An Innovative Framework for Combining Set Similarity Join Algorithms and Clustering

Leonardo Andrade Ribeiro[1], Alfredo Cuzzocrea[2],
Karen Aline Alves Bezerra[3], Ben Hur Bahia do Nascimento[3], and Massimiliano Nolich[2]

[1] Instituto de Informática, Universidade Federal de Goiás, Goiânia, Goiás, Brazil
laribeiro@inf.ufg.br
[2] DIA Department, University of Trieste and ICAR-CNR, Italy
alfredo.cuzzocrea@dia.units.it, mnolich@units.it
[3] Departmento de Ciência da Computação, Universidade Federal de Lavras, Lavras, Brazil
karen.bezerra@posgrad.ufla.br, bhn@computacao.ufla.br

Abstract. Centered around the *data cleaning and integration* research area, in this paper we propose *SjClust*, a framework to integrate similarity join and clustering into a single operation. The basic idea of our proposal consists in introducing a variety of cluster representations that are smoothly merged during the set similarity task, carried out by the join algorithm. An optimization task is further applied on top of such framework. The framework exposes a wide number of application scenarios where it can be used effectively and efficiently.

1 Introduction

Data cleaning and integration (e.g., [11, 5]) found on *duplicate record identification* (e.g., [6, 20]), which aims at detecting duplicate records that represent the same real-world entity. This is becoming more and more relevant in emerging *big data research* (e.g., [21, 10, 18]), as a plethora of real-life applications are characterized by the presence of multiple records representing the same real-world entity, which practically plagues every large database. Such records are often referred to as *fuzzy duplicates* (duplicates, for short), because they might not be exact copies of one another. Duplicates arise due to a variety of reasons, such as typographical errors and misspellings, different naming conventions, and as a result of the integration of data sources storing overlapping information.

Duplicates degrade the quality of the data delivered to application programs, thereby leading to a myriad of problems. Some examples are misleading data mining models owing to erroneously inflated statistics, inability of correlating information related to a same entity, and unnecessarily repeated operations, e.g., mailing, billing, and leasing of equipment. Duplicate identification is thus of crucial importance in data cleaning and integration.

Duplicate identification is computationally very expensive and, therefore, typically done offline. However, there exist important application scenarios that demand (near) real-time identification of duplicates. Prominent examples are data exploration [8], where new knowledge has to be efficiently extracted from databases without a clear definition of the information need, and virtual data integration [5], where the integrated data is not materialized

SEBD 2018, June 24-27, 2018, Castellaneta Marina, Italy. Copyright held by the author(s).

and duplicates in the query result assembled from multiple data sources have to be identified — and eliminated — on-the-fly. Such scenarios have fueled the desire to integrate duplicate identification with processing of complex queries [1] or even as a general-purpose physical operator within a DBMS [4].

An approach to realize the above endeavor is to employ *similarity join* in concert with a *clustering algorithm* [7]. Specifically, similarity join is used to find all pairs of records whose similarity is not less than a specified threshold; the similarity between two records is determined by a *similarity function*. In a post-processing step, the clustering algorithm groups together records using the similarity join results as input. For data of string type, *set similarity join* is an appealing choice for composing a duplicate identification operator. Set similarity join views its operands as sets — strings can be easily mapped to sets. The corresponding similarity function assesses the similarity between two sets in terms of their overlap and a rich variety of similarity notions can be expressed in this way [4]. Furthermore, a number of optimization techniques have been proposed over the years [15, 4, 2, 20, 14, 19] yielding highly efficient and scalable algorithms.

The strategy of using a clustering algorithm strictly for post-processing the results of set similarity join has two serious drawbacks, however. First, given a group of n, sufficiently similar, duplicates, the set similarity join performs $\binom{n}{2}$ similarity calculations to return the same number of set pairs. While this is the expected behavior considering a similarity join in isolation, it also means that repeated computations are being performed over identical subsets. Even worse, we may have to perform much more additional similarity calculations between non-duplicates: low threshold values are typically required for clustering algorithms to produce accurate results [7]. Existing filtering techniques are not effective at low threshold values and, thus, there is an explosion of the number of the comparisons at such values. Second, the clustering is a blocking operator in our context, i.e., it has to consume all the similarity join output before producing any cluster of duplicates as result element. This fact is particularly undesirable when duplicate identification is part of more complex data processing logic, possibly even with human interaction, because it prevents pipelined execution.

