Pay-As-You-Go Entity Resolution

Steven Euijong Whang, David Marmaros, and Hector Garcia-Molina, Member, IEEE

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.

Index Terms—Entity Resolution, Pay-as-you-go, Data cleaning

1 INTRODUCTION

Entity resolution [1], [2] (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 news feed entity matching algorithm may have very limited time for resolving company names and individuals in a stock market trading scenario where financial data is generated with high frequency.

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.

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. Also, hints are tools that may or may not be compatible with a specific ER algorithm and are not interchangeable modules.

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.

Again, we cannot give a comprehensive answer but we do illustrate the use of different types of hints in several representative instances.

In summary, our contributions 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 [3], which is a standard technique for scaling ER.
- We propose three types of hints: a sorted list of record pairs, a hierarchy of likely record partitions, and an ordered list of records (Sections 3–5). 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 experimentally evaluate how applying hints can help ER do more work up front (Section 7). 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 ER, 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 output $E(R)$ is a partition of the input that groups together records describing 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 $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, $M(E(R)) = \frac{2}{3}$.

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 record pairs to compare ($\frac{n(n-1)}{2}$ 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 [3], [4], [5], [6]. 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) [6] 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.

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 2.1:** 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) \( t_g \) 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 4.1) 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 sections we describe each hint type in detail and illustrate how it can be incorporated into an ER algorithm that is amenable to that type of hint. For simplicity we will focus on processing a single block of records (although as noted earlier a single hint could span multiple blocks). In our technical report [7], 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.

### 3.1 Use

We now discuss how an ER algorithm can use a pair-list hint. While the details of usage depend on the actual ER algorithm used, 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 [4] (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 cluster the matching pairs of records to 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 \( s \), \( e(r, s) = |\text{Rank}(r) - \text{Rank}(s)| \) where \( \text{Rank}(r) \) 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 \( \text{Rank}(r_3) = 1, \text{Rank}(r_2) = 2, \) and \( \text{Rank}(r_1) = 3 \). 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.

### 3.2 Generation

We first discuss how to generate pair-list hints using cheaper estimations. We then discuss a more general technique that does not require application estimates.

#### 3.2.1 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 \( O(|R|^2) \).

3.2.2 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 7, it can yield
surprisingly good estimates in some cases.

The basic idea is to use the expensive function \( d \) to
calculate 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

\[
\frac{(\frac{1000}{2}) + 990 \times 10}{10000} = \frac{9945}{10000} \approx 2\%.
\]

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 \( e(r, s) = \sum d(r, t) - d(t, s) \|^2 \). 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
calculate 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 our technical report [7]).

\[
\begin{align*}
P_3 & : \{r_1, r_2, r_3, r_4, r_5\} \\
P_2 & : \{r_1, r_2, r_3\}, \{r_4, r_5\} \\
P_1 & : \{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}
\end{align*}
\]

Fig. 3. A partition hierarchy hint for resolving \( R \)

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 4.1: 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 4.2: 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 4.3: The children of a cluster \( c \) (denoted as
\( c.ch \)) in the \( i \)th 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 \( \forall c' \in S, c.e \leq c \).

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 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 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 1 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 1 using the Single-link Hierarchical Clustering algorithm [8] (which we call \( HCS \)). The \( HCS \) 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 \( HCS \) 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 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 1 to run the \( HCS \) 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 1, we first resolve the clusters in \( P_1 \) of the hint. Thus, we compare \( r_1 \) with \( r_2 \) by invoking \( \text{Resolve}(E, \{r_1, r_2\}, h) \) 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 then run \( \text{Resolve}(E, \{r_1, r_2, r_3\}, h) \) 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 \( \text{Resolve} \) 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 \( \{\{r_1, r_2\}, \{r_3\}\} \) would have been returned using only one record comparison.

**Proposition 4.4:** Given a valid ER algorithm \( E \), Algorithm 1 returns a correct ER result when \( P_L = \{R\} \) and \( W \) is unlimited.

The complexity of Algorithm 1 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 depends on the implementation of \( \text{Resolve}(E, c, h) \) 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 \( \text{Resolve}(E, c, h) \) 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 \( r_2 \). Details on how the ER algorithm can exploit \( h \) and what properties of ER make this technique feasible can be found in our technical report [7].

4.2 Generation

We propose various methods for efficiently constructing a partition hierarchy. In the following section, we con-
the edit distance does not exceed $P$
Bobby
the first cluster in $P$
Bob
and
Bob
previous record
Bobby
we set two thresholds
represented and sorted by their names). Suppose that
$P$
number of levels
(The thresholds values are pre-specified based on the
partitioning records based on their key value distances.
Algorithm 2 first reads
is created for both
is 2. Since this value is larger than
T
(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
Bobbyj
, the edit distance with the previous record
Bobby
is 4, which exceeds both thresholds. As a result, a new cluster with
Bobbyj
is created for both $P_1$ and $P_2$. The resulting
hint thus contains two partitions: $P_1 = \{\{Bob\}, \{Bobby\}, \{Bobbyj\}\}$ and $P_2 = \{\{Bob, Bobby\}, \{Bobbyj\}\}$.

