Infrastructures and Bounds for Distributed Entity Resolution*

Csaba István Sidló András Garzó András Molnár András A. Benczúr
Institute for Computer Science and Control, Hungarian Academy of Sciences
{sidlo, garzo, modras, benczur}@ilab.sztaki.hu

ABSTRACT

Entity resolution (ER), deduplication or record linkage is a computationally hard problem with distributed implementations typically relying on shared memory architectures. We show simple reductions to communication complexity and data streaming lower bounds to illustrate the difficulties with a distributed implementation: If the data records are split among servers, then basically all data must be transferred.

As a key result, we demonstrate that ER can be solved using algorithms with three different distributed computing paradigms:

- Distributed key-value stores;
- Map-Reduce;
- Bulk Synchronous Parallel.

We measure our algorithms in the real-world scenario of an insurance customer master data integration procedure. We show how the algorithms can be modified for non-Boolean fuzzy merge functions and similarity indexes.

1. INTRODUCTION

Entity Resolution (ER) is the process of identifying groups of records that refer to the same real-world entity [3, 4, 16, 31]. In this paper we demonstrate that distributed software infrastructures enable ER for huge data sets. We consider two branches of distributed infrastructures. The first one is designed for implementing algorithms and includes Map-Reduce and Bulk Synchronous Parallel. The second one supports database operations and includes NoSQL and distributed key-value store implementations. We select three out of these infrastructures to compare distributed ER implementations.

Entity resolution appears in a wide range of applications. Author name resolution in bibliographic databases form the first and ever since popular task in ER research. Industrial applications are found in client or product databases. Clients for example may appear multiple times in multiple source systems, with a record for a contract, another for a purchase, or 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 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?

In this paper we demonstrate that smart algorithms over emerging distributed infrastructures enable ER for several orders of magnitude larger data sets than in earlier research. Existing ER algorithms are often not scalable enough, resolving a larger record set may require days on conventional architectures. Earlier results give experiments for a few 100,000 records (see Section 1.1). Similar to the size of our data set, in [41] 2 million records are processed, but processing took several hours and distribution to more servers was not considered. For our 20 million record real world customer data set, our fastest algorithm completes its task in less than an hour over 15 low-end servers. Furthermore, we experimented by replicating our data to a 300 times larger data set than in [41]. Experiments with data closest to our size is reported in [39], 10 million records. They rely on a simple sorting based algorithm [21] that also forms the base of our methods. They produce approximate results and deploy commercial database management systems, that however impose limitations to their procedure, for example the usability of similarity indexes is unclear.

A no shared memory distributed ER implementation that we are aware of, D-swoosh [2] does not rely on shared memory but uses no distributed software infrastructure either. They communicate by sockets between pairs of servers; also the number of records is a mere few 100,000 in their experiment. While programming sockets gives the most flexibility and should yield the fastest distributed implementation, is sacrifices the services of a distributed infrastructure such as fault tolerance, dynamic load allocation and process monitoring. Other parallel algorithms are presented in [26], tested on a few thousands of records. More recently [27] in-

*This work was supported by the EU FP7 Projects LAWA (Large-Scale Longitudinal Web Analytics) and SCIIMS (Strategic Crime and Immigration Management System).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. This article was presented at:
9th International Workshop on Quality in Databases (QDB) 2011
Copyright 2011.

troduces parallel matching and a distributed infrastructure, experimenting with a data set of 114 thousand records, and using similarity-based matchers.

Our main contributions are as follows.

- We show that, simply speaking, any distributed ER algorithm must exchange all of its data across different servers, hence significantly hardening the task compared to pre-existing shared memory ER solutions (Section 3). In particular this means that blocking may result in significant gains for multi-threaded ER but much less for ER over distributed data.
- We demonstrate that ER can be implemented under three major distributed programming paradigms: distributed key-value stores, Map-Reduce, and Bulk Synchronous Parallel.
- We measure our algorithms on very large scale real world data. Our client database consists of 20 million records that we blow up to 600 million for scalability tests.

The goal with our implementations is to evaluate the applicability of the frameworks and show their strength and limitation for complex ER tasks. We emphasize that our codes contain little optimization in their interaction with the corresponding software framework; also the frameworks, in particular the Bulk Synchronous Parallel, is yet in an incubatory phase and we may observe significant speedup by using an improved release.

The rest of this paper is organized as follows. In Section 2 we define the ER problem and introduce notation. We give the basic algorithms and show how the simple implementations can be enhanced by more complex matching and similarity functions. In Section 3 we describe our lower bounds on the amount of communication across servers in a distributed ER algorithm. The three main algorithms are described in separate sections. The algorithms in Section 4 rely on distributed key-value stores; in Section 5 on Map-Reduce; and in Section 6 on Bulk Synchronous Parallel. Finally the experimental results based on operational insurance client databases is given in Section 7.

1.1 Related results

In this section we only discuss general results on entity resolution. More comparison with existing results is given thematically on similarity relations in Section 2.1 and on partitioning, blocking in Section 3. Background information on the three distributed programming paradigms are found in the corresponding sections on the algorithms, Sections 4–6.

One of the first descriptions of the record linkage problem appears in Fellegi and Sunter [17] 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. In [16] 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 [3], where resolution means generating the closure of the original entity set according to these functions. In their result, indexes for simple combinations of attributes called features are also used. We give

name	e-mail	ID
Mary Smith	m.smith@mail-1.com	50071
Mary Doe	mary@mail-2.com	50071
M. Doe	mary@mail-2.com	79216
M. Smith	m.smith@mail-1.com	34302

Figure 1: Example of records belonging to the same entity. The first two records have identical ID, representing a customer with name Mary Smith and maiden name Mary Doe. The remaining two records are variants of the name and maiden name, respectively.

generic ER algorithms for relational databases in [35]. Entity resolution as a hypergraph clustering problem can be found in [4], under the name of relational clustering.

