Developments in Generic Entity Resolution

Steven Euijong Whang
Stanford University
swhang@cs.stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Abstract

Entity resolution (ER) is the problem of identifying which records in a database refer to the same entity. Although ER is a well-known problem, the rapid increase of data has made ER a challenging problem in many application areas ranging from resolving shopping items to counter-terrorism. The SERF project at Stanford focuses on providing scalable and accurate ER techniques that can be used across applications. We introduce generic ER and explain the recent advances made in our project.

1 Introduction

Information integration is one of the most important and challenging computer science problems: Information from diverse sources must be combined, so that users can access and manipulate the information in a unified way. One of the central problems in information integration is that of *Entity Resolution (ER)* (sometimes referred to as deduplication). ER is the process of identifying and merging records judged to represent the same real-world entity. ER is a well-known problem that arises in many applications. For example, mailing lists may contain multiple entries representing the same physical address, but each record may be slightly different, e.g., containing different spellings or missing some information. As a second example, a comparative shopping website may aggregate product catalogs from multiple merchants.

Identifying records that *match* (e.g., records that represent the same product) is challenging because there are no unique identifiers across sources (e.g., merchant catalogs that contain the information of products). A given record may appear in different ways in each source, and there is a fair amount of guesswork in determining which records match. Deciding if records match is often computationally expensive, e.g., may involve finding maximal common subsequences in two strings. How to *merge* records, i.e., combine records that match is often application dependent. For example, say different prices appear in two records to be merged. In some cases we may wish to keep both of them, while in others we may want to pick just one as the "consolidated" price. Such ER techniques find direct use in disciplines like computer science, biology, medicine, and even counter-terrorism.

The SERF project [7] at Stanford focuses on providing a general framework for ER that can span multiple applications. We do not study the internal details of the functions that determine if one or more records represent the same real-world entity. Rather, we view such functions as "black boxes", making our solutions useful across applications. In the initial phases of our project [1], we studied black-boxes that compared pairs of records

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

only (pair-wise matching functions). If two records matched, they were deemed to represent the same real-world entity and were merged into a composite record (via a merge function). As we will see (Section 2), more recently we have considered more general black-boxes that simply take as input a set of records (or an initial partition of records) and produce a partition of the records, where records in the same output partition are deemed to represent the same real-world entity. We call such a black-box the "core ER algorithm." The core algorithm could use pairwise matching as its strategy, but could use any other scheme.

After defining our model, we introduce recent developments for generic ER in the SERF project. We first consider scaling ER on a single data type where all the records have the same schema (e.g., all records refer to people). Many blocking techniques [3] have been proposed for scaling ER where the entire data is divided into (possibly overlapping) blocks, and the core ER algorithm is run on one block at a time. However, the downside is that one may miss matching records across blocks. One solution is to consider the interaction between blocks where resolving one block may help resolve others. In Section 3, we illustrate how this approach can maintain accuracy while providing scalability.

We also study scaling ER when resolving multiple types of data. For example, some records may refer to papers while others refer to authors. Since papers and authors are related to each other, resolving papers may help resolving authors and vice versa. While many ER solutions in the literature focus on resolving one type of records, few of them study the interactions between resolving different types of records. In Section 4 we illustrate joint ER and discuss scalable solutions.

In practice, a core ER function continuously "evolves" as the application developer has a better understanding on the application and data. For instance, when comparing people records, the application developer may at first think that comparing the names and zip codes of people is a good choice. After resolving the records, however, the developer may realize that comparing the names and phone numbers of people produces better ER results. (And this decision may further change as well.) When the ER algorithm logic changes, the naïve solution for ER is to simply re-run the ER algorithm from scratch, which may be prohibitively expensive when resolving large datasets. In Section 5, we provide incremental solutions for quickly updating ER results when the ER algorithm logic evolves.

