Entity Resolution with Heavy Indexing*

Csaba István Sidló

Data Mining and Web Search Group, Informatics Laboratory
Institute for Computer Science and Control, Hungarian Academy of Sciences
sidlo@ilab.sztaki.hu

Abstract. Entity resolution (ER), or deduplication is a computationally hard problem with $O(n^2)$ time complexity. We reformulate ER as a search problem, and develop algorithms using efficient indices. Indices can enhance algorithm scalability, facilitate distributed processing, but require additional storage space. We study the performance and tradeoffs between index update and search in ER algorithms, and show that significant performance gain can be obtained by using indices. We also demonstrate the strength of our algorithms in the real-world scenario of an insurance customer master data creation procedure.

1 Introduction and Related Work

Entity Resolution (ER) is the process of identifying groups of records that refer to the same real-world entity (eg. [2, 9]). The first description of the record linkage problem appears in Fellegi and Sunter [5] in 1969 who use a probabilistic model. Since then, the process was described in many different contexts under many different names including duplicate detection, instance identification, merge/purge, reference reconciliation, etc. Entity resolution can be formulated in many different ways and appears in a wide range of applications. In [4] a survey is given on duplicate record detection, who describes supervised, unsupervised and active learning, and summarizes statistical and machine learning solutions based on various text similarity and matching measures.

Generic entity resolution with black-box match and merge functions was first described in [1], where resolution means the closure of the original entity set according to these functions. Simple feature indices are also used. We give generic ER algorithms for relational databases in [10]. Entity resolution as a hypergraph clustering problem can be found in [2], under the name of relational clustering.

When building a client database, companies typically face the entity resolution problem. Clients may appear multiple times in multiple source systems, e.g. the same person may appear in several marketing databases obtained by different means.

ER is the key step in producing sound and clean client master data. Client records may consist of attributes, both of persons (birth data, tax and social

* This work was supported by the EU FP7 Project SCIIMS (Strategic Crime and Immigration Management System).

security numbers, postal address, etc.) and of organizations (client ID, contract number). Attribute values are often missing or erroneous, and some attributes change in time (name, postal address).

By resolving the record set, simple but fundamental as well as more complex questions can be answered: How many clients we actually have? Can a given client be addressed in a marketing campaign, or we just made an offer a few days ago? Does a new client have ever contacted, or had any transaction with our company?

Several new results were published recently. A new approach can be found in [13]: entity behavior is recorded as transactional log. Common patterns of these transactions are used to identify similar or identical entities. Measuring the quality of entity resolution results is a crucial problem: the possible quality metrics are considered in [9]. As an algorithmic aspect, core ER algorithms are enhanced by combining the results of different blocking strategies in [12]. Our formal model of entity resolution is similar to the model used in [12] when forming dominated record partitions. When finding duplicates, the role of constraints is exploited in [6]. Whang and Garcia-Molina [11] deals with the effect of match/merge rule evolution, and gives methods to preserve results when rules change. In [3], special inverted indices are built to speed up ER with blocking. Entity resolution frameworks are developed, like SERF, MTB, DDUpe and MARLIN (see [7]). A practical comparison of ER approaches can be found in [8] using the FEVER framework.

Our main contributions are as follows. First, we define a model suited to a search-based resolution process and practical aspects. Second, we give efficient algorithms with different indexing schemes and a blocking method capable of improving scalability. Third, we demonstrate usability and scalability of our algorithms.

2 Problem Formulation

Entities of the real world are typically hidden and only indirect observations are recorded in a database. This intuition is formalized in the ER framework as follows. Let a set of **records** be $R = \{r_1, r_2, ...r_m\}$, where each r_j consists of a set of **attributes**. The goal is to partition records according to the entities they belong: let $E = \{e_1, e_2, ...e_n\}$ be a set of **entities**, each e_i consisting of a subset of records $e_i \subseteq R$ such that the union of the entities covers all records, $\cup_{i=1}^{n} e_i = R$, and no record belongs to more than one entity: $r \in e_i \wedge r \in e_j \Rightarrow i = j$.

To give an example, consider records and attributes in a client database stating that "the name of the client is John Doe" or "the date of birth is January 1, 1977". An entity can have more than one attribute values, for example multiple names may exist for a real-world client. **Features** form a set of functions $F = \{f_1, f_2, ...f_m\}$, each f_i mapping entities to feature subsets. In an ER algorithm, entities e_1 and e_2 are merged if a matching feature f_i is found, denoted by $f_i(e_1) \sim f_i(e_2)$. The notion of two entities **matching** along feature f_i is defined depending on the application as a function of the two attribute subsets $f_i(e_1)$

and $f_i(e_2)$. For example, we may require $f_i(e_1) \cap f_i(e_2) \neq \emptyset$ but the definition may involve distance functions over the attribute space as well.