Recently, several new ER results were published. A new approach can be found in [43]: 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, [31] deals with possible quality metrics. [41] enhances core ER algorithms by combining the results of different blocking strategies. Our formal model of entity resolution is similar to the model used in [41] when forming dominated record partitions. [20] exploits the role of constraints when finding duplicates. [40] deals with the effect of match/merge rule evolution, and gives methods to preserve results when rules change. [10] builds special inverted indexes to speed up ER with blocking. A survey of indexing techniques available for deduplication is provided in [8], including blocking, sorted neighborhood, Q-grams and canopies.

Entity resolution frameworks are developed, like SERF, MTB, DDUpe and MARLIN (see [28]). A practical comparison of ER approaches can be found in [29] using the FEVER framework. Besides, the Febrl framework provides parallelization [9].

2. PROBLEM FORMULATION

Entities of the real world are typically hidden and only indirect observations are recorded in a database. This intuition is formalized as follows. Let a set of **records** be $R = \{r_1, r_2, ...r_m\}$, where each r_j is described by its k **attribute values** $a_{r_j 1}$, ..., $a_{r_j k}$ such as ID, name, address, etc. Some attribute values may be missing, e.g. we may not know the e-mail address of customer j that we denote by $a_{r_j \ell} = \emptyset$.

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: $r \in e_i \wedge r \notin e_j \Rightarrow i = j$.

To give an example, consider records and attributes in a client database stating that "the name of the client is Mary Doe" or "the email address is `mary@mail-2.com`". An example using person records is depicted in Fig. 1 with Mary Doe already a customer before her marriage when her name changes to Mary Smith. Both of her names will appear in one record as name and maiden name but duplicates for

both versions may appear in the data.

An entity can have more than one attribute values a_1, $a_2, \ldots a_i$ of the same type, for example multiple names may exist for a real-world client. We represent such an entity e by a set of records $e = \{r_1, r_2, \ldots\ r_i\}$ such that there are records for each value, $a_{r_1\ell} = a_1$, $a_{r_2\ell} = a_2, \ldots a_{r_i\ell} = a_i$.

For an entity match relation $e_1 \sim e_2$, 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 record sets where no more separate but matching entities can be found.

In our ER algorithms, entities e_1 and e_2 are merged, i.e. $e_1 \sim e_2$ if a matching attribute value is found in their records. Match is a relation between attribute values $a_1 \sim a_2$ depending on the application. For example, we may require identity but the definition may involve distance functions over the attribute space as well. Formally, e_1 and e_2 are matched if there exists $r_1 \in e_1$ and $r_2 \in e_2$ and ℓ with $a_{r_1\ell} \sim a_{r_2\ell}$.

We also require indexability: we require that the attribute values matching to a given a can be enumerated and a represented by a single value or a set of values. In this latter case, we may handle multiple representative values in the same way as we normally handle multiple attribute values. Examples of indexing methods working this way include n-grams or fingerprints of shingles [5], or multidimensional search structures such as R or KD-trees. More details are given in the next subsection.

2.1 Attribute value similarity and fuzzy matching

Our requirement of entity match based on an indexable attribute value match relation may seem overly restrictive at the first glance. Next we list alternatives from related results and show how they in fact reduce to our simple settings. For the rest of the paper, we will then only describe algorithms for attribute equality as a match function and, in each of Sections 4–6, we elaborate on how more complex match functions can be implemented and whether the given framework imposes additional constraints. An exhaustive list of such possibilities can for example be found in [7, Table 9]. We give extensions of our methods in three directions:

1. Fuzzy or machine learning based matching, with levels of confidence;
2. Complex similarity of the values of a single attribute;
3. Similarity based on a collection of attributes, also called *features* in [3].

As a first potential limitation (1), our match function, by strict definition, does not allow fuzzy matching by confidence levels or learning based approaches. In fact, in most of the previous work [32, 14, 21] an exact match function is assumed, similar to our starting point R-swoosh [3] where a Boolean pairwise match function is used, with no fuzzy merge and confidence values. Furthermore, their efficient F-swoosh variant, similar to our results, uses feature level comparisons.

Our algorithms may however reach beyond this limitation by shifting its goal towards efficient candidate generation. We may rephrase our output as an efficient way to distribute the work of sophisticated match functions and reshuffle potential matches to the same server. In this sense, splitting by post-processing as proposed among others in [41] fits very well into the distributed processing framework. Also, we may easily accommodate complex match criteria or even use our solutions to distribute machine learning.

Next, for (2), we argue that, since pairwise comparison of all attribute values is computationally infeasible, ER can efficiently solved only in the case when similarity indexes can be built for all attributes. This requirement gives no additional restriction if we use similarity search indexes. When using similarity-based or probabilistic features and match conditions, the indexes provide entities with similar attribute values beyond a given similarity threshold. Examples include finding duplicated web pages, using features based on geographic location and distance, features with name similarities etc. Document Q-gram and TF-IDF indexes, min-hash fingerprint [5] for example have multiple efficient standalone implementations, for distributed environments as well. Similarity-based indexes as described for example in [10] are also useful in this case, and are exhaustively investigated topics.

Finally for (3), features as subsets of attributes [3] can be handled, in the simplest setting, as attribute value sets where match operators are equality tests. For example, two clients match if they share a birth date, a birth name and a postal address. In this scenario we use multiple indexes that we directly implement in our algorithms. Or, when having a candidate generation phase, a birth name B-tree index [11] may be used, if birth name is always known and has a good selectivity. An other possibility is to use multidimensional indexes, e.g. R-trees to use multiple attributes as search key.

