Adaptive Windows for Duplicate Detection

Uwe Draisbach #1, Felix Naumann #2, Sascha Szott *3, Oliver Wonneberg +4

#Hasso-Plattner-Institute, Potsdam, Germany
[1]uwe.draisbach@hpi.uni-potsdam.de
[2]felix.naumann@hpi.uni-potsdam.de

*Zuse Institute, Berlin, Germany
[3]szott@zib.de

+R. Lindner GmbH & Co. KG, Berlin, Germany
[4]owonneberg@lindner-esskultur.de

***Abstract*—Duplicate detection is the task of identifying all groups of records within a data set that represent the same real-world entity, respectively. This task is difficult, because (i) representations might differ slightly, so some similarity measure must be defined to compare pairs of records and (ii) data sets might have a high volume making a pair-wise comparison of all records infeasible. To tackle the second problem, many algorithms have been suggested that partition the data set and compare all record pairs only within each partition. One well-known such approach is the Sorted Neighborhood Method (SNM), which sorts the data according to some key and then advances a window over the data comparing only records that appear within the same window.**

We propose with the Duplicate Count Strategy (DCS) a variation of SNM that uses a varying window size. It is based on the intuition that there might be regions of high similarity suggesting a larger window size and regions of lower similarity suggesting a smaller window size. Next to the basic variant of DCS, we also propose and thoroughly evaluate a variant called DCS++ which is provably better than the original SNM in terms of efficiency (same results with fewer comparisons).

I. Motivation

Duplicate detection, also known as entity matching or record linkage was first defined by Newcombe et al. [1] and has been a research topic for several decades. The challenge is to effectively and efficiently identify pairs of records that represent the same real world object. The basic problem of duplicate detection has been studied under various further names, such as object matching, record linkage, merge/purge, or record reconciliation.

With many businesses, research projects, and government organizations collecting enormous amounts of data, it becomes critical to identify the represented set of distinct real-world entities. Entities of interest include individuals, companies, geographic regions, or households [2]. The impact of duplicates within a data set is manifold: customers are contacted multiple times, revenue per customer cannot be identified correctly, inventory levels are incorrect, credit ratings are miscalculated, etc. [3].

The challenges in the duplicate detection process are the huge amounts of data and that finding duplicates is resource intensive [4]. In a naive approach, the number of pairwise comparisons is quadratic in the number of records. Thus, it is necessary to make intelligent guesses which records have a high probability of representing the same real-world entity. These guesses are often expressed as partitionings of the data in the hope that duplicate records appear only within individual partitions. Thus, the search space can be reduced with the drawback that some duplicates might be missed.

Two important approaches for reducing the search space are *blocking* and *windowing*[1], which are evaluated and compared in [6]. The most prominent representative for windowing is the Sorted Neighborhood Method (SNM) by Hernández and Stolfo [7], [8]. SNM has three phases, illustrated in Fig. 1:

1) Key assignment: In this phase a sorting key is assigned to each record. Keys are usually generated by concatenating certain parts of attribute values (e.g., `first 3 letters of last name | first 2 digits of zip code`) in the hope that duplicates are assigned similar sorting keys and are thus close after the sorting phase. Sorting keys are not necessarily unique.
2) Sorting: All records are sorted by the sorting key.
3) Windowing: A fixed-size window slides over the sorted data. All pairs of records within a window are compared and duplicates are marked.

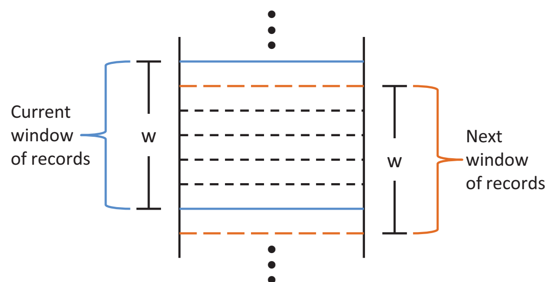


Fig. 1. Illustration of the Sorted Neighborhood method [7].

A disadvantage of this approach is that the window size is fixed and difficult to configure: If it is selected too small, some duplicates might be missed. On the other hand, a too large window results in many unnecessary comparisons. If effectiveness is most relevant, the ideal window size is equal to the size of the largest duplicate cluster in a data set. If a perfect sorting key exists, which sorts all records of the

[1]called “non-overlapping blocking” in [5].

duplicate clusters next to each other in the sort sequence, then all duplicates could be found. But even with the ideal window size, many unnecessary comparisons are executed, because not all clusters have that maximum size.

An example for this is the Cora Citation Matching data set[2], which comprises 1,879 references of research papers and is often used to evaluate duplicate detection methods [9], [10], [11]. In [12] we have described the definition of a gold standard for the Cora data set and other data sets. In Cora there are 118 clusters with at least 2 records. The histogram in Fig. 2 shows on the x-axis the clusters sorted by their size and on the y-axis the corresponding cluster size. As we can see, there are a few clusters with more than 100 records, but most groups have less than 50 records. Clusters with different sizes are quite common for deduplication [13] and also agree with our experience with industry partners [14].