Use of merged, representative records as entities is a common practice. Merged records can be however impractical and hard to construct. Deciding feature matches is sometimes impossible without referring to all the feature values, not only a representative value. We think that ER models representing an entity with record sets are favorable.

Definition 1. *Let the* ***entity resolution*** *of an entity set E be another set of entities $ER(E)$, where*

$$\forall e_1, e_2 \in E, e_1 \sim e_2 \Rightarrow \exists e' \in ER(E) : e_1 \subseteq e' \wedge e_2 \subseteq e',$$

$$\forall e \in E \Rightarrow \exists e' \in ER(E) : e \subseteq e'.$$

Thus, the entity resolution is a refined partitioning of the original entity set, where no more separate but matching entities can be found.

We use feature indices to identify possible entity matchings. A feature index provides candidates for a given entity: if there exists an entity with matching feature, the feature index must include it in the result.

Definition 2. *For feature f and entity $e \in E$, let the* ***feature index*** *$index_f(e)$ be the subset of entities E such that*

$$\textit{if } f(e_1) \sim f(e_2), \textit{ then } e_2 \in index_f(e_1) \textit{ and } e_1 \in index_f(e_2).$$

Given two entities e_1 and e_2 and features $f_1, f_2, \ldots, f_n$, the match condition is $f_1(e_1) \sim f_1(e_2) \vee f_2(e_1) \sim f_2(e_2) \vee \ldots \vee f_n(e_1) \sim f_n(e_2)$. The next completeness property formalizes if a set of feature indices can be used to find all matching candidates:

Definition 3. *A feature index set $\{index_i | i = 1, ..., m\}$ is a* ***complete index set*** *for f_j $(j = 1, ..., n)$ features and for E entities, if*

$$\forall e_1, e_2 \in E, k \in [1, n] : f_k(e_1) \sim f_k(e_2) \Rightarrow$$

$$\exists l \in [1, m] : e_2 \in index_l(e_1) \wedge e_1 \in index_l(e_2).$$

The completeness property for given entities states that there exists at least one feature index candidate for all possible match. A trivial complete index set contains only one index, returning the whole E as candidates.

3 Algorithms

The following algorithms all solve the ER problem. They differ how they build and use indices. Indexing solutions are not necessary faster than non-indexing variants: we investigate the efficiency of different index realizations later.

In what follows, we consider the input of the ER algorithms to be the set of records as entities, with each record corresponding to its unique own entity; as the output, some records will be merged to form a smaller size entity set E'. The algorithms may also work with partially merged records as entities, as long as only matching records are merged.

Algorithm 1 Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

```
1: E' ← ∅
2: merged ← null
3: while E ≠ ∅ ∨ merged ≠ null do
4:    if merged ≠ null then
5:       e ← merged
6:    else
7:       e ← an element from E
8:       remove e from E
9:    candidates ← ∪_f index_f(e)
10:   merged ← null
11:   if candidates = ∅ then
12:      E' ← E' ∪ {e}
13:      for all indices: add e to index_f
14:   else
15:      for all c ∈ candidates do
16:         if c ∼ e then
17:            merged ← merged ∪ {c} ∪ {e}
18:            remove c from E'
19:            remove c from all index_f
```

3.1 Basic Feature Indexing

IndexER (Algorithm 1) is our basic indexing solution to the ER problem where feature indices are handled as search data structures. IndexER maintains a result set E' containing no unexplored matches and extends this set by entities from the input. Feature indices contain only entities of E', therefore have to be updated when E' changes.

E' always contains the resolution of the processed entities, and while E diminishes, IndexER solves the $ER(E)$ problem. Efficiency of the algorithm depends both on the indexing tools used, and on properties of the input data set (eg. how many matching records it contains), as well as on the match logic (how many features there are, are they similarity-based etc.). We explore some aspects of performance later.

Entities may usually have feature values that cannot be matched. If for example the telephone number of a given person in a customer database is unknown, then it is not possible to find any matching entity based on this feature.

Definition 4. *Let* $satisfactory_f(e)$ *be a true or false valued function for entities* $e \in E$ *and features* f *such that*

$$satisfactory_f(e) = false \text{ if } \nexists e' \in E : f(e) \sim f(e').$$

A satisfactory function can be defined based on heuristics of the given feature and match logic, for example by filtering incomplete or empty attribute values.