Our issues of indexability and (1–3) above are somewhat orthogonal to the so-called ICAR properties introduced in [3]. For a non-ICAR problem we will merge more than allowed and require the split operation as post-processing. Requiring ICAR may also help in somewhat limiting the size of the components that we create by merges. Finally we note that ICAR is introduced for the same purpose as our indexing requirements, by noting that G-Swoosh, the most general ER algorithm, is inefficient.

2.2 Basic algorithms and efficiency considerations

A naive solution to solve the ER problem would be to iterate through the input entity set and find matching pairs, as long as such pairs exist. Such algorithms, including G-swoosh [3, Algorithm 2] and even the improved R-swoosh, run in quadratic time and are hence inefficient for large data sets.

The starting point of our results is F-swoosh [3, Algorithm 4] that keeps a table of values for features, i.e. compositions of attribute values. In our algorithms we will either use feature indexes combining several attributes or rely solely on attributes and undo false merges in the post-processing phase.

Next we describe two critical issues of a parallel implementation that are normally considered as minor details in the literature since they are naturally resolved by sequential algorithms. Note that D-swoosh [2] and the older version P-swoosh [25] as well as the first results in the area [21] require shared memory and do not address these problems.

First, there is no general method for data distribution that

is significantly more efficient that the ER problem itself that keeps the majority of candidate duplicates at the same location. Stated in another way, there is no locality sensitive hashing for the minimum distance. We discuss negative results about problem partitioning in Section 3 by rephrasing data distribution as communication problem. For inputs of length n, the probabilistic (bounded error) communication complexity of set intersection is $\Theta(n)$ [23].

Second, the graph of entity mergers along matching values of various attributes may form an arbitrary graph. Parallel connected components [22] is not as simple and the general solution may require several passes. For efficiency it is crucial to base our algorithm on the assumption that the graphs are tiny.

The need for connected component identification is described first in the iterative blocking algorithm of [41], a set of algorithms that can be considered the sequential version of some of our methods.

3. A LOWER BOUND

In this section we briefly survey a set of results that indicate the impossibility an ER algorithm that distributes the data and exchanges records among the parts without communicating significantly less than the entire data set among the processes. Common to these results is that they use a reduction to the probabilistic (bounded error) communication complexity of set disjointness. By the result of [23], the question whether two sets intersect cannot be decided by communicating less than $\Theta(n)$ bits.

Blocking is a proven method to speed up ER algorithms but, as we will show, its power diminishes when data is distributed across multiple servers with no shared memory. Blocking divide records into smaller subsets based on expert heuristics including ZIP code, first letter of family names, etc., so that ER is performed on the smaller subsets more easily. In this way we can reduce complexity, but may miss potential matching pairs between different blocks. One solution is to use multiple blocking criteria, and then combine the results of the different ER results together. Another potential solution is iterative blocking [41], where merged entities are delegated to other affected blocks.

For efficient blocking, we may use external knowledge, e.g. only compare records with the same zip code, but this approach may fail if a person moves to another location. As another example that is characteristic to our real data set, maiden names have arbitrary connections to names (in some languages, even the first names may change!) for married women representing probably at least a quarter of most customer data sets. Clustering is also suggested [21]; however as the argument below shows that even checking a perfect partitioning with no entities merging along boundaries requires passing basically all the data across the data nodes.

Our first negative result is a direct consequence of the set disjointness communication complexity bound. Given that the records are partitioned to at least two data servers in some clever way, we may want to test if ER can be solved by local computations. In other words, for a given attribute we need to check whether there is a value that appears at both servers. By the communication bound, this task requires $\Theta(n)$ bits of communication for n records. Although a single bit for an attribute value may be considered a very efficient encoding, still the $\Theta(n)$ bound means that we basically have to communicate all data between the parts regardless of how smart algorithm is used for partitioning.

The above negative result can be reformulated by using another lower bound [1, Proposition 3.8] stating (among others) that the number of distinct elements cannot be exactly computed in less than $\Omega(n)$ memory bits. In addition, this problem cannot be solved by sampling either [6]. There are however low space relative error approximation results that may give hope of speeding up some ER heuristics both in [1] and earlier in [19]. Note however that for an exact ER algorithm, the negative results apply.

The impossibility of finding a "very smart" partitioning method can be expressed in another way. In the space of attributes as dimensions, two records are similar if they agree in at least one coordinate. Hence another way to state the negative results, there is no efficient locality sensitive hashing for entity linkage as similarity. Our argument is reminiscent to the non-existence for extremely nonconvex dissimilarity functions such as the minimum or a variant that simply counts the number of nonzeroes frequently termed Donoho's zero-'norm' [15].

Common to the three algorithms in Sections 4–6 is that they first communicate all data records among different parts of the data set and then perform additional steps to compute connected components. Communication is performed in iterations; in one iteration, the attribute values of a single attribute is communicated. Disregarding the work performed by the connected component algorithms that we consider low in practice, our algorithms are optimal by the above lower bounds.

4. ALGORITHM BASED ON KEY-VALUE STORES

High performance key-value storage systems are growing in both size and importance; they now are critical parts of major Internet services such as Amazon (Dynamo [13]), LinkedIn (Project Voldemort [33]), and Facebook (memcached [18]).

In our first implementation we choose Project Voldemort due to its ease of installation and APIs. We note that our solution works also based on various other indexing tools not covered in this paper, including Kyoto Cabinet, Scalien DB, or even MySQL.