Finally, we study a configurable and scalable quality measure for ER results. Many ER measures used in the literature "conflict" in a sense that, depending on the measure, different ER algorithms appear to perform better than others. However, many existing ER works tend to arbitrarily choose a measure "out of the hat" and evaluate their algorithms. In Section 6, we illustrate how conflicts may occur and introduce a scalable and configurable measure that can capture different qualities of ER results.

2 Core ER Model

A core ER algorithm takes as input a set of records R and groups together records that represent the same real world entity. We represent the output of the ER process as a partition of the input. We do not assume any particular form or data model for representing records.

To illustrate a possible core algorithm, consider the records of Table 1. Note that the structure of these records and the way we determine if records refer to the same real-world entity (via a pair-wise comparison) are part of the example. For this example, the core ER algorithm works as follows: A function compares the name, phone and email values of pairs of records. If the names are very similar (above some threshold), the records are said to match. The records also match if the phone and email are identical.

For our sample data, the ER algorithm determines that r_1 and r_2 match, but r_3 does not match either r_1 or r_2. For instance, the function finds that "John Doe" and "J. Doe" are similar, but finds "John D." not similar to anything (e.g., because John is a frequent first name). Thus, r_1 and r_2 are clustered together to form the partition $\{\{r_1, r_2\}, \{r_3\}\}$ where the curly brackets denote the clusters in the output partition.

Since records in an output cluster represent some real world entity, the cluster can be considered a "compos-

	Name	Phone	E-mail
r_1	John Doe	235-2635	jdoe@yahoo
r_2	J. Doe	234-4358	
r_3	John D.	234-4358	jdoe@yahoo

Table 1: A set of records representing persons

ite" new record. In some cases we may apply a merge function to actually generate the composite record. For example, suppose that the ER algorithm combines the names into a "normalized" representative, and performs a set-union on the emails and phone numbers. Then the records r_1 and r_2 will merge into the new records $\langle r_1, r_2 \rangle$ as shown below. (Here we denote a merged record with angle brackets.)

$\langle r_1, r_2 \rangle$	John Doe	234-4358, 235-2635	jdoe@yahoo

In this case, the ER result is the set of records $\{\langle r_1, r_2 \rangle, r_3\}$. Even if the records are merged, the lineage of the new record can be used to identify the original records in the cluster. Notice that in this case, we can also iteratively match $\langle r_1, r_2 \rangle$ with r_3 and merge them together because they now have the same phone and email. Throughout this paper, we will either cluster or merge records depending on the given setting.

3 Iterative Blocking

We now show how a core ER algorithm can be extended to become more scalable while maintaining accuracy. Blocking [3] is a common technique used in the ER literature where the data is divided into (possibly overlapping) blocks and a core ER algorithm is used to compare records within the same block, assuming that records in different blocks are unlikely to match. For example, we might partition a set of people records according to the zip codes in address fields. We then only need to compare the records with the same zip code. Since a single blocking criterion may miss matches (e.g., a person may have moved to places with different zip codes), several blocking criteria (i.e., dividing the data in several ways) are typically used to ensure that all the likely matching records are compared, improving the accuracy of the result.

Although many previous works focus on finding the best blocking criteria, most of them assume that all the blocks are processed separately one at a time. In many cases, however, it is useful to exploit an ER result of a previously processed block. First, when two records are grouped together (by the core algorithm) in one block, their composite may then be grouped with records in other blocks. Second, an ER result of a block can be used to reduce the processing time of another block. That is, the same record comparisons made by the core algorithm in one block may be repeated in another block, and hence can be avoided. To address these two points, we propose an iterative blocking framework [12] where the ER result of a block is immediately reflected to other blocks. Unlike previous blocking techniques, there is an additional stage where newly created partitions (i.e., groups of matching records) of a block are distributed to other blocks. Since the propagation of ER results can generate new record matches in other blocks, the entire operation becomes iterative in the sense that we are processing blocks (possibly the same block multiple times) until we arrive at a state where the resulting partition of records is stable. Our work is thus focused on effectively processing the blocks given a blocking criteria.