The following result shows the correctness of Algorithm 2. Proofs for this result and subsequent ones can be found in our technical report [7].

**Proposition 4.5:** Algorithm 2 returns a valid hint.

Given that the input $Sorted$ is already sorted, Algorithm 2 runs in $O(L \times |R|)$ time by iterating all records in $Sorted$ and, for each record, iterating through all thresholds.

### 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. Moreover, it is harder to exploit a sorted list of records than say a sorted list of pairs.

#### 5.1 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 $F$.

The exact way the record list is exploited depends on the given ER algorithm. For example, we consider hierarchical clustering based on a Boolean comparison rule [9] (called $HCB$), which can benefit from record lists. The $HCB$ 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 $HCB$ 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 $HCB$), the $HCB$ algorithm first compares $r_1$ and $r_2$. If we set the work limit $W$ to one record comparison, then $HCB$ will terminate returning $\{\{r_1, r_2\}, \{r_3\}\}$.  

```
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 $H = \{P_1, \ldots, P_k\}$
3: Initialize partitions $P_1, \ldots, P_L$
4: for $r \in Sorted$ do
5:   for $T_j \in T$ do
6:     if $r.prev\.exists() \land KeyDistance(r, r.prev) \leq T_j$ then
7:         Add $r$ into the newest cluster in $P_j$
8:     else
9:         Create new cluster in $P_j$ containing $r$
10: return $\{P_1, \ldots, P_L\}$
```

**ALGORITHM 2:** Generating a partition hierarchy hint from sorted records

struct hints based on sorted records, which are application estimates. In our technical report [7], we discuss how partition hierarchies can also be generated using hash functions (which are application estimates) and sampling (which are not application estimates).

#### 4.2.1 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 2 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 $[Bob, Bobby, Bobby]$ (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 2 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
Bobbyj
, the edit distance with the previous record
Bobby
is 4, which exceeds both thresholds. As a result, a new cluster with
Bobbyj
is created for both $P_1$ and $P_2$. The resulting
hint thus contains two partitions: $P_1 = \{\{Bob\}, \{Bobby\}, \{Bobbyj\}\}$ and $P_2 = \{\{Bob, Bobby\}, \{Bobbyj\}\}$.  

The following result shows the correctness of Algorithm 2. Proofs for this result and subsequent ones can be found in our technical report [7].

**Proposition 4.5:** Algorithm 2 returns a valid hint.

Given that the input $Sorted$ is already sorted, Algorithm 2 runs in $O(L \times |R|)$ time by iterating all records in $Sorted$ and, for each record, iterating through all thresholds.
5.2 Generation

We propose methods for efficiently constructing a list of records. The following section uses a partition hierarchy for generation. In our technical report [7], we also discuss how record lists can be generated using sampling.

5.2.1 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 5.1 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 \(S\) as \(\frac{w_i \times nEntities_j(S)}{\Sigma_j=1..L S_j}\). Since each partition is equally likely to be the answer, we define the overall estimate of the number of entities fully identified by \(S\) as \(\Sigma_j=1..L \frac{w_i \times nEntities_j(S)}{\Sigma_j=1..L S_j}\).

For example, suppose that the partition hierarchy \(H_1\) has 3 levels where \(P_1 = \{\{r_1, r_2\}, \{r_3\}\}, P_2 = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}, \) and \(P_2 = \{\{r_1, r_2, r_3, r_4, r_5\}\}\). Each partition is equally likely to be the ER answer. Suppose that we resolve the set of records \(S = \{r_1, r_2, r_1, r_5\}\), 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 \(\frac{2}{2} + \frac{2}{2} = 2\) 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 \(\frac{1}{2}\). Our overall estimate for the actual number of entities identified \(nEntities(S)\) is thus \(\frac{1}{2} \times (2 + 2 + \frac{1}{2}) = \frac{15}{4}\), 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 \(P_s\), 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 \(\sum_{j=1..L} (w_j \times nEntities_j(S))\). 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 experimental results in our technical report [7]).

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 5.1: A record list \(H\) of \(R\) is optimal if any prefix set \(P\) of \(H\) has a minimum value of \(\sum_{j=1..L} (w_j \times nEntities_j(P))\) among all subsets of \(R\) with size \(|P|\).

Interestingly, we can always generate a record list that is optimal according to Definition 5.1. A key observation is that \(nEntities(S) = \sum_{j=1..L} nEntities_j(S)\), 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 \(S\).