The ER algorithm itself is an adaptation of a sequential connected component algorithm by relying on the distributed key-value store. The implementations manage entities by their entity ID $\mathrm{ID}(e)$. We propose two variants:

1. The list of entities is read sequentially from disk. There are k indexes, one for each attribute, and another index that points to the parent of the given entity. This algorithm implements a union-find data structure [11, Section 21] over this last key-value store.
2. The entities reside in a Voldemort store with $\mathrm{ID}(e)$ as key. There are k indexes, one for each attribute, as in the previous solution. Entities are merged by performing a breadth-first search [11, Section 22] and the entity store is immediately updated during merge.

The union-find based solution (Algorithm 1) simply iterates through all attribute values and unites all pairs that match in this attribute. It requires a data structure in line 6 that can be implemented by another distributed key-value store. As in [11, Section 21], we need a pointer for all entities to a parent entity; the chain of pointers must lead to

Algorithm 1 Union-Find ER by distributed key-value store

input: Entity set E; one attribute store for each attribute with attribute value as key and entity ID as value; another auxiliary store with both key and value as entity ID.
output: $E' = ER(E)$

```
1: for all entities e do
2:   for ℓ = 1, ..., k do
3:     for all r ∈ e do
4:       for all IDs i with attribute matching a_rℓ do
5:         erase i from the attribute store
6:         union(i, ID(e))
```

Algorithm 2 BFS ER by distributed key-value store

input: Entity set E; one attribute store for each attribute with attribute value as key and entity ID as value; another entity store with entity ID as key and the complete entity data as value.
output: $E' = ER(E)$

```
1: while not at the end of the entity store do
2:   get next entity e from entity store
3:   erase e from all attribute stores and put its records into Q
4:   while Q ≠ ∅ do
5:     get next r from Q
6:     for ℓ = 1, ..., k do
7:       for all entities e' with attribute matching a_rℓ do
8:         erase e' from all attribute stores
9:         update e by e ∪ e' in the entity store.
10:        add all records of e' to Q
```

the root. We may implement pointers by a key-value store where the key is the entity ID and the value is the parent ID. To save work, optionally in line 5 we may remove those record-attribute pairs that are already considered for merging.

Since the components are assumed to be small, a simple union-find implementation suffices. We apply path collapsing: whenever a pair of entities are united, both have to seek down to the root containing the component IDs. Along these paths, we may replace all pointers directly to the root, thus allowing a near optimal average logarithmic search time.

The theoretical complexity of the algorithm is equal to that of union-find, i.e. slightly better than $O(n \log n)$ [11]. The amount of communication equals to the total size of the data, i.e. this algorithm is optimal with respect to the bounds in Section 3.

The second implementation (Algorithm 2) performs breadth-first search as in [11, Section 22]. We use queues storing new records find to be merged into the current entity. We iterate through records and, as in Algorithm 1, we obtain all matching entities by using the attribute stores. We immediately delete these records from the attribute stores and update the entity store to avoid infinite loops in a component.

Efficiency of the algorithm depends both on the indexing tools used, and on properties of the input data set (e.g. how many matching records it contains), as well as on the match logic (how many attributes there are, are they similarity-based etc.).

Algorithm 3 ER by Map-Reduce

input: Entity set E over a distributed file system.
output: $E' = ER(E)$

```
1: for ℓ = 1, ..., k do
2:   sort records r by attribute value a_rℓ
3:   for all attribute values a do
4:     for all pairs of records r, r' with a_rℓ = a_r'ℓ = a do
5:       write (ID(r), ID(r')) to graph G
6: Map-Reduce connected components(G)
7: sort records by component ID and merge groups of identical ID
```

We need to be able to list all attribute values matching $a_{r\ell}$ in line 4 of Algorithm 1 and line 7 of Algorithm 2. While this is trivial if the match function is the equality, our algorithms are suitable for similarity functions by an appropriate distributed similarity index. We may also trivially handle multi-value attributes, or, as in our case, name and maiden name by fetching both values from the corresponding attribute index.

In Algorithm 2 we may apply complex multi-attribute rules or learning based approaches as well, since we have access to the entire entity data via the entity index. For Algorithm 1 for example we handle a feature consisting of name, birth date and mother's name by indexing the concatenation of the three attributes as one single string.

The theoretical complexity of the algorithm is equal to that of BFS, i.e. $O(n \log n)$ [11]. The amount of communication equals to the total size of the data, i.e. this algorithm is also optimal with respect to the bounds in Section 3.

Finally we note that Algorithm 2 can take further advantage of parallelism: we may run the algorithm in several copies, each reading entities depending on a hash value of their ID. An additional step is needed to resolve the cases when two parallel runs accidentally reach the same merged entity starting out with an initial entity their data portion that we do not discuss here. If we distribute computation to t nodes, we may basically achieve a speedup factor of t if the data is randomly split to both the computing and the data store servers.

5. ALGORITHM BASED ON MAP-REDUCE

Our next algorithm is based on Hadoop [42], an open source implementation of the Map-Reduce framework [12]. It can be considered an extension of the Map-Reduce set similarity join of [38]; note that we may enhance our algorithm by techniques such as basic or indexed kernels from that result. Compared to a set similarity join, key difference is that we have to merge records over several attributes that will eventually require a connected component algorithm. This latter task can be solved by iterated matrix multiplication [11, Section 25]; a similar implementation is given in the Hadoop-based Pegasus framework [24].