In this paper, we propose *SjClust*, a framework to integrate set similarity join and clustering into a single operation, which addresses the above issues. The main idea behind our framework is to represent groups of similar sets by a *cluster representative*, which is incrementally updated during the set similarity join processing. Besides effectively reducing the number similarity calculations needed to produce a cluster of n sets to $O(n)$, we are able to fully leverage state-of-the-art optimization techniques at high threshold values, while still performing well at low threshold values where such techniques are much less effective. Indeed, the resulting composed algorithm is even up to an order of magnitude faster than the original set similarity join algorithm for low threshold values. Moreover, we exploit set size information to identify when no new set can be added to a cluster; therefore, we can then immediately output this cluster and, thus, avoid the blocking behavior. Furthermore, there exists a plethora of clustering algorithms suitable for duplicate identification and no single algorithm is overall the best across all scenarios [7]. Thus, versatility in supporting a variety of clustering methods is essential. Our framework smoothly accommodates various cluster representation and merging strategies, thereby yielding different clustering methods for each combination thereof.

This paper is the short version of the papers [13, 12], where we present the main results of our research.

2 Fundamental Concepts and Background Knowledge

ncepts and definitions related to set similarity joins before present important optmization techniques. Then, we describe a general set similarity join algorithm, which provides the basis for our framework.

2.1 Basic Concepts and Definitions

We map strings to *sets of tokens* using the popular concept of *q-grams*, i.e., sub-strings of length q obtained by "sliding" a window over the characters of an input string v. We (conceptually) extend v by prefixing and suffixing it with $q-1$ occurrences of a special character *"\$"* not appearing in any string. Thus, all characters of v participate in exact q q-grams. For example, the string *"token"* can be mapped to the set of *2*-gram tokens $\{$*\$t*, *to*, *ok*, *ke*, *en*, *n\$*$\}$. As the result can be a multi-set, we simply append the symbol of a sequential ordinal number to each occurrence of a token to convert multi-sets into sets, e.g, the multi-set $\{$a,b,b$\}$ is converted to $\{$*a∘1*, *b∘1*, *b∘2*$\}$. In the following, we assume that all strings in the database have already been mapped to sets.

We associate a weight with each token to obtain *weighted sets*. A widely adopted weighting scheme is the Inverse Document Frequency (*IDF*), which associates a weight $idf(tk)$ to a token tk as follows: $idf(tk)=ln(1+N/df(tk))$, where $df(tk)$ is the *document frequency*, i.e., the number of strings a token tk appears in a database of N strings. The intuition behind using IDF is that rare tokens are more discriminative and thus more important for similarity assessment. The weight of a set r, denoted by $w(r)$, is given by the weight summation of its tokens, i.e., $w(r) = \sum_{tk \in r} w(tk)$.

We consider the general class of set similarity functions. Given two sets r and s, a set similarity function $sim(r,s)$ returns a value in $[0,1]$ to represent their similarity; larger value indicates that r and s have higher similarity. Popular set similarity functions are defined as follows.

Definition 1 (Set Similarity Functions). *Let r and s be two sets. We have:*

- *Jaccard similarity:* $J(r,s) = \frac{w(r \cap s)}{w(r \cup s)}$.
- *Dice similarity:* $D(r,s) = \frac{2 \cdot w(r \cap s)}{w(r)+w(s)}$.
- *Cosine similarity:* $C(r,s) = \frac{w(r \cap s)}{\sqrt{w(r) \cdot w(s)}}$

We now formally define the set similarity join operation.