As an example, consider the five people records shown in Table 2, and a core algorithm that works by comparing pairs of records. We would like to merge (i.e., place in the same output partition) records that actually refer to the same person. Suppose that records r_1 and r_2 match with each other because their names are the same, but do not match with r_3 because the strings differ too much. However, once r_1 and r_2 are merged into a new record $\langle r_1, r_2 \rangle$, the combination of the address and email information of r_1 and r_2 may lead us to discover a new match with r_3, therefore yielding an initially unforeseen merge $\langle r_1, r_2, r_3 \rangle$. Notice that, in order to find this new merge, we need to compare the merged result of r_1 and r_2 with all the other records again.

Record	**Name**	**Address(zip)**	**Email**
r_1	John Doe	02139	jdoe@yahoo
r_2	John Doe	94305	
r_3	J. Foe	94305	jdoe@yahoo
r_4	Bobbie Brown	12345	bob@google
r_5	Bobbie Brown	12345	bob@google

Table 2: Customer records

In reality, our dataset can be very large, and it may not be feasible to compare all pairs of the dataset. Hence, we divide the customer records in Table 2 into blocks. We start by dividing the records by their zip codes. As a result, we only need to compare customers that are in the same geographical region. In Table 3, the first blocking criterion SC_1 uses zip codes to divide the records. Records r_2 and r_3 have the same zip code and are assigned to the block 2 (denoted as $b_{1,2}$) and records r_4 and r_5 are assigned to $b_{1,3}$ while r_1 is assigned to $b_{1,1}$. Since we may miss matches for people who have moved to several places with different zip codes, say we also divide the customer records according to the first characters of their last names. Hence, even if two records referring to the same person have different zip codes, we will have a better chance of comparing them because their last names might be similar. In Table 3, the matching records r_1 and r_2 can be compared because, although they have different zip codes, they have the same last name. After processing all the blocks, the final result of blocking is $\{\langle r_1, r_2\rangle, r_3, \langle r_4, r_5\rangle\}$.

Criterion	**Partitions by**	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	zip code	r_1	r_2, r_3	r_4, r_5
SC_2	1st char of last name	r_1, r_2	r_3	r_4, r_5

Table 3: Multiple blocking

Although the blocking of Table 3 reduces the number of records to compare, it misses the iterative match between $\langle r_1, r_2\rangle$ and r_3. Iterative blocking can find this match by distributing the newly created $\langle r_1, r_2\rangle$ (found in block $b_{2,1}$) to the other blocks. Assuming that $\langle r_1, r_2\rangle$ contains the zip codes of both r_1 and r_2 (i.e., 02139 and 94305), $\langle r_1, r_2\rangle$ is then assigned to both blocks of SC_1. In $b_{1,2}$, $\langle r_1, r_2\rangle$ can then be compared with r_3, generating the record $\langle r_1, r_2, r_3\rangle$. Eventually, the final iterative blocking solution becomes $\{\langle r_1, r_2, r_3\rangle, \langle r_4, r_5\rangle\}$. Thus, the iterative blocking framework helps find more record matches compared to simple blocking.

Iterative blocking also provides fast convergence. For example, once the records r_4 and r_5 are merged in $b_{1,3}$, they do not have to be compared in $b_{2,3}$. While the blocks in Table 3 are too small to show any improvements in runtime, iterative blocking can actually run faster than blocking for large datasets when the runtime savings exceed the overhead of iterative blocking. Intuitively, the more work we do for each block, the runtime savings for other blocks become significant.

In reference [12], we formalize the iterative blocking model and present two iterative blocking algorithms:

- *Lego:* An in-memory algorithm that efficiently processes blocks within memory.
- *Duplo:* A scalable disk-based algorithm that processes blocks from the disk.

We also experimentally evaluate Lego and Duplo and show that iterative blocking can improve over blocking both in accuracy and runtime.

