International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064

Fast and Accurate Incremental Entity Relationships

Rajeshkumar S[1], Geofrin Shirly S[2]

[1]School of Engineering, Department of Computer Science and Engineering,
Vels Institute of Science, Technology and Advanced Studies (VISTAS), VELS University, Chennai-600117

[2]Assistant Professor, School of Engineering, Department of Computer Science and Engineering,
Vels Institute of Science, Technology and Advanced Studies (VISTAS), VELS University, Chennai-600117

Abstract: ***Entity resolution (ER) is the problem of identifying which records in a database refer to the same entity. This project 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. This project introduces 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, this project illustrates the potential gains of our pay-as-you-go approach compared to running ER without using hints.***

Keywords: Entity resolution, data cleaning

1. Introduction

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.

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 Ear is a partition of the input that groups together records describing the same real-world entity. For example, the output F ¼ ffr$_1$; r$_3$g; fr$_2$g; fr$_4$; r$_5$; r$_6$gg 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., ffr$_1$g; fr$_2$g; fr$_3$g; fr$_4$g; fr$_5$g; fr$_6$gg.

We denote the ER result of E on R at time t as EðRÞ½t&. In the above example, if E has grouped {r$_1$} and {r$_3$} after 5 seconds, then EðRÞ½5& ¼ ffr$_1$; r$_3$g; fr$_2$g; fr$_4$g; fr$_5$g; fr$_6$gg. We denote the total runtime of Ear as T be; RÞ. Qualities 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 Ear ¼ ffr$_1$; r$_2$; r$_3$g; fr$_4$gg and the correct clustering is ffr$_1$; r$_2$g; fr$_3$g; fr$_4$gg; ¼ $\frac{1}{3}$.

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 appears "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 ($\frac{n(n-1)}{2}$ 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. Locality sensitive hashing (LSH) [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 fr$_1$ _ r$_2$, r$_1$ _ r$_3$, r$_2$ _ r$_3$, r$_4$ _ r$_5$ . . .g. 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, and 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.)

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 dry; so 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 mar; so 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 dry; so < T and other conditions then true," where T is a threshold.

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 mar; so or dark; so 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.

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 ear; so 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 ear; so 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 reinsert r into the heap with that new estimate. The space requirement in this case is proportional to jar, the number of records. On the other hand, if we store all pairs of records in the heap, the space requirement is order of OðjRj^{2}Þ.

3.2.2 Application Estimate Not Available

In some cases, there may be no known inexpensive application specific estimate function ear; so. 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 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 jest is significantly smaller than the total number of records or j, the number of real distances measured is much smaller than the total number of pair wise distances.

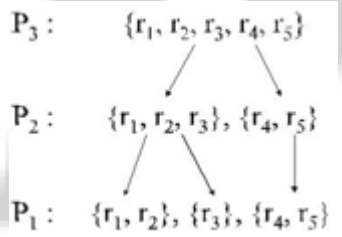


Figure 1: A partition hierarchy hint for resolving R

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 dark; to and dot; so for each t 2 S. formally, the estimate ear; so ¼ $_{t2S}$ðdðr; to _ dot; sÞÞ^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 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 jest 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 jest records works reasonably well (see our technical report [7]).

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 coarser 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.

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).

4.2 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. 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; Bobbi] (the records are represented and sorted by their names). Suppose that we set two thresholds T_1 ¼ 1 and T_2 ¼ 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 Bobbi, the edit distance with the previous record Bobby is 4, which exceeds both thresholds. As a result, a new cluster with Baoji is created for both P_1 and P_2. The resulting hint thus contains two partitions.

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 nontrivial 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^0 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 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 ¼ $fr_1; r_2; r_3g$ (which can

also be viewed as a list of three singleton clusters) and the comparison function B where Bðr$_1$; r$_2$Þ ¼ true, Bðr$_2$; r$_3$Þ ¼ true, but Bðr$_1$; r$_3$Þ ¼ 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 Bðr$_1$; r$_3$).

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; . . . ; 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

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.

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 2.4 GHz Intel(R) Core 2 processor with 4 GB of RAM.

7.1.1 Real Data

The comparison shopping data set 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 data set provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbits.-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 data sets fit in memory, the 2 million shopping data set did not fit in memory and had to be stored on disk.