Definition 2 (Set Similarity Join). *Given two set collections $\mathcal{R}$ and $\mathcal{S}$, a set similarity function sim, and a threshold τ, the set similarity join between $\mathcal{R}$ and $\mathcal{S}$ returns all scored set pairs $\langle (r,s), \tau\prime \rangle$ s.t. $(r,s) \in \mathcal{R} \times \mathcal{S}$ and $sim(r,s) = \tau\prime \geq \tau$.*

In this paper, we focus on self-join, i.e., $\mathcal{R} = \mathcal{S}$; we discuss the extension for binary inputs in Section 2.3. For brevity, we use henceforth the term similarity function (join) to mean set similarity function (join). Further, we focus on the Jaccard similarity and the IDF weighting scheme, i.e., unless stated otherwise, $sim(r,s)$ and $w(tk)$ denotes $J(r,s)$ and $idf(tk)$, respectively.

Example 1. Consider the sets r and s below

$$r = \{\mathbf{A}, \mathbf{B}, C, \mathbf{D}, \mathbf{E}\}$$
$$s = \{\mathbf{A}, \mathbf{B}, \mathbf{D}, \mathbf{E}, F\}$$

and the following token-IDF association table:

tk	*A*	*B*	*C*	*D*	*E*	*F*
idf (*tk*)	*1.5*	*2.5*	*2*	*3.5*	*0.5*	*2*

Thus, we have $w(r) = w(s) = 10$ and $w(r \cap s) = 8$; thus $sim(r,s) = \frac{8}{10+10-8} \approx 0.66$.

2.2 Optimization Techniques

Similarity functions can be equivalently represented in terms of an *overlap bound* [4]. Formally, the overlap bound between two sets r and s, denoted by $O(r,s)$, is a function that maps a threshold τ and the set weights to a real value, s.t. $sim(r,s) \geq \tau$ iff $w(r \cap s) \geq O(r,s)$. The similarity join can then be reduced to the problem of identifying all pairs r and s whose overlap is not less than $O(r,s)$. For the Jaccard similarity, we have $O(r,s) = \frac{\tau}{1+\tau} \cdot (w(r) + w(s))$.

Further, similar sets have, in general, roughly similar weights. We can derive bounds for immediate pruning of candidate pairs whose weights differ enough. Formally, the weight bounds of r, denoted by $min(r)$ and $max(r)$, are functions that map τ and $w(r)$ to a real value s.t. $\forall s$, if $sim(r,s) \geq \tau$, then $min(r) \leq w(s) \leq max(r)$ [15]. Thus, given a set r, we can safely ignore all other sets whose weights do not fall within the interval $[min(r), max(r)]$. For the Jaccard similarity, we have $[min(r), max(r)] = \left[\tau \cdot w(r), \frac{w(r)}{\tau}\right]$. We refer the reader to [16] for definitions of overlap and weight bounds of several other similarity functions, including Dice and Cosine.

We can prune a large share of the comparison space by exploiting the *prefix filtering principle* [15, 4], which allows discarding candidate pairs by examining only a fraction of the input sets. We first fix a global order $\mathcal{O}$ on the universe $\mathcal{U}$ from which all tokens are drawn. A set $r' \subseteq r$ is a prefix of r if r' contains the first $|r'|$ tokens of r. Further, $pref_\beta(r)$ is the shortest prefix of r, the weights of whose tokens add up to more than β. The prefix filtering principle is defined as follows.

Definition 3 (Prefix Filtering Principle [4]). *Let r and s be two sets. If* $w(r \cap s) \geq \alpha$, *then* $pref_{\beta_r}(r) \cap pref_{\beta_s}(r) \neq \varnothing$, *where* $\beta_r = w(r) - \alpha$ *and* $\beta_s = w(s) - \alpha$, *respectively.*

We can identify all candidate matches of a given set r using the prefix $pref_\beta(r)$, where $\beta = w(r) - min(r)$. We denote this prefix simply by $pref(r)$. It is possible to derive smaller prefixes for r, and thus obtain more pruning power, when we have information about the set weight of the candidate sets, i.e., if $w(s) \geq w(r)$ [2] or $w(s) > w(r)$ [14]. Note that prefix overlap is a condition necessary, but not sufficient to satisfy the original overlap constraint: an additional verification must be performed on the candidate pairs. Finally, the number of candidates can be significantly reduced by using the *inverse document frequency ordering*, $\mathcal{O}_{idf}$, as global token order to obtain sets ordered by decreasing IDF weight . The idea is to minimize the number of sets agreeing on prefix elements and, in turn, candidate pairs by

For ease of notation, the parameter τ is omitted.