4 Joint Resolution

The core ER algorithms can also be extended to jointly resolve multiple datasets of different entity types. Since relationships between the datasets exist, the result of resolving one dataset may benefit the resolution of another dataset. For example, the fact that two (apparently different) authors a_1 and a_2 have published the same papers could be strong evidence that the two authors are in fact the same. Also, by identifying the two authors to be the same, we can further deduce that two papers p_1 and p_2 written by a_1 and a_2, respectively, are more likely to be the same paper as well. This reasoning can easily be extended to more than two datasets. For example, resolving p_1 and p_2 can now help us resolve the two corresponding venues v_1 and v_2. Compared to resolving each dataset separately, joint ER can achieve better accuracy by exploiting the relationships between datasets.

To illustrate joint ER, consider two datasets P and V (see Table 4) that contain paper records and venue records, respectively. (Note that Table 4 is a simple example used for illustration. In practice, the datasets can be much larger and more complex.) The P dataset has the attributes Title and Venue while V has the attributes Name and Papers. For example, P contains record p_1 that has the title "The Theory of Joins in Relational Databases" presented at the venue v_1. The Papers field of a V record contains a set of paper records because one venue typically has more than one paper presented.

Paper	**Title**	**Venue**
p_1	The Theory of Joins in Relational ...	v_1
p_2	Efficient Optimization of a Class of ...	v_1
p_3	The Theory of Joins in Relational ...	v_2
p_4	Optimizing Joins in a Map-Reduce ...	v_3

Venue	**Name**	**Papers**
v_1	ACM TODS	$\{p_1, p_2\}$
v_2	ACM Trans. Database Syst.	$\{p_3\}$
v_3	EDBT	$\{p_4\}$

Table 4: Papers and Venues

Suppose that two papers are considered the same and are clustered if their titles and venues are similar while two venues are clustered if their names and papers are similar. Say that we resolve the paper records first. Since p_1 and p_3 have the exact same title, p_1 and p_3 are considered the same paper. We then resolve the venue records. Since the names of v_1 and v_2 are significantly different, they cannot match based on name similarity alone. Luckily, using the information that p_1 and p_3 are the same paper, we can infer that v_1 and v_2 are most likely the same venue. (In fact "ACM TODS" and "ACM Trans. Database Syst." stand for the same journal.) We can then re-resolve the papers in case there are newly matching records. This time, however, none of the papers match because of their different titles. Hence, we have arrived at a joint ER result where P was resolved into the partition $\{\{p_1, p_3\}, \{p_2\}, \{p_4\}\}$ while V was resolved into $\{\{v_1, v_2\}, \{v_3\}\}$. Notice that we have followed the sequence of resolving papers, venues, then papers again.

Given enough resources, we can improve the runtime performance by exploiting parallelism and minimizing unnecessary record comparisons. For example, if we have two processors, then we can resolve the papers and venues concurrently. As a result, p_1 and p_2 match with each other. After the papers and venues are resolved, we resolve the venues again, but only perform the incremental work. In our example, since p_1 and p_2 matched in the previous step, and p_1 was published in the venue v_1 while p_2 was published in the venue v_2, we only need to check if v_1 and v_2 are the same venue and can skip any comparison involving v_3. Notice that the papers do not have to be resolved at the same time because none of the venues merged in the previous step. However, after v_1 and v_2 are identified as the same venue, we resolve the papers once more. Again, we only perform the incremental work necessary where we resolve the three records p_1, p_2, and p_3 (because v_1 and v_2 matched in the previous step), but not p_4. In total, we have concurrently resolved the papers and venues, then incrementally resolved the venues, and then incrementally resolved the papers. If the incremental work is much smaller than resolving a dataset from the beginning, the total runtime may improve.

Among the existing works on joint ER [2], few have focused on scalability, which is crucial in resolving large data (e.g., hundreds of millions of people records crawled from the Web). The solutions that do address scalability propose custom joint ER algorithms for efficiently resolving records. However, given that there exists ER algorithms that are optimized for specific types of records (e.g., there could be an ER algorithm that specializes in resolving authors only and another ER algorithm that is good at resolving venues), replacing all the ER algorithms with a single joint ER algorithm that customizes to all types of records may be challenging for the application developer.