Algorithm 2 Pre-Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

1: $E' \leftarrow \emptyset$
2: $merged \leftarrow null$
3: prepare all $index_f$ with E
4: **while** $E \neq \emptyset$ or $merged \neq null$ **do**
5: **if** $merged \neq null$ **then**
6: $e \leftarrow merged$
7: **else**
8: $e \leftarrow$ an element from E
9: remove e from E
10: $candidates \leftarrow \cup_{f:satisfactory_f(e)}\{e' \in E' : e'$ originates from an $index_f(e)$ entity $\}$
11: $merged \leftarrow null$
12: **if** $candidates = \emptyset$ **then**
13: $E' \leftarrow E' \cup \{e\}$
14: **else**
15: **for all** $c \in candidates$ **do**
16: **if** $c \sim e$ **then**
17: $merged \leftarrow merged \cup \{c\} \cup \{e\}$
18: remove c from E'

Using $satisfactory$ index lookup and update (Line 9 and 13) become conditional, dealing only with $satisfactory_f(e)$ features.

3.2 Feature Pre-indexing

Index updates are usually expensive. Note that records do not change during the resolution process: Algorithms only re-partition records when merging entities. If we build feature indices in a batch for all records before the resolution process, we can save index maintenance costs.

Definition 5. *Feature f is a **stable feature** for an entity set E, if*

$\forall e_1, e_2 \in E$ *with* $f(e_1) \sim f(e_2)$ *satisfies* $f(e_1) \sim f(e_1 \cup e_2)$ *and* $f(e_2) \sim f(e_1 \cup e_2)$.

We can build the index for a stable feature preliminary. At the beginning each entity consists of a single record, and feature indices refer to that initial entity. Additional data structures are needed to track entity merges, and to record if an entity belongs to the resolved set. Pre-Index-ER (Algorithm 2) implements this indexing scheme.

3.3 Feature-based Blocking

Blocking is a proven method to speed up ER algorithms. Blocking divide records into smaller subsets based on expert heuristics including ZIP code, first letter of

Algorithm 3 Block-Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

1: $E' \leftarrow \emptyset$
2: **for all** f feature **do**
3: $\quad B_f \leftarrow$ partition E' according to f
4: $\quad$ **for all** B_f^i partition in B_f **do**
5: $\quad\quad$ update E' with $ER(B_f^i)$

family names, etc. ER is performed on the smaller subsets more easily, especially when they fit in the memory. We speed up processing, but may miss potential matching pairs between different blocks. One solution is to use multiple blocking criteria, and combine the results. Another potential solution is iterative blocking [12], where merged entities are delegated to other affected blocks.

Block-Index-ER (Algorithm 3) implements a new partitioning scheme different from both multiple and iterative blocking. We iterate through all features, partition the input set in every round, and solve the sub-problem with an arbitrary ER algorithm, e.g. Index-ER. In Line 3 a feature-based partitioning is performed. We form up blocks according to feature boundaries:

Definition 6. *$B_f^i(i = 1..n)$ is a **feature-based blocking** of E entity set with f feature, if*

$$\forall i \in [1..n] : e \in B_f^i \Rightarrow \nexists e' \in B_f^j, j \in [1..n], j \neq i : f(e) \sim f(e'),$$

$$\cup_{i \in [1..n]} B_f^i = E$$

In Line 5 the algorithm applies all entity merges to E'. If an entity does not exists in E', it is appended. The algorithm iterates through all blocks of a feature-based blocking and applies every merge. Matching entities always fall into a common block for some of the stable features, therefore at the end Block-Index-ER produces $ER(E)$. We have to process all partitions for all features, but the number of these partitions is relatively small for real-world problems.

4 Index Realizations

Efficiency of Index-ER algorithms depends on properties of the input entity set, the features used and the indexing methods and tools chosen. Next we briefly examine a few alternatives.

The most useful and simplest algorithm variations handle features as attribute value sets, match operators as equality tests. For example, two clients match if they share a birth date, a birth name and a postal address. Conventional search structures are applicable in this scenario. B-trees for example are proven to be optimal and useful general search constructs.

For features based on multiple attributes an index can be built for an arbitrary attribute with good selectivity. Eg. for a complicated birth data feature a birth name B-tree index may be used, if birth name is always known. An other possibility is to use multidimensional indices, eg. R-trees. With R-trees we can use multiple attributes as search key.

Scalability can be improved by relying on external memory indexes that become slower when running out of cache memory, but keep serving the algorithm. Another possibile enhancement is the use of various distributed key-value stores or indexes.