A secondary ordering is used to break ties consistently (e.g., the lexicographic ordering).

shifting lower frequency tokens to the prefix positions — recall that higher IDF weights are associated to low-frequency tokens.

Example 2. Consider the sets r and s in Example 1 and $\tau = 0.6$. We have $O(r,s) = 7.5$; $[min(r), max(r)]$ and $[min(s), max(s)]$ are both $[6, 16.7]$. By ordering r and s according to O_{idf} and the IDF weights in Example 1, we obtain:

$$r = [D, B, C, A, E]$$
$$s = [D, B, F, A, E].$$

We have $pref(r) = pref(s) = [D]$.

2.3 Similarity Join Algorithms: Definitions and Usage

Similarity join algorithms based on inverted lists are effective in exploiting the previous optimizations [15, 2, 20, 14]. Most of such algorithms have a common high-level structure following a filter-and-refine approach.

Algorithm 1 formalizes the steps of a similarity join algorithm. The algorithm receives as input a set collection sorted in increasing order of set weights, where each set is sorted according to O_{idf}. An inverted list I_t stores all sets containing a token t in their prefix. The input collection R is scanned and, for each *probe set* r, its prefix tokens are used to find *candidate sets* in the corresponding inverted lists (lines 4–10); this is the *candidate generation phase*, where the map M is used to associate candidates to its accumulated overlap score os (line 3). Each candidate s is dynamically removed from the inverted list if its weight is less than $min(r)$ (lines 6–7). Further filters, e.g., filter based on overlap bound, are used to check whether s can be a true match for r, and then the overlap score is accumulated, or not, and s can be safely ignored in the following processing (lines 8–10). In the *verification phase*, r and its matching candidates, which are stored in M, are checked against the similarity predicate and those pairs satisfying the predicate are added to the result set. To this end, the *Verify* procedure (not shown) employs a merge-join-based algorithm exploiting token order and the overlap bound to define break conditions (line 11)[14]. Finally, in the *indexing phase*, a *pointer* to set r is appended to each inverted list I_t associated with its prefix tokens (lines 12 and 13).

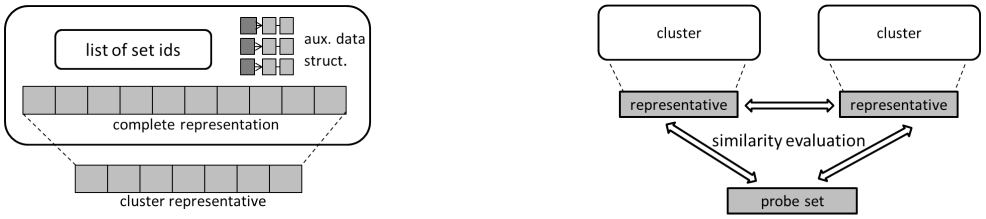
Algorithm 1 is actually a self-join. Its extension to binary joins is trivial: we first index the smaller collection and then go through the larger collection to identify matching pairs. For simplicity, several filtering strategies such positional filtering [20] and min-prefixes [14], as well as inverted list reduction techniques [2, 14] were omitted. Nevertheless, these optimizations are based on bounds and prefixes and, therefore, our discussion in the following remains valid.

3 Our Proposal: The Innovative *SjClust* Framework

We now present *SjClust*, a general framework to integrate clustering methods into similarity joins algorithms. The goals of our framework are threefold: 1) *flexibility and extensibility* by accommodating different clustering methods; 2) *efficiency* by fully leveraging existing optimization techniques and by reducing the number of similarity computations to form clusters; 3) *non-blocking behavior* by producing results before having consumed all the input, preferably much earlier.