Our Map-Reduce algorithm is split into two parts. The first part, Algorithm 3, iterates through all attributes. For each, it sorts attribute values and records all potential matches in a graph file. Then the connected component Algorithm 4 is called that assigns a component ID to all records. Finally, the last line of the main algorithm merges all records with the same ID. In this step, additional split heuristics can be implemented to undo some of the unnecessary merges simi-

Algorithm 4 Map-Reduce connected components

input: Graph G of record IDs.
output: Component ID for all record IDs.

```
1: sort G to form sequences S_i = {i, ID_i, list of edges (i, j)}
2: change = true
3: while change = true do
4:    change = false
5:    Map:
6:    for all IDs i do
7:       for all IDs j with (i, j) ∈ S_i do
8:          emit ID_i to reducer j
9:       emit entire S_i to reducer i
10:   Reduce:
11:   for all reducers j do
12:      ID' = min of all ID values received
13:      if ID' < ID_j then
14:         change = true
15:         replace ID_j by ID in S_j
16:      write S_j
```

lar to the algorithms in Section 4.

In Algorithm 3 we assign IDs to records as follows. If there are entities that consist of more than one record at start, we spit it into two records, both with the same ID. By a slight abuse of notation we may even handle entities with multiple values a and a' for certain attributes: we may place its ID multiple times, for both a, a' and possibly more, into an array to be sorted in line 2.

Note that placing an entity multiple times in line 2 we may also handle complex features such as the combination of a name and maiden name. In this case we consider name and maiden name as a single attribute with multiple values and proceed as above.

We describe the connected component Algorithm 4 in detail. The algorithm implements the matrix multiplication based all-pairs reachability algorithm of [11, Section 25] in a way similar to [24]. Two ingredients are the reduction of the problem to iterated matrix multiplication with a modified associative operation and the implementation of the matrix operation over Hadoop. For the first, let us replace addition by the minimum function and let

$$ID_j = \min(ID_j, \min_{i:(ij) \text{ is an edge}} \{ID_i\}). \quad (1)$$

In iteration s, this method selects the minimum value in the s step neighborhood of every record. If we record the fact that some ID_j decrease in an iteration, then we may terminate if there is no change.

Finally we show how to compute the matrix-vector multiplication type step of (1) by Map-Reduce. Starting at line 5, mapper i sends its current ID to reducer j for all edges ij in the graph to prepare the data needed to compute (1). In addition, reducer j starting in line 10 must write data S_j suitable for the next matrix-vector multiplication iteration. In addition to ID_j, this S_j must contain the edges out of record j. For this purpose, mapper i sends its entire data S_i to reducer i, competing the description of the algorithm.

The running time of the algorithm is $O(\ell(n \log n)/t)$ for the ℓ mergesort operations over t servers and $O(sn/t)$ for connected components over t servers where s is the size of the largest component. The implementation transmits all data ℓ times during the sort operations, hence requires ℓ times more communication than the optimum. This algorithm is the least efficient in theory also since Hadoop introduces additional disk I/O operations when storing partial results.

Algorithm 5 Bulk Synchronous Parallel (BSP) ER

input: Entity set E.
output: $E' = ER(E)$

```
1: Partition entities E = {E_1, ..., E_d} to data nodes 1...d.
2: for i = 1, ..., d do
3:    send E_i to data node i
4: start master node and data nodes 1, ..., d
5: wait for termination
6: start BSP_connected_components on data nodes 1, ..., d
7: wait for termination
8: merge results from data nodes 1, ..., d to E'
```

Algorithm 6 Data node algorithm for BSP ER

input: Entity set E_d such that each record corresponds to a unique entity.
output: E'_d and list L_e for each entity $e \in E'_d$

```
1: E'_d = ER(E_d)
2: for ℓ = 1, ..., k do
3:    start phase 2ℓ − 1
4:    sort records r ∈ E'_d by attribute value a_rℓ
5:    for all attribute values a in sorted order do
6:       m_a ← {ID(r_1), ID(r_2), ...} for all records r_i with a_{r_i ℓ} = a
7:       send m_a to master node
8:    send "end" to master node
9:    end phase 2ℓ − 1; start 2ℓ
10:   repeat
11:      receive a, ID_1, d_1, ID_2, d_2, ... from master
12:      add ID_1, d_1, ID_2, d_2, ... to entity list L(e) holding r ∈ e with a_rℓ = a
13:   until master node sends "end"
14:   end of phase 2ℓ
```

6. A BULK SYNCHRONOUS PARALLEL ALGORITHM

In his seminal paper, Valiant [37] introduced the Bulk Synchronous Parallel framework for distributed algorithms. This framework is reintroduced as evolving infrastructures for distributed processing both proprietary as Google's Pregel [30] and open source such as HAMA [34]. The idea is to divide the algorithm into phases divided by barriers. In each phase, nodes may produce messages to other nodes that they receive in the next phase.

In our HAMA implementation, the first step of the main Algorithm 5 is to distribute the data to servers (line 1). We may use a strong attribute or set of attributes for distribution that, similar to blocking ER algorithms, may result in significant speedup. Then a merge-sort algorithm is started over a master and d data servers in line 4. After initiating connected component computation in line 6 over the same data nodes, a last step is called that merges all records into

Algorithm 7 Master node algorithm for BSP ER

input: list of data nodes $1, \ldots, d$.

1: **for** $\ell = 1, \ldots, k$ **do**
2: ***start phase*** $2\ell - 1$
3: **for** data nodes $p = 1, \ldots, d$ **do**
4: receive $m_a^{(p)} = \{ID_1^{(p,a)}, ID_2^{(p,a)}, \ldots\}$ from node p
5: ***end phase*** $2\ell - 1$***; start*** 2ℓ
6: **for all** attribute values a in sorted order, merging the message queues from nodes $1, \ldots, d$ **do**
7: **if** $s > 1$ for nodes $p_1, \ldots, p_s$ that submitted message $m_a^{(p)}$ for attribute value a **then**
8: **for** $i = 1, \ldots, s$ **do**
9: submit a to node p_i
10: **for** $j = 1, \ldots, s$ with $j \neq i$ **do**
11: submit $(p_j, ID_t^{(p_j,a)})$ to node p_i for all t
12: ***end phase*** 2ℓ
13: terminate master node after phase k

the parent entity e at the data server holding e.