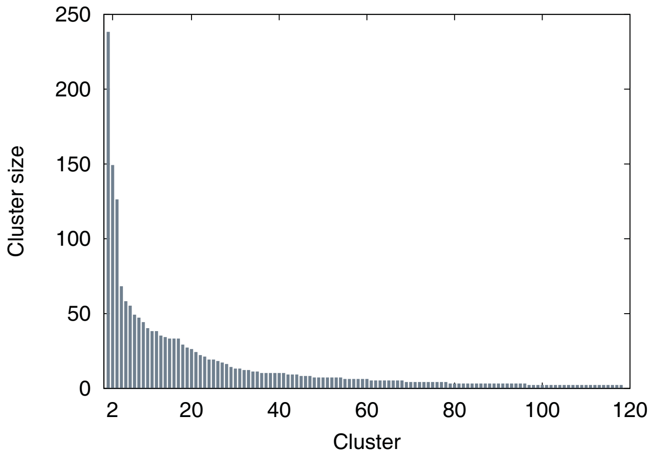


Fig. 2. Number of records per cluster in the Cora data set.

In this paper[3] we discuss the Duplicate Count strategy that adapts the window size to increase the efficiency of the duplicate detection process without reducing the effectiveness. In the following Sec. II we discuss related work. The Duplicate Count strategy and a variant called DCS++ is described and elaborated in more detail in Sec. III. An experimental evaluation is presented in Sec. IV and finally we conclude and discuss our future work in Sec. V.

II. Related Work

Several variations of the Sorted Neighborhood Method (SNM) have been proposed. As the result is highly depending on the used sorting key, multi-pass variants with multiple keys and a finally calculated transitive closure can help to improve the accuracy [3]. Monge and Elkan [13] adopt the SNM and propose the union-find data structure that defines a representative for each detected duplicate group. Records are first compared to the representatives and only if the similarity is high enough, they are compared with the other members of that cluster.

Yan et al. [16] discuss adaptivity of record linkage algorithms using the example of SNM. They use the window to build non-overlapping blocks that can contain different numbers of records. The pairwise record comparison then takes place within these blocks. The hypothesis is that the distance between a record and its successors in the sort sequence is monotonically increasing in a small neighborhood, although the sorting is done lexicographically and not by distance. They present two algorithms and compare them with the basic SNM. *Incrementally Adaptive-SNM* (IA-SNM) is an algorithm that incrementally increases the window size as long as the distance of the first and the last element in the current window is smaller than a specified threshold. The increase of the window size depends on the current window size. *Accumulative Adaptive-SNM* (AA-SNM) on the other hand creates windows with one overlapping record. By considering transitivity, multiple adjacent windows can then be grouped into one block, if the last record of a window is a potential duplicate of the last record in the next adjacent window. After the enlargement of the windows both algorithms have a retrenchment phase, in which the window is decreased until all records within the block are potential duplicates. We have implemented both IA-SNM and AA-SNM, and compare them to our work in our experimental evaluation. However, our experiments do not confirm that IA-SNM and AA-SNM perform better than SNM.

"Blocking" is an umbrella term for approaches that reduce the search space for duplicate detection. The idea is to partition the set of records into blocks and then compare all records only within these blocks, assuming that records in different blocks are unlikely to represent the same entity [17]. Thus, the overall number of comparisons depends on the number and the sizes of the blocks. Köpcke and Rahm divide these approaches into *disjoint* and *overlapping* blocking methods [5]. The disjoint blocking methods use a blocking predicate (e.g., the zip code for person records) to create mutually exclusive blocks, whereas overlapping blocking methods create overlapping blocks of records. Examples for overlapping blocking methods are SNM, canopy clustering, suffix array-based blocking, and Q-gram based indexing, surveyed in [18]. A generalization of standard blocking and the Sorted Neighborhood method is presented in [6].

Whang et al. [19] propose an iterative blocking model in which they use multiple blocking criteria at the same time to build overlapping blocks. The detected duplicates are then distributed to other blocks which can help to find additional duplicates and reduces the processing time for the other blocks. They propose two algorithms: Lego and Duplo. While Lego assumes that blocks are not stored on the disk and is therefore not applicable for data sets with millions of records, Duplo uses a disk-based iterative approach that can handle huge data sets. The concept of using the knowledge about already detected duplicates to save comparisons is also an essential part of our algorithm DCS++. However, in contrast

[2] http://www.cs.umass.edu/~mccallum/code-data.html
[3] An extended version of this paper is available [15].

to iterative blocking, our algorithm does not include a merging step.