Algorithm 1: Similarity join algorithm

```
Input: A set collection $\mathcal{R}$ sorted in increasing order of the set weight; each set is sorted
       according to $O_{idf}$; a threshold $\tau$
Output: A set $S$ containing all pairs $(r,s)$ s.t. $Sim(r,s) \geq \tau$
1  $I_1, I_2, \ldots I_{|\mathcal{U}|} \leftarrow \varnothing, S \leftarrow \varnothing$
2  foreach $r \in \mathcal{R}$ do
3      $M \leftarrow$ empty map from set id to overlap score (os)
4      foreach $t \in pref(r)$ do                                  // can. gen. phase
5          foreach $s \in I_t$ do
6              if $w(s) < min(r)$
7                  Remove $s$ from $I_t$
8              if $filter(r, s, M(s))$
9                  $M(s).os \leftarrow -\infty$                     // invalidate $s$
10             else $M(s).os = M(s).os + w(t)$
11     $S \leftarrow S \cup Verify(r, M, \tau)$                      // verif. phase
12     foreach $t \in pref(r)$ do                                  // index. phase
13         $I_t \leftarrow I_t \cup \{r\}$
14 return $S$
```



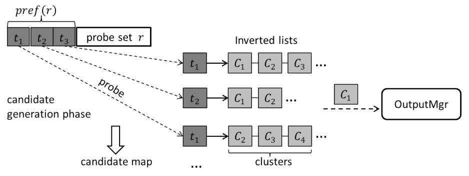
(a) Representation details. (b) Similarity evaluation.

Fig. 1. Cluster representation.

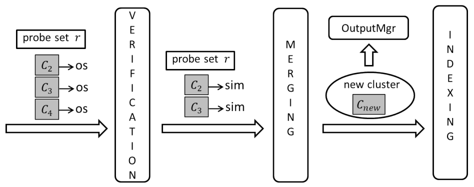
The backbone of *SjClust* is the similarity join algorithm presented in Section 2. In particular, *SjClust* operates over the same input of sorted sets, without requiring any pre-processing, and has the three execution phases present in Algorithm 1, namely, candidate generation, verification, and indexing phases. Nevertheless, there are, of course, major differences.

First and foremost, the main objects are now cluster of sets, or simply clusters. Figure 1 illustrates strategy adopted for cluster representation. The internal representation contains a list of its set element's ids, an (optional) auxiliary structure, and the cluster's *complete representation*, a set containing all tokens from all set elements. A cluster exports its external representation as the so-called *cluster representative* (or simply representative) (Figure 1(a)). Representatives are fully comparable to input sets and similarity evaluations are always performed on the representatives, either between a probe set and a cluster or between two clusters (Figure 1(b)). In the following, we use the term cluster and representative interchangeably whenever the distinction is unimportant for the discussion.

Figure 2 depicts more details on the *SjClust* framework. In the candidate generation phase, prefix tokens of the current probe set are used to find cluster candidates in the inverted lists (Figure 2(a)). Also, there is a *merging phase* between verification and indexing phases (Figure



(a) Candidate generation.



(b) *SjClust* course of evaluation.

Fig. 2. *SjClust* framework components.

2(b)). The verification phase reduces the number of candidates by removing false positives, i.e., clusters whose similarity to the probe set is less than the specified threshold. In the merging phase, a new cluster is generated from the probing set and the clusters that passed through the verification are considered for merging with it according to a *merging strategy*. In the indexing phase, references to the newly generated cluster are stored in the inverted lists associated with its prefix tokens. Finally, there is the so-called *Output Manager*, which is responsible for maintaining references to all clusters—a reference to a cluster is added to the Output Manager right after its generation in the merging phase (Figure 2(b)). Further, the Output Manager sends a cluster to the output as soon as it is identified that no new probing set can be similar to this cluster. Clusters in such situation can be found in the inverted lists during the candidate generation (Figure 2(a)) as well as identified using the weight of the probe set (not shown in Figure 2).