Instead, we propose a flexible, modular resolution framework where existing ER algorithms developed for a given record type can be plugged in and used in concert with other ER algorithms. While many previous joint ER works assume that all the datasets are resolved at the same time in memory using one processor, our framework allows efficient resource management by resolving a few datasets at a time in memory using multiple processors as illustrated above. A conceptually similar ER technique is blocking where the entire data of one type is resolved in small subsets or blocks. Similarly, our framework resolves multiples types of data and divides the resolution based on the data type. Our approach may especially be useful when there are many large datasets that cannot be resolved altogether in memory. Thus one of the key challenges is determining a good sequence for resolution. For instance, should we resolve all venues first, and then all papers and then all authors? Or should we consider a different order? Or should we resolve some venues first, then some related papers and authors, and then return to resolve more venues? And we may also have to resolve a type of record multiple times, since subsequent resolutions may impact the work we did initially. These issues (and others) are explored in a technical report [10] that describes our ongoing work in this area.

5 Incremental Resolution

We now show how certain properties of the core ER algorithm enable incremental solutions when the core ER algorithm itself changes frequently. In practice, an entity resolution (ER) result is not produced once, but is constantly improved based on better understandings of the data, the schema, and the logic that examines and compares records. In particular, here we focus on changes to the logic that compares two records in the core ER algorithm. We call this logic the *rule*, and it can be a Boolean function that determines if two records represent the same entity, or a distance function that quantifies how different (or similar) the records are. Initially the core ER algorithm starts with a set of records S, then produce a first ER result E_1 based on S and a rule B_1. Some time later rule B_1 is improved yielding rule B_2, so we need to compute a new ER result E_2 based on S and B_2. The process continues with new rules B_3, B_4 and so on.

A naïve approach would compute each new ER result from scratch, starting from S, a potentially very expensive proposition. Instead, we explore an incremental approach [9], where for example we compute E_2 based on E_1. Of course for this approach to work, we need to understand how the new rule B_2 relates to the old one B_1, so we can understand what changes incrementally in E_1 to obtain E_2.

For example, consider the set of people records S shown in Table 5. The first rule B_1 (see Table 5) says that two records match (represent the same real world entity) if predicate p_{name} evaluates to true. Predicates can in general be quite complex, but for this example assume that predicates simply perform an equality check using its attribute. The ER algorithm calls on B_1 to compare records and groups together records with name "John," producing the result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$.

Record	Name	Zip	Phone
r_1	*John*	54321	123-4567
r_2	*John*	54321	987-6543
r_3	*John*	11111	987-6543
r_4	*Bob*	*null*	121-1212

Comparison Rule	Definition
B_1	p_{name}
B_2	$p_{name} \wedge p_{zip}$
B_3	$p_{name} \wedge p_{phone}$

Table 5: Records and Comparison Rules

Next, say users are not satisfied with this result, so a data administrator decides to refine B_1 by adding a predicate that checks zip codes. Thus, the new rule is B_2 shown in Table 5. The naïve option is to run the same ER algorithm with rule B_2 on set S to obtain the partition $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. (Only records r_1 and r_2 have the same name and same zip code.) This process repeats much unnecessary work: For instance, we would need to compare r_1 with r_4 to see if they match on name and zip code, but we already know from the first run that they do not match on name (B_1), so they cannot match under B_2.

Because the new rule B_2 is "stricter" than B_1 (i.e. all records that match according to B_2 also match according to B_1), we can actually start the second ER from the first result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$. That is, we only need to check each cluster separately and see if it needs to split. In our example, we find that r_3 does not match the other records in its cluster, so we arrive at $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. This approach only works if the ER algorithm satisfies certain properties and B_2 is stricter than B_1. One of the properties that the ER algorithm must satisfy is being able to resolve clusters "independent" of each other. In our example above, splitting the cluster $\{r_1, r_2, r_3\}$ must have no affect on the cluster $\{r_4\}$. If B_2 is not stricter and the ER algorithm satisfies different properties, there are other incremental techniques we can apply.