We illustrate the generation of an optimal record list using \(H_1\) from above (the algorithm can be found in our technical report [7]). We first compute the estimated number of entities identified for each record. According to \(H_1\), record \(r_3\) has a \(nEntities\) value of \(\frac{2}{3} \times (\frac{2}{2} + \frac{2}{2} + \frac{1}{2}) = \frac{14}{9}\). Similarly, \(r_1\) and \(r_2\) each have a value of \(\frac{1}{3} \times (\frac{2}{2} + \frac{1}{2} + \frac{1}{2}) = \frac{14}{9}\). Finally, records \(r_4\) and \(r_5\) each have a value of \(\frac{1}{3} \times (\frac{2}{2} + \frac{1}{2} + \frac{1}{2}) = \frac{14}{9}\) (the fractions were not reduced for easy comparison). We now sort the records by their \(nEntities\) values 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 that \(H’\) is optimal, observe that no prefix set of size 2 has a \(nEntities\) value smaller than that of \([r_1, r_2]\), which is \(\frac{14}{9} + \frac{14}{9} = \frac{28}{9}\).

6 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 7.1.2 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.

### 7 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. For our quality metric $M$ we use recall: the fraction of discovered matching record pairs. We do not use precision since our algorithms always find correct record matches (precision is always 1). More experiments on cases where the precision is not 1 can be found in our technical report [7]. Due to space restrictions, we also only show in our technical report how to find the right number of levels in a partition hierarchy hint and how to find the right sample size.

#### 7.1 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.

##### 7.1.1 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 7.4), 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.

##### 7.1.2 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 the Boolean match function $B$ for comparing two records. For shopping records, $B$ compares the titles, prices, and categories. For 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 pair lists using cheap distance functions, we used the estimate function $e(r, s) = |R - \hat{r} - \hat{s}|$ to limit the time and space overhead. When using sampling to generate pair lists, we used a sample of 10 records.

The $HC_S$ algorithm uses the distance function $D$ for comparing two records. For shopping records, $D$ measures the Jaro distance [10] between the titles of two records. For 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 our technical report [7] for more details). 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 our technical report [7], we show that small sample sizes are sufficient for reasonable results.) A partition hierarchy is suitable for the HC algorithm because the hint suggests sets of records to resolve first, and the HC algorithm can easily resolve subsets of records at a time.

The HCB algorithm uses B (defined above) for comparing two records. 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 [11] generated from titles (names) 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., 3 × (|R| − 1)/2 pairs) and generated a record list. For the partial sampling scheme, we only computed and stored the top-5 closest pairs to limit the time and space overhead. In both schemes, we used a random sample of 10 records.

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.

### 7.2 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 3.2.2) and compared the records with the closest estimated distance first. In our experiments, we set the sample size to 10 records. (In our technical report [7], 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 our technical report [7], 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 HCS 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 HCS 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 HCS 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 HCS 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 HCS without
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 7.3.2 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.

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 7.3.2.

7.3 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.

7.3.1 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 decreases).

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.

### 7.3.2 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.

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. Experiments on when hints start to benefit can be found in our technical report [7].

7.4 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 Section 2.1. We used minhash signatures [11] 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 record, 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 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 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 \( \frac{1}{20} \times 20 \times 19 = 19 \) record comparisons.

Figure 7 shows how the two schemes perform compared to when ER runs without hints. We measured the entire ER runtimes including the hint construction times and 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.

### 7.5 Scalability of Generating Hints

Table 8 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 7.2).

<table>
<thead>
<tr>
<th>Hint</th>
<th>Generation</th>
<th>0.5M</th>
<th>1M</th>
<th>2M</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Cheap dist. fns</td>
<td>4</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Sampling</td>
<td>47</td>
<td>120</td>
<td>309</td>
</tr>
<tr>
<td>H</td>
<td>Sorted records</td>
<td>5</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Hash functions</td>
<td>8</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Sampling</td>
<td>40</td>
<td>130</td>
<td>379</td>
</tr>
<tr>
<td>RL</td>
<td>Par. hierarchy</td>
<td>11</td>
<td>19</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Com. sampling</td>
<td>291</td>
<td>1256</td>
<td>OOM</td>
</tr>
<tr>
<td></td>
<td>Par. sampling</td>
<td>53</td>
<td>158</td>
<td>597</td>
</tr>
</tbody>
</table>

![Fig. 7. Runtime (hrs) and recall for different schemes, 2M shopping records](image)

8 RELATED WORK

Entity Resolution has been studied under various names including record linkage, merge/purge, deduplication, reference reconciliation, object identification, and others (see [1], [10] 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 [5], [4] 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 [6] 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 [12], [13] 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 [14], [15] 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 [16] 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.

Several works [17], [18] propose pay-as-you-go information integration 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.

## 9 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.

## References


Steven Euijong Whang is a Computer Science PhD candidate at Stanford University. He received a CS BS from the Korea Advanced Institute of Science and Technology (KAIST) in 2003 and a CS MS from Stanford in 2007.

David Marmaros helped generate data for this paper while studying at Stanford. He currently works as a Software Engineer at Google.

Hector García-Molina is the Leonard Bosack and Sandra Lerner Professor in the Departments of Computer Science and Electrical Engineering at Stanford University. From 1979 to 1991 he was on the faculty at Princeton University. He received a BSEE from the Instituto Tecnologico de Monterrey, Mexico, in 1974, and a CS PhD from Stanford in 1979.