When using similarity-based or probabilistic features and match conditions, the indices provide entities with similar feature value beyond a given similarity threshold. Examples include finding duplicated web pages, using features based on geographic location and distance, features with name similarities etc. Similarity-based indices are useful in this case, and are exhaustively investigated topics. Document Q-gram and TF-IDF indices for example have multiple efficient standalone implementations, for distributed environments as well.

5 Experiments

Experiments were performed on a Linux server containing an AMD 2 GHz Opteron CPU, 7 GB of main memory and a 7200 RPM disk without RAID. We used PostgreSQL 8.4 using 1GB memory, Berkeley DB Java Edition (BDB) 4.1 using 500 MB cache, Sun JDK 1.6 with 3 GB maximum heap size. A relatively weak hardware architecture was chosen intentionally: behavior of the algorithms and indexing schemes become problematic and therefore interesting when reaching the given constraints (eg. the memory limit).

Algorithms were implemented using Java, input and output were stored in PostgreSQL. We note that our solution works also based on various other indexing tools not covered in this paper, including Project Voldemort, Kyoto Cabinet, ScalienDB, etc. We performed repeated executions and averaged the results. Time needed for input read and output write are not included.

For exeriments, we choose the data set of the AEGON Hungary Insurance Ltd.[1] containing approximately 20 million client records. Records consist of both personal attributes (names, birth data, tax number, etc.) and company-dependent identifiers. According to preliminary estimates and experimental results each client has 1.95 records in average. We used random sampling to obtain smaller subsets and selection heuristics to influence the average record count per user (eg. selecting records for the family name 'Smith').

Match logic provided by experts included simple attribute-equality testing, e.g. "two entities match, if they have common tax numbers", and more complicated ones. For example, the birth data feature used 5 attributes, with multiple attribute-value equality testing.

[1] The AEGON Hungary has been a member of the AEGON Group since 1992, one of the world's largest life insurance and pension groups, and a strong provider of investment products.

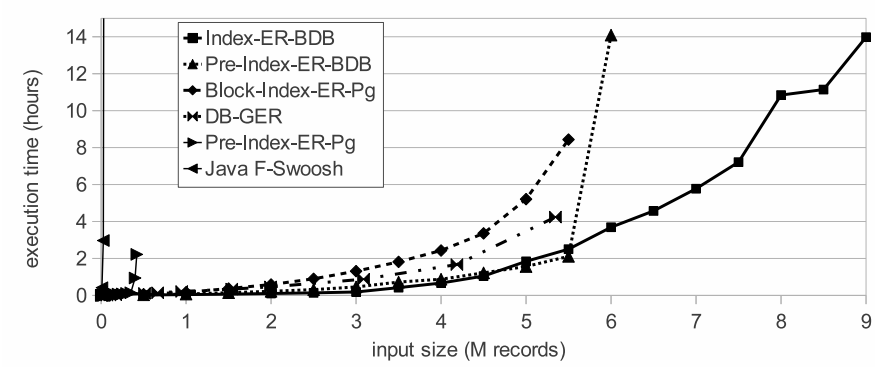

Fig. 1. Execution times against input size

We used two previously known algorithms for comparison. DB-GER is an SQL-based ER algorithm for relational databases (see [10]). Java F-Swoosh is a basic Java in-memory F-Swoosh implementation of [1]. Both DB-GER and Java F-Swoosh experiments were performed with Oracle 10g database. Index-ER-BDB and Pre-Index-ER-BDB used Berkeley DB B-trees for feature indices and also to store records. Pre-Index-ER-Pg used standard PostgreSQL indices. Block-Index-ER-Pg used feature-based blocking and in-memory algorithms for feature indices. Feature block construction and the update operation with block results is done by PostgreSQL.

Figure 1 plots execution times against input size. Java F-Swoosh showed poor performance without proper indexing. Performance of Block-Index-ER-Pg was still inferior: PostgreSQL through JDBC handles batch updates slow on the whole entity set. In-memory processing and Index-ER variation on blocks costs negligible time. So Block-Index-ER stays promising, supposing that a better entity store can be found with faster updates. Interestingly Pre-Index-ER-Pg also performed poor: feature index lookups were much slower than by BDB.

Both Pre-Index-ER-BDB and Index-ER-BDB outperform previous solutions. Figure 2 shows how the number of records processed changes for a smaller record set. Index-ER-BDB slows down as the size of the feature indices increase. With Pre-Index-ER-BDB processing becomes faster as more and more merges were performed.

The reason why Index-ER-BDB outperforms Pre-Index-ER-BDB variation is depicted in Figure 3. At around 3.5 M records, Index-ER-BDB slows down since at this point, BDB runs out of cache memory. However, Pre-Index-ER-BDB builds the whole index at the beginning, runs out of cache memory right away, and runs slow all along.