A complementary technique is to "materialize" auxiliary results during one ER run, in order to improve the performance of future ER runs. To illustrate, say that when we process $B_2 = p_{name} \wedge p_{zip}$, we concurrently produce the results for each predicate individually. That is, we compute three separate partitions, one for the full B_2, one for rule p_{name} and one for rule p_{zip}. The result for p_{name} is the same $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ seen earlier. For p_{zip} it is $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. The cost of computing the two extra materializations can be significantly lower than running the ER algorithm three times, as a lot of the work can be shared among the runs.

The materializations pay off when rule B_2 evolves into a related rule that is not quite stricter. For example, say that B_2 evolves into $B_3 = p_{name} \wedge p_{phone}$, where p_{phone} checks for matching phone numbers. In this case, B_3 is not stricter than B_2 so we cannot start from the B_2 result. However, we can start from the p_{name} result, since B_3 is stricter than p_{name}. Thus, we examine each cluster in $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, splitting the first cluster because r_2 has a different phone number. The final result is $\{\{r_1, r_3\}, \{r_2\}, \{r_4\}\}$. Clearly, materialization of partial results may or may not pay off, just like materialized views and indexes may or may not help.

In reference [9], we formalize rule evolution and identify desirable properties of ER algorithms that enable fast rule evolution and categorize existing ER algorithms based on the properties they satisfy. We then propose efficient rule evolution techniques that use one or more of the properties. We then experimentally evaluate the rule evolution algorithms for various ER algorithms and show that rule evolution can be significantly faster than the naïve approach with reasonable time and space overheads.

6 Quality Measure

Properly evaluating ER results is important for comparing the performances of various ER algorithms. Usually when we compare entity resolution algorithms, we run them on a data set and compare the results to a "gold standard." The gold standard is an ER result that we assume to be correct. In many cases, the gold standard is generated by a group of human experts. On large data sets where the task is too large to be handled by a human, it is not uncommon to run an exhaustive algorithm to generate a result, and treat that result as the gold standard. Then we can compare the results of other approximate or heuristic-based algorithms to this standard in the same manner we would compare them to a human-generated gold standard.

A key component of this type of evaluation is a method of assigning a number to express how close a given ER result is to the gold standard. Many ER measures have been proposed for comparing the results of ER algorithms, but there is currently no agreed standard measure for evaluating ER results. Most works tend to use one ER measure over another without a clear explanation of why that ER measure is most appropriate. The pitfall of using an arbitrary measure is that different measures may disagree on which ER results are the best.

For example, consider an entity resolution problem with an input set of records $I = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7,$

r_8, r_9, $r_{10}\}$. Three possible ER results are shown in Table 6, along with the gold standard.

Set	ER Result
Gold Standard	$\{\{r_1, r_2\}, \{r_3, r_4\}, \{r_5, r_6, r_7, r_8, r_9, r_{10}\}\}$
R_1	$\{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_5, r_6, r_7, r_8, r_9, r_{10}\}\}$
R_2	$\{\{r_1, r_2\}, \{r_3, r_4\}, \{r_5, r_6, r_7\}, \{r_8, r_9, r_{10}\}\}$
R_3	$\{\{r_1, r_2, r_3, r_4\}, \{r_5, r_6, r_7, r_8, r_9, r_{10}\}\}$

Table 6: ER results and Gold Standard

Suppose we are evaluating two ER results R_1 and R_2, against the gold standard G. Using an ER measure that evaluates a result based on the number of record pairs that match, R_1 could be a better solution because it found 15 correct pairs (i.e., all record pairs in $\{r_5, r_6, r_7, r_8, r_9, r_{10}\}$) while R_2 only found 8 correct pairs. On the other hand, if we use a measure that evaluates results based on correctly resolved entities in the gold standard, R_2 could be considered better than R_1 because R_2 contains two correctly resolved entities $\{r_1, r_2\}$ and $\{r_3, r_4\}$ while R_1 only has one correct entity $\{r_5, r_6, r_7, r_8, r_9, r_{10}\}$. As another example, suppose that we compare R_2 and R_3. One measure could be more focused on high precision and prefer R_2 over R_3 because R_2 has only found correctly matching records while R_3 has found some non-matching records (e.g., r_1 and r_3 do not match). On the other hand, another measure might consider recall to be more important and prefer R_3 over R_2 because R_2 has not found all the matching record pairs (unlike R_3).

