_1 = s_{n+1}$ and $P_2 = \alpha(n+1) \cup \beta(n+1)$. The first condition $P_1 \subseteq P_2$ is satisfied because $\beta(n+1)$ contains P_1. We then set $F_1 = E(P_1) = E(s_{n+1})$ and $F_2 = E(F_1 \cup (P_2 - P_1)) = E(E(s_{n+1}) \cup \alpha(n+1) \cup \beta(n)) = E(\alpha(n) \cup \beta(n))$. The general incremental property tells us that $F_2 \in \bar{E}(P_2) = \bar{E}(\alpha(n+1) \cup \beta(n+1))$. Thus, any $E(\alpha(n) \cup \beta(n)) \in \bar{E}(\alpha(n+1) \cup \beta(n+1))$. Using our induction hypothesis, we conclude that $F = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n)) \subseteq \bar{E}(\alpha(n+1) \cup \beta(n+1))$.

PROPOSITION 4 *If E is general incremental (satisfying Definition 5), Algorithm 4 correctly returns an ER result $F \in \bar{E}(\{\{r\} | r \in c\})$.*

Proof. In the case where $R_p \neq \emptyset$ in Step 5, we have $R_p = \bigcup_{s \in c.ch} E(s)$. By Proposition 3, the final ER result $E(R_p) = E(\bigcup_{s \in c.ch} E(s)) \in \bar{E}(c)$. Otherwise, if $R_p = \emptyset$ in Step 5, then again by Proposition 3, $E(R_p) = E(\{\{r\} | r \in c\} = E(\bigcup_{r \in c} E(\{r\})) \in \bar{E}(c)$.

PROPOSITION 5 *The HC_S algorithm is general incremental.*

Proof. We first define the notation of connectedness for HC_S. Two records r and s are connected under D, T, and R_p if there exists a sequence of records $[r_1\ (= r), \ldots, r_n\ (= s)]$ where for each pair (r_i, r_{i+1}) in the path, either $D(r_i, r_{i+1}) \leq T$ or $\exists c \in R_p$ s.t. $r_i \in c, r_{i+1} \in c$.

We now prove the following Lemma.

Lemma 1. *Two records r and s are connected under D, T, and R_p if and only if r and s are in the same cluster in $E(R_p)$ using the HC_S algorithm.*

Proof. Suppose that r and s are in the same cluster in $E(R_p)$. If r and s are in the same cluster in R_p, then r and s are trivially connected under D, T, and R_p. Otherwise, there exists a sequence of merges of the clusters in R_p that grouped r and s together. When two clusters c_i and c_j in R_p merge, all the records in c_i and c_j are connected by transitivity because two records within c_i or c_j are trivially connected and there exists at least one pair of records from c_i and c_j whose distance according to D does not exceed T. Furthermore, for any two clusters (not necessarily in R_p) that merge, all the records in the two clusters are also connected by transitivity. Since the clusters containing r and s merged at some point, r and s are thus connected under D, T, and R_p. Conversely, suppose that r and s are connected as the sequence $[r_1(= r), \ldots, r_n(= s)]$ under D, T, and R_p. If r and s are in the same cluster in R_p, they are already clustered together. Otherwise, all the clusters that contain $r_1, \ldots, r_n$ eventually merge together according to the HC_S algorithm, clustering r and s together.

Lemma 1 directly implies that HC_S returns a unique solution. Suppose that there are two possible solutions for $E(R_p)$: F_1 and F_2. Without loss of generality, suppose that the records r and s are in the same cluster in F_1, but not so in F_2. Then r and s are connected under D, T, and R_p according to F_1 and Lemma 1, but not connected according to F_2, which is a contradiction.

We now prove that the HC_S algorithm is general incremental. In Definition 5, suppose that the three conditions hold, i.e., $P_1 \subseteq P_2$, $F_1 \in \bar{E}(P_1)$, and $F_2 \in \bar{E}(F_1 \cup (P_2 - P_1))$. Since HC_S returns a unique solution regardless of the order of records resolved, the ER results $E(P_2)$ and $E(F_1 \cup (P_2 - P_1))$ are both unique.

A.2 Ordered List of Records

PROPOSITION 6 *Algorithm 5 returns an optimal record list.*

Proof. We first prove that $nEntities(S) = \Sigma_{r \in S} nEntities(\{r\})$. The reason is that $nEntities(S) = \Sigma_{j=1...L}(\frac{1}{L} \times \Sigma_{c \in P_j} \frac{|S \cap c|}{|c|}) = \Sigma_{r \in S} \Sigma_{j=1...L}(\frac{1}{L} \times \Sigma_{c \in P_j} \frac{|\{r\} \cap c|}{|c|}) = \Sigma_{r \in S} nEntities(\{r\})$. Now given a prefix S of R sorted by $nEntities$ values, we show that $nEntities(S) \leq nEntities(S')$ for any $S' \subseteq R$. Suppose that there exists an $S' \subseteq R$ such that $nEntities(S) > nEntities(S')$ while $|S| = |S'|$. Then there exists an $r' \notin S$ where $nEntities(\{r'\})$ is smaller than the $nEntities$ value of the $|S|$th record in the record list. However, we have just contradicted the fact that the record list is sorted by the $nEntities$ values.

References

1. A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
2. R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proc. of ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.
3. O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.
4. S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.
5. W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Trans. Inf. Syst.*, 18(3):288–321, 2000.
6. X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, pages 85–96, 2005.
7. 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.
8. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.
9. J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Applied Statistics*, 18(1):54–64, 1969.
10. O. Hassanzadeh, F. Chiang, R. J. Miller, and H. C. Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.
11. O. Hassanzadeh and R. J. Miller. Creating probabilistic databases from duplicated data. *VLDB J.*, 18(5):1141–1166, 2009.
12. M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of ACM SIGMOD*, pages 127–138, 1995.
13. P. Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.
14. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.
15. S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD Conference*, pages 847–860, 2008.
16. J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, 2007.
17. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

18. A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of ACM SIGKDD*, pages 169–178, Boston, MA, 2000.
19. D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results. *PVLDB*, 2010.
20. H. B. Newcombe and J. M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *CACM*, 5(11):563–566, 1962.
21. H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.
22. S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD*, Edmonton, Alberta, 2002.
23. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
24. S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8):635–656, 2001.
25. M. Weis and F. Naumann. Detecting duplicates in complex xml data. In *ICDE*, page 109, 2006.
26. S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *PVLDB*, 3(1):1326–1337, 2010.
27. S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, pages 219–232, 2009.
28. W. Winkler. Overview of record linkage and current research directions. Technical report, U.S. Bureau of the Census, Washington, DC, 2006.
29. W. Yancey. Bigmatch: A program for extracting probable matches from a large file for record linkage. Technical report, US Bureau of the Census, 2002.
30. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer, 2005.