The data node Algorithm 6 assumes, as in the Map-Reduce algorithm, that entities with multiple values for certain attributes are split into records and uses the same $ID(r)$ notation for obtaining the entity ID. A farm of d such data nodes perform merge-sort with a master running Algorithm 7. Note that there may be more masters to speed up merge-sort by e.g. hashing the attribute value sets.

We perform merge-sort of the attribute values in iterations over attributes $\ell = 1, \ldots, k$. Algorithms 6 and 7 run in BSP phases $1, \ldots, 2k$. In the odd numbered phases data nodes send the values a along with the list m_a of entity IDs holding value a. The master collects all values shared between different data nodes. In the even numbered phases the master sends out these values a to each data node holding a, followed by the list of entity IDs and corresponding data node ID. Data nodes store these lists at the entity e holding a for the connected component algorithm.

Compared to the current HAMA based implementation, the above algorithm could be more efficient by using primitives for merging sorted lists. In theory the master(s) could balance the amount of data received from data nodes by, for the next attribute value a, immediately discarding if the value appears only once or emitting if matching is found. In this case the master could always requesting the next data from those that hold a. In its current state however we are even less efficient as communication in phases $1 \ldots 2k$ have be broken into smaller blocks so that all messages from data nodes fit into the master's memory.

Finally we turn to the connected component Algorithm 8 implemented as the classical "minimum-over-graph" algorithm. We proceed in BSP phases until there is no change in the entity IDs that we perform by marking entity e if its ID changes in the phase. In a phase, every marked entity sends its ID to its neighbors and unmarks itself. Receiving entities change their ID to minimum and mark themselves if there is a change. It is easy to see that in phase t, each entity gets the minimum ID of its neighbors within at most t steps.

The algorithm can be made more flexible by implementing more complex functions at the master nodes. For example it may collect all attribute values in a similarity index and send response to merge all similar entity values.

Algorithm 8 BSP Connected Component algorithm: finds minimal ID of connected components of the graph in phases until termination

input: E'_d and list L_e for each entity $e \in E'_d$
output: parent entity ID and node $ID(e), P(e)$ for each entity $e \in E'_d$

1: sort E'_d by ID
2: **for all** $(ID, p) \in L(e)$ for entities $e \in E'_d$ with $ID < ID(e)$ **do**
3: mark e
4: $ID'(e) \leftarrow ID$
5: $P(e) = p$
6: **while** there are marks on entities **do**
7: ***start next phase***
8: **for all** $(ID, p) \in L(e)$ for marked entities e with $ID > ID(e)$ **do**
9: send $ID(e), ID, P(e)$ to data node p
10: unmark e
11: **while** messages to the current data node d exists **do**
12: receive ID, ID', p
13: find e with $ID(e) = ID$
14: **if** $ID'(e) > ID'$ **then**
15: mark e
16: $ID'(e) \leftarrow ID$
17: $P(e) = p$
18: ***start last phase***: move all data to the lowest ID representative that will hold the resolved entity

The running time of the algorithm is $O(ER(n/t))$ for the resolution of the initial data at each server where $ER(n)$ is the running time of an optimal sequential ER algorithm. Phases $1, \ldots, 2k$ communicate all data exactly once and run linear in the data transmitted, hence they altogether take $O(n)$ time. Finally the connected component algorithm takes $O(sn/t)$ time over t servers where s is the size of the largest component. By the assumption that s is very small, this algorithm is theoretically the best and also uses near optimal amount of communication.

7. EXPERIMENTS

Experiments were performed on 15-server Linux farms containing identical dual core 3 GHz Pentium CPUs, 4 GB of main memory. Software versions were Sun Java 1.6, Project Voldemort 0.81, Hadoop 0.20.3, and a patched[1] HAMA 0.3.0. Voldemort was configured to use a replication factor of 1. We configured our systems to use all available internal memory. The largest test data sets do not fit in the memory of one node but still fit in the entire 15×4 GB of the cluster.

The data set is provided by AEGON Hungary Insurance Ltd.[2] containing approximately 20 million client records. Records consist of both personal attributes (names, birth data, tax number, etc.) and internal identifiers. According to preliminary estimates and experimental results, each client is duplicated into an average of 1.95 records. The size of the record set is 1.7 GB in a flat CSV file.

We used random sampling to obtain smaller subsets. We

[1]We fixed the distributed double barrier implementation.
[2]The AEGON Hungary has been a member of the AEGON Group since 1992, one of the world's largest life insurance and pension groups.

also used selection heuristics to influence the count of records per user. For example, selecting all records for the family name 'Smith' instead of random sample will increase the match count. We created larger data by replication and random permutation. In each replica, we added a version tag to all attributes so that the original structure of matches was preserved but no new matches were introduced between replicas.

In the experimental settings we used a real-world match condition composed of 9 attributes in 5 orthogonal matching features. Attribute matches included simple attribute-equality testing, e.g. "two entities match, if they have common tax numbers". Match in names however also include maiden name as second attribute. Finally we used more complicated match conditions, e.g. match in birth data composed of four attributes requiring multiple attribute-value equality testing. All of our algorithms produce the same exact results determined by the match conditions, therefore accuracy is not measured, only verified.