The aforementioned goals of *SjClust* are met as follows: flexibility and extensibility are provided by different combinations of cluster representation and merging strategies, which can be independently and transparently plugged into the main algorithm; efficiency is obtained by the general strategy to cluster representation and indexing; and non-blocking behavior is ensured by the Output Manager. Next, we provide details of each *SjClust* component.

4 Conclusions and Future Work

In this paper, we presented *SjClust*, a framework to integrate clustering into set similarity join algorithms. We demonstrated the flexibility of *SjClust* in incorporating different clustering methods by proposing several cluster representation and merging strategies. *SjClust* is an order of magnitude faster than the original set similarity join algorithm for lower thresholds, which are often needed in practice to obtain accurate results in duplicate identification. Furthermore, our proposal produces results earlier, thereby avoiding blocking behavior. We described *SjClust* and its main components in detail. Future work is mainly oriented towards enriching our framework with advanced features such as *uncertain data management* (e.g., [9]), *adaptiveness* (e.g., [3]), and *execution time prediction* (e.g, [17]).

Acknowledgments

This research was partially supported by the Brazilian agencies CNPq and CAPES.

References

1. Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3):120–131, 2015.
2. Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proc. of the WWW Conference*, pages 131–140, 2007.
3. Mario Cannataro, Alfredo Cuzzocrea, Carlo Mastroianni, Riccardo Ortale, and Andrea Pugliese. Modeling adaptive hypermedia with an object-oriented approach and xml. *WebDyn 2002*, 2002.
4. Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. of the 22nd Intl. Conf. on Data Engineering*, page 5, 2006.
5. AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.
6. Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.
7. Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.
8. Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *Proc. of the SIGMOD Conference*, pages 277–281, 2015.
9. Carson Kai-Sang Leung, Alfredo Cuzzocrea, and Fan Jiang. Discovering frequent patterns from uncertain data streams with time-fading and landmark models. *T. Large-Scale Data- and Knowledge-Centered Systems*, 8:174–196, 2013.
10. Hong Liu, Ashwin Kumar T. K, and Johnson P. Thomas. Cleaning framework for big data - object identification and linkage. In *Proc. of the Big Data Congress*, pages 215–221, 2015.
11. Arturas Mazeika and Michael H. Böhlen. Cleansing databases of misspelled proper nouns. In *Proc. of the VLDB Workshop on Clean Databases*, 2006.
12. Leonardo Andrade Ribeiro, Alfredo Cuzzocrea, Karen Aline Alves Bezerra, and Ben Hur Bahia do Nascimento. Incorporating clustering into set similarity join algorithms: The sjclust framework. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part I*, pages 185–204, 2016.
13. Leonardo Andrade Ribeiro, Alfredo Cuzzocrea, Karen Aline Alves Bezerra, and Ben Hur Bahia do Nascimento. Sjclust: Towards a framework for integrating similarity join algorithms and clustering. In *ICEIS 2016 - Proceedings of the 18th International Conference on Enterprise Information Systems, Volume 1, Rome, Italy, April 25-28, 2016*, pages 75–80, 2016.
14. Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62–78, 2011.
15. Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *Proc. of the SIGMOD Conference*, pages 743–754, 2004.
16. Natália Cristina Schneider, Leonardo Andrade Ribeiro, Andrei de Souza Inácio, Harley Michel Wagner, and Aldo von Wangenheim. SimDataMapper: An architectural pattern to integrate declarative similarity matching into database applications. In *Proc. of the SBBD Conference*, pages 967–972, 2015.
17. Christiane Faleiro Sidney, Diego Sarmento Mendes, Leonardo Andrade Ribeiro, and Theo Härder. Performance prediction for set similarity joins. In *Proc. of the SAC Conference*, pages 967–972, 2015.
18. Nan Tang. Big RDF data cleaning. In *Proc. of the ICDE Conference Workshops*, pages 77–79, 2015.
19. Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proc. of the SIGMOD Conference*, pages 85–96, 2012.
20. Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, 2011.
21. Feng Zhang, Hui-Feng Xue, Dong-Sheng Xu, Yong-Heng Zhang, and Fei You. Big data cleaning algorithms in cloud computing. *iJOE*, 9(3):77–81, 2013.