Surprisingly, such conflicts between ER measures can occur frequently. It is tempting to suggest that when conflicts arise, one of the measures involved must be faulty in some way. However, since different applications may have different criteria that define the "goodness" of a result, we cannot simply claim one measure to be better than another.

In reference [6], we provide a survey of ER measures that have been used to date, experimentally demonstrates the frequency of conflicts between these measures, and provides an analysis of how the measures differ.

We also propose a configurable measure inspired by the edit distance of strings. Rather than the insertions, deletions and swaps of characters used in edit distance, our measure is based upon the elementary operations of merging and splitting clusters. We therefore call this measure "merge distance." A basic merge distance simply counts the number of splits and merges. For example, R_1 has a distance of 2 from G because it takes two cluster merges to convert R_1 to G. On the other hand, R_3 only has a distance of 1 because one cluster split is required to convert R_3 to G. But as we have mentioned, no single ER measure is better than all the others. We thus generalize merge distance by letting the costs of merge and split operations be determined by "cost functions," arriving at an intuitive, configurable measure that can support the needs of a wide variety of applications. For example, the cost of splitting a cluster could be defined to be twice as large as the cost of merging two clusters. Or the costs of merging and splitting clusters could be proportional to the sizes of the input clusters. While differently configured merge distance measures may still conflict, we now have a better understanding of what quality each configured measure is evaluating. Surprisingly, at least two state-of-the-art measures are closely related to generalized merge distance: the Variation of Information (VI) [5] clustering measure is a special case of generalized merge distance while the pairwise F_1 [4] measure can be directly computed using generalized merge distance. We also propose an efficient linear-time algorithm that computes generalized merge distance for a large class of cost functions that satisfy reasonable properties.

7 Conclusion

In this paper, we have introduced the recent advances on generic ER in the SERF project. We have first explained our basic ER model where a core ER algorithm resolves a set of records. We have then shown two methods – iterative blocking and joint ER – that can scale ER while maintaining accuracy using the core ER algorithms. We have also shown incremental ER solutions for evolving ER rules. Finally, we have proposed a configurable and scalable quality measure for evaluating ER results. Although not covered in this paper, we have also studied the problems of correcting inconsistencies [8] and producing good partial ER results within a limited runtime [11].

References

[1] O. Benjelloun, H. Garcia-Molina, H. Kawai, T. E. Larson, D. Menestrina, Q. Su, S. Thavisomboon, and J. Widom. Generic entity resolution in the serf project. *IEEE Data Eng. Bull.*, 29(2):13–20, 2006.

[2] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, pages 85–96, 2005.

[3] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Conference*, pages 127–138, 1995.

[4] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[5] M. Meila. Comparing clusterings by the variation of information. In *COLT*, pages 173–187, 2003.

[6] D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results. *PVLDB*, 3(1):208–219, 2010.

[7] Stanford Entity Resolution Framework. http://infolab.stanford.edu/serf/.

[8] S. E. Whang, O. Benjelloun, and H. Garcia-Molina. Generic entity resolution with negative rules. *VLDB J.*, 18(6):1261–1277, 2009.

[9] S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *PVLDB*, 3(1):1326–1337, 2010.

[10] S. E. Whang and H. Garcia-Molina. Joint entity resolution. Technical report, Stanford University, available at http://ilpubs.stanford.edu:8090/1002/.

[11] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. Technical report, Stanford University, available at http://ilpubs.stanford.edu:8090/979/.

[12] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, pages 219–232, 2009.