7.1 Comparison of overall running times

Figure 2 and 3 depict the overall execution times including reading the input and finalizing the output. By these figures we observe the superiority of Map-Reduce: It was able to process one order of magnitude larger input sets than the others. Given the fixed duplication factor obtained by our data replication procedure, the running time up to 600 million records appears to be linear in input size, which makes the algorithm very useful in practice. The effect of varying duplication factor is tested later in Fig. 4. We did not lay stress on optimization, therefore further performance improvements could be achieved by fine tuning Hadoop and the Java code.

Theoretically, the BSP algorithm requires the least communication between nodes, therefore we expected similar or better behavior compared to the Map-Reduce version. BSP, however, performed poorly. The reasons can be both non-optimal coding, bad initial data distribution to nodes, or the experimental state of the Hama framework. BSP therefore requires further investigation. Re-implementing the algorithm using sockets could clear up the role of the infrastructure.

Our key-value store algorithms give the worst performance. This is no surprise since they are basically sequential in that they process the input one-by-one on a single main node. The main bottleneck turns out neither CPU nor I/O but the communication between the main node and the key-value store nodes. Distributing not just the data but also the computation would be therefore useful, however that requires a careful locking mechanism over the feature indexes and the entity store. Distributed key-value stores are slower than their non-distributed counterpart (in our case, BerkeleyDB) for small data (see [36] for related experiments). The main reason for using distribution is that the performance of single-node key-value stores falls drastically when reaching memory limits.

Although our BFS distributed key-value store algorithm is the slowest, it has the advantage of applying a general match logic: Attribute indexes provide candidates for a general match function. The main reason of the relatively poor performance of the BFS algorithm is that for larger data sets we get too many candidates for sophisticated matching conditions and indexing a single attribute of a sophisticated feature is simply not efficient enough.

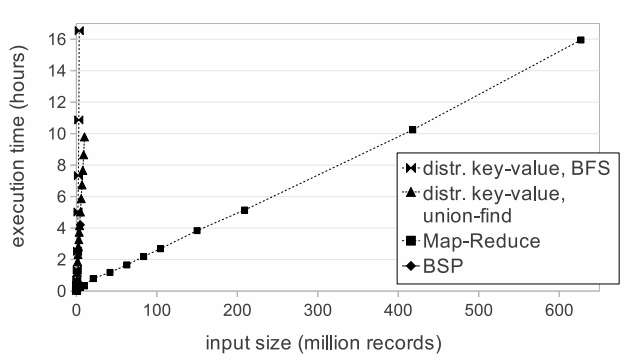

Figure 2: Execution time against input size for all algorithms

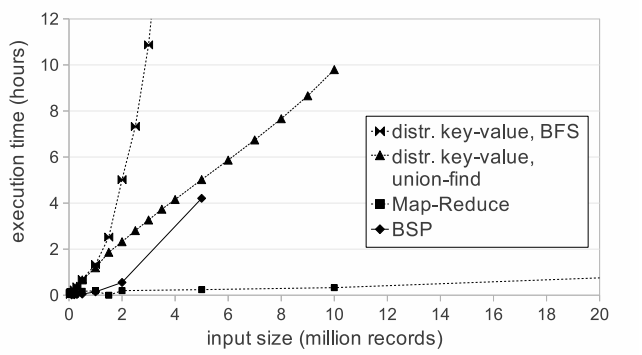

Figure 3: Execution time against input size for all algorithms, below 20 million records

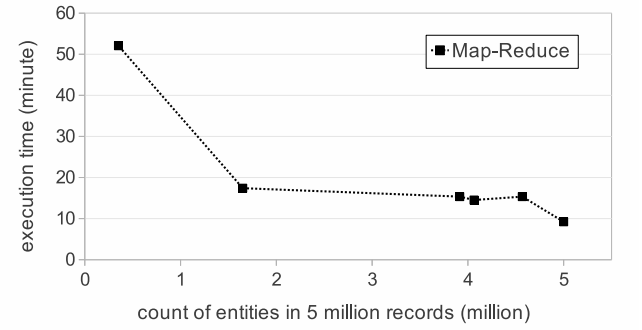

Figure 4: Execution times plotted against the number of merged entities in 5 million records

Figure 4 shows the impact of match count on the execution times of the Map-Reduce algorithm. As we expected, increasing match count (less hidden entity) results in larger running times: more edges appear in the match graph and more iterations have to be performed when finding connected components.

7.2 Running times of algorithm phases

Except for the key-value store procedures, all of our algorithms can be separated to different phases. Our first algorithm, the Union-Find key-value store one finds all match-

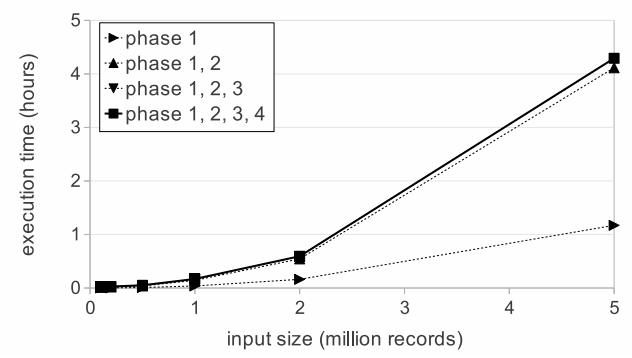

Figure 5: Cumulative execution times of the distinct phases in the BSP algorithm

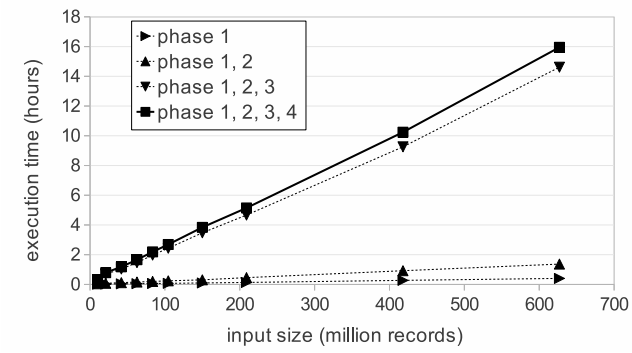

Figure 6: Cumulative execution times of the distinct phases in the Map-Reduce algorithm

ings first. Then in a second round it produces the output based on the union-find store. The time needed for the second round was approximately 20% of the first phase in all settings.