7.1.2 Hints and ER Algorithms

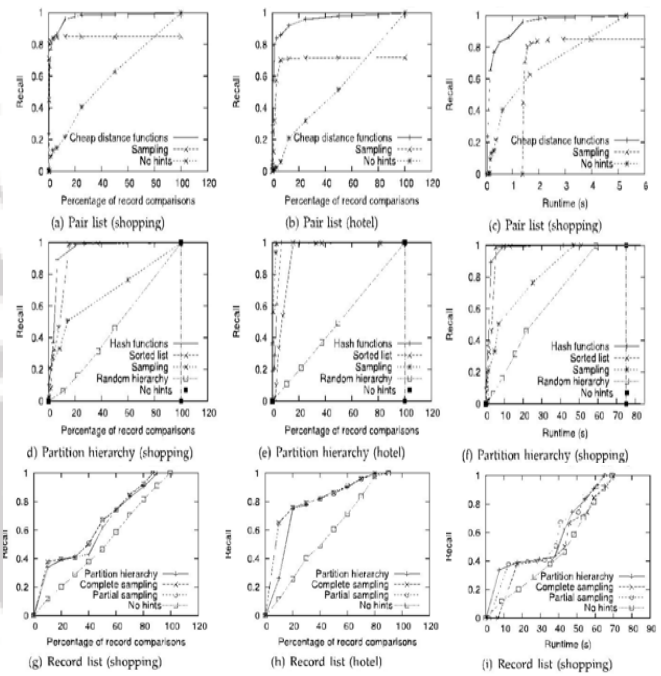
For our experiments we use the three ER algorithms used to illustrate our hints (and summarized earlier in Fig. 4). In this section, we provide some implementation details for the ER

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 ear; so ¼ using the title (name) attributes of shopping (hotel) records as the sort key. Using e, we only computed and stored the top-(δ_w _ 1Þ _ jRj _ $^{w\_\delta w\_1Þ}_{2}$) 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. 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 Jar distance [10] between the titles of two records. For hotel records, D measures the Jar 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

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. Fig. 5a 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 pair wise 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 percent of the record comparisons required when running SN without hints. The quality of SN using the sampling technique achieves 0.8 recalls with 0.78 percent of the entire work. While the sampling techniques give 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). 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 recalls with only 12.5 percent 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 percent of the entire work. While the sampling techniques give a high recall early on, it does not give 1.0 recall even after performing as many comparisons as the SN algorithm without hints.



5d 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 Fig. 5d) shows the progress of the original HC_S algorithm where records are clustered only after all pairs of base records are percent 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 percent of the total comparisons of HC_S without hints. Finally, when using a hint generated using hash functions,

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.

Hint	Generation	Time OH		Space OH(Const/Use)	
		Sho3K	Ho3K	Sho3K	Ho3K
PL	Cheap dist. fns	0.005	0.19	22 / 22	7.8 / 7.8
	Sampling	0.16	3.56	22 / 22	7.8 / 7.8
H	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
RL	Par. hierarchy	7E-4	0.01	0.08 / 0	0.03 / 0
	Com. sampling	0.09	1.07	349 / 0	119 / 0
	Par. sampling	0.02	0.31	1.15 / 0	0.4 / 0

The usage time overhead for accessing a partition hierarchy is an iteration of the clusters from the "Time Overhead" column in Fig. 6 shows the construction time overhead for each type of hint in Fig. 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 0:0001_ the time for running the HC_S algorithm without hints.

7.4 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.

8. Conclusion

We have proposed a pay-as-you-go approach for 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

[1] A.K. Elmagarmid, P.G. Ipeirotis, and V.S. Verykios, "Duplicate Record Detection: A Survey," IEEE Trans. Knowledge Data Eng., vol. 19, no. 1, pp. 1-16, Jan. 2007.
[2] A.K. Jain, M.N. Murty, and P.J. Flynn, "Data Clustering: A Review," ACM Computing Surveys, vol. 31, no. 3, pp. 264-323, 1999.
[3] H.B. Newcombe and J.M. Kennedy, "Record Linkage: Making Maximum Use of the Discriminating Power of Identifying Information," Comm. ACM, vol. 5, no. 11, pp. 563-566, 1962.
[4] M.A. Herna´ndez and S.J. Stolfo, "The Merge/Purge Problem for Large Databases," Proc. ACM SIGMOD Int'l Conf. Management of Data, pp. 127-138, 1995.
[5] A.K. McCallum, K. Nigam, and L. Ungar, "Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching," Proc. ACM Sixth SIGKDD Int'l Conf. Knowledge Discovery and Data Mining, pp. 169-178, 2000.
[6] Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," Proc. 25th Int'l Conf. Very Large Databases (VLDB), pp. 518-529, 1999.
[7] S.E. Whang, D. Marmaros, and H. Garcia-Molina, "Pay-As-You-Go Entity Resolution," technical report, Stanford Univ., available at http://ilpubs.stanford.edu:8090/979/, 2012.
[8] C.D. Manning, P. Raghavan, and H. Schu¨tze, Introduction to Information Retrieval. Cambridge Univ. Press, 2008.

Author Profile

Rajeshkumar S received the bachelor's degree B.Tech IT from Dr. M. G. R. Educational and Research Institute and University during the year of 2007-2011 And now currently doing M.E., CSE in Vels Institute of Science, Technology and Advanced Studies (VISTAS) VELS University during 2012-2014 batch.