The additional space cost to store feature indices depends both on the number of features and on the distribution of feature values. In our experiments 5 features were used, using diverse attributes with more or less values. Index size varied

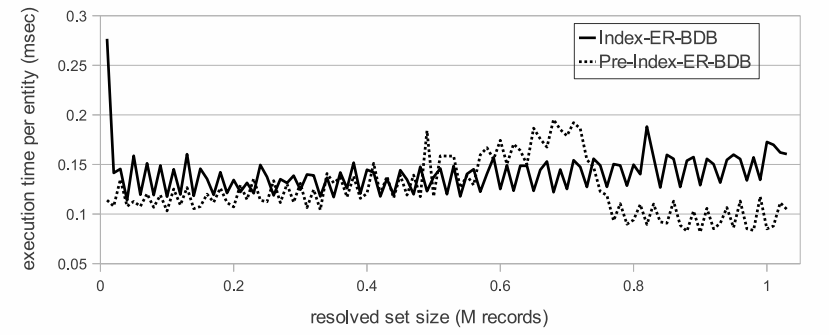

Fig. 2. Processing speed change for 1.1 M records

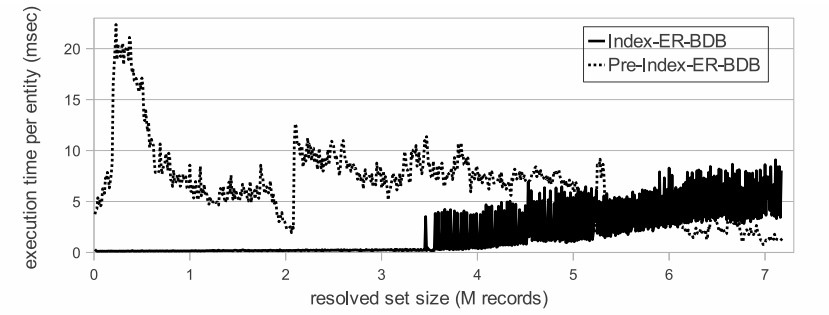

Fig. 3. Processing speed change for 7.2 M records

from 2 to 3 times the original database size. For PostgreSQL this realized as around 6.9 GB index size compared to the 2.4 GB full database size.

6 Conclusion and Future Work

We described algorithms to solve the ER problem based on feature indices. We showed that feature indexing is costly but beneficial, while the constructs used for indices, features and blocking enable solving a wide range of practical problems. We believe that appropriate selection of indexing tools and algorithms can further improve performance. Our algorithms may include arbitrary indexing and search solutions, both for exact and for similarity-based matching. The class of suitable tools depends on the data set, on the features, on the match logic and on the architectural environment as well.

In future work our algorithms and index alternatives should be tested in other settings, e.g. on conceptually different data sets, or with similarity-based feature matching. Feature-based blocking also enables distributing our algorithms in a parallel environment, a next step to overcome scalability problems. The selection of appropriate data structures for merging partial results needs however

further investigation. Human supervision is unavoidable: probabilistic models with uncertain statements should also be worked out.

Acknowledgments

To András Vereczki and Zoltán Hans as domain experts on the AEGON Hungary side for discussion on the problem formulation and clarification of the user requirements.

References

1. O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.
2. I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1):5, 2007.
3. P. Christen, R. Gayler, and D. Hawking. Similarity-aware indexing for real-time entity resolution. In *Proc. of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1565–1568, New York, NY, USA, 2009. ACM.
4. A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–16, 2007.
5. I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.
6. S. Guo, X. L. Dong, D. Srivastava, and R. Zajac. Record linkage with uniqueness constraints and erroneous values. *Proc. VLDB Endow.*, 3:417–428, September 2010.
7. H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69:197–210, February 2010.
8. H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.*, 3:484–493, September 2010.
9. D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results. *Proc. VLDB Endow.*, 3:208–219, September 2010.
10. C. Sidló. Generic entity resolution in relational databases. In J. Grundspenkis, T. Morzy, and G. Vossen, editors, *Advances in Databases and Information Systems*, volume 5739 of *LNCS*, pages 59–73. Springer, 2009.
11. S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *Proc. VLDB Endow.*, 3:1326–1337, September 2010.
12. S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *Proc. of the 35th SIGMOD int. conf. on Management of data*, pages 219–232, New York, NY, USA, 2009. ACM.
13. M. Yakout, A. K. Elmagarmid, H. Elmeleegy, M. Ouzzani, and A. Qi. Behavior based record linkage. *Proc. VLDB Endow.*, 3:439–448, September 2010.