For the BSP algorithm, Figure 5 depicts the time required by its phases. In Phase 1, we read the input and resolve the initial datasets per node by an in-memory ER implementation similar to the union-find variation. Phase 2 covers the main process of master and data nodes and Phase 3 the connected component search. The output is finalized in Phase 4. The running time share of the phases seems to be fixed over our range of input sizes with most of the time devoted to Phase 2, the main part of the procedure.

Figure 6 shows running times of the Map-Reduce implementation phases. This implementation first sorts the data by feature values and produces the edge matrix in Phase 2. The edge matrix is used to find connected components in Phase 3, taking the majority of execution time. Note the complementarity with the BSP algorithm where the time for connected component computation is negligible. In Phase 4, we produce the output based on the list of the connected components.

Conclusion and Future Work

We described algorithms to solve the ER problem over three different types of distributed software architectures. Our algorithms also proved to be useful in practice.

Distributed key-value stores provide the easiest extensions towards similarity search and fuzzy match functions: Instead of lexicographical or numeric indexes, we may simply use distributed similarity search structures. This solution is however the slowest of the alternatives. A further possible way of parallel processing have to be studied: If the resolved entity set resides in the key-value store, then parallel resolving tasks can speed up the entire executions. However, the soundness of this parallelization have to be ensured by locking mechanisms not covered in this paper.

Hadoop is apparently a mature Map-Reduce infrastructure capable of efficiently implementing ER algorithms. For Bulk Synchronous Parallel algorithms, there is no open source infrastructure of similar level of maturity yet. We demonstrated that HAMA, an incubatory project is capable of supporting ER algorithms and we expect BSP implementations may eventually be superior since they reduce cross-server communication and may completely eliminate slow disk I/O operations. Also, partitioning along a strong feature may result in significant speedup and our solution may also combine well with feature-based blocking.

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.

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.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *Proc. of the 27th Int. Conf. on Distributed Computing Systems*. IEEE, 2007.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.

[4] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1):5, 2007.

[5] A. Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1997.

[6] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 268–279. ACM, 2000.

[7] P. Christen. Development and user experiences of an open source data cleaning, deduplication and record linkage system. *ACM SIGKDD Explorations Newsletter*, 11(1):39–48, 2009.

[8] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2011.

[9] P. Christen, T. Churches, and M. Hegland. Febrl - a parallel open source data linkage system. In *PAKDD*, volume 3056 of *Lecture Notes in Computer Science*, pages 638–647. Springer, 2004.

[10] 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. ACM, 2009.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT press, 2001.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[14] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proc. of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005.

[15] D. Donoho. Neighborly polytopes and sparse solution of underdetermined linear equations. Technical report, Stanford University, 2004.

[16] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–16, 2007.

[17] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[18] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[19] P. Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[20] 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.

[21] M. Hernández and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data mining and knowledge discovery*, 2(1):9–37, 1998.

[22] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[23] B. Kalyanasundaram and G. Schintger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5:545, 1992.

[24] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[25] H. Kawai, H. Garcia-Molina, O. Benjelloun, D. Menestrina, E. Whang, and H. Gong. P-swoosh: Parallel algorithm for generic entity resolution. Technical report, Stanford, 2006.

[26] H.-s. Kim and D. Lee. Parallel linkage. In *Proc. of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07. ACM, 2007.

[27] T. Kirsten, L. Kolb, M. Hartung, A. Gross, H. Kpcke, and E. Rahm. Data partitioning for parallel entity matching. *Computing Research Repository*, 2010.

[28] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69:197–210, February 2010.

[29] 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.

[30] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the 2010 Int. Conf. on Management of Data*, pages 135–146. ACM, 2010.

[31] D. Menestrina, S. E. Whang, and H. Garcia-Molina. Evaluating entity resolution results. *Proc. VLDB Endow.*, 3:208–219, September 2010.

[32] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *SIGMOD DMKD*, 1997.

[33] Project Voldemort. A distributed key-value storage system. `http://project-voldemort.com/`.

[34] S. Seo, E. Yoon, J. Kim, S. Jin, J. Kim, and S. Maeng. HAMA: An efficient matrix computation with the MapReduce framework. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 721–726. IEEE, 2010.

[35] C. I. Sidló. Generic entity resolution in relational databases. In *Advances in Databases and Information Systems*, volume 5739 of *LNCS*, pages 59–73. Springer, 2009.

[36] C. I. Sidló. Entity resolution with heavy indexing. In *Proc. of the 2011 Int. Conf. on Advances in Databases and Information Systems*, CEUR Workshop Proceedings, 2011.

[37] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[38] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proc. of the 2010 Int. Conf. on Management of Data*, pages 495–506. ACM, 2010.

[39] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proc. of the VLDB Endow.*, 1(2):1253–1264, 2008.

[40] S. E. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *Proc. VLDB Endow.*, 3:1326–1337, September 2010.

[41] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *Proc. of the 35th Int. Conf. on Management of Data*, pages 219–232. ACM, 2009.

[42] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[43] M. Yakout, A. K. Elmagarmid, H. Elmeleegy, M. Ouzzani, and A. Qi. Behavior based record linkage. *Proc. VLDB Endow.*, 3:439–448, September 2010.