The paper of Benjelloun et al. [20] defines the ICAR properties (idempotence, commutativity, associativity, and representativity) for match and merge functions in the duplicate detection process. Idempotence means that a record matches itself, whereas commutativity describes whether the order of the records has an impact on the matching result. We assume that the matching functions used with our algorithm fulfill these two properties. We do not have to consider associativity and representativity, because these are properties of the merge function and our algorithm does not merge records. However, Benjelloun et al. do not assume that the match function is transitive (i.e., $r_1 \approx r_2$ and $r_2 \approx r_3$ does not imply $r_1 \approx r_3$), whereas transitivity is a key aspect of our algorithm DCS++. They propose three algorithms: G-Swoosh is expensive, but can be used if the ICAR properties do not hold. R-Swoosh exploits the ICAR properties to reduce the number of comparisons. Finally, F-Swoosh also exploits the four properties and additionally avoids repeated feature comparisons. This last feature is irrelevant for our experimental setting; we include R-Swoosh in our evaluation.

For algorithms that rely on the sorting order of the records, the choice of a good sorting key is essential. It should be distinct enough, that the result of the duplicate detection process is not affected, e.g., for SNM the number of records with the same sorting key should not be greater than the window size. Furthermore, attributes that are less likely to contain erroneous values should be used, especially for the first few characters of the sorting key, as they are more important for the sorting order than the last few [3]. For our experimental evaluation, we used the same sorting key for all evaluated approaches.

III. Duplicate count strategy

The Duplicate Count strategy (DCS) is based on the Sorted Neighborhood Method (SNM) and varies the window size based on the number of identified duplicates. Due to the increase and decrease of the window size, the set of compared records differs from the original SNM. Adapting the window size does not inevitably result in additional comparisons; it can also reduce the number of comparisons. However, adapting the window size should result in an overall higher effectiveness for a given efficiency or in a higher efficiency for a given effectiveness.

DCS uses the number of already classified duplicates as an indicator for the window size: The more duplicates of a record are found within a window, the larger is the window. On the other hand, if no duplicate of a record within its neighborhood is found, then we assume that there are no duplicates or the duplicates are very far away in the sorting order. Each record t_i is once the first record of a window. In the beginning, we have a starting window size w, which is, as for SNM, domain-dependent. In the first step, record t_i is compared with $w - 1$ successors. So the current window can be described as $W(i, i + w - 1)$. If no duplicate can be found within this window, we do not increase the window. But if there is at least one duplicate, then we start increasing the window.

A. Basic strategy

The basic strategy increases the window size by one record. Let d be the number of detected duplicates within a window, c the number of comparisons and ϕ a threshold with $0 < \phi \leq 1$. Then we increase the window size as long as $\frac{d}{c} \geq \phi$. Thus, the threshold defines the average number of detected duplicates per comparison. The pseudocode of this variant can be found in Algorithm 1.

Algorithm 1 DCS (*records*, sorting key *key*, initial window size w, threshold ϕ)

Require: $w > 1$ and $0 < \phi \leq 1$

```
1.  sort records by key
2.  populate window win with first w records of records
4.  /*iterate over all rec. and search for
    duplicates*/
5.  for j = 1 to records.length − 1 do
10.   numDuplicates ← 0 /*number of det. duplicates*/
11.   numComparisons ← 0 /*number of comparisons*/
12.   k ← 2
13.   /*iterate over win to find dup. of rec. win[1]*/
14.   while k ≤ win.length do
15.     /*check if record pair is a duplicate*/
16.     if isDuplicate(win[1], win[k]) then
17.       emit duplicate pair (win[1], win[k])
18.       numDuplicates ← numDuplicates + 1
19.     end if
28.     numComparisons ← numComparisons + 1
29.     /*potentially increase window size by 1*/
30.     if k = win.length and j + k < records.length
        and (numDuplicates/numComparisons) ≥ φ then
31.       win.add(records[j + k + 1])
32.     end if
33.     k ← k + 1
34.   end while
36.   /*slide window*/
37.   win.remove(1)
38.   if win.length < w and j + k < records.length then
39.     win.add(records[j + k + 1])
40.   else /*trim window to size w*/
41.     while win.length > w do
42.       /*remove last record from win*/
43.       win.remove(win.length)
44.     end while
45.   end if
46.   j ← j + 1
47. end for
48. calculate transitive closure
```

B. Multiple record increase

The multiple record increase variant, dubbed DCS++, is an improvement of the basic strategy. It is based on the assumption that we have a perfect similarity measure (all record pairs are classified correctly as duplicate or non-duplicate; we show the performance of our algorithm with non-perfect similarity measures in Sec. IV-C). Instead of increasing the window by just one record, we add for each detected duplicate the next $w-1$ adjacent records of that duplicate to the window, even if the average is then lower than the threshold ϕ. Of course, records are added only once to that window. We can then calculate the transitive closure to save some of the comparisons: Let us assume that the pairs $\langle t_i, t_k \rangle$ and $\langle t_i, t_l \rangle$ are duplicates, with $i < k < l$. Calculating the transitive closure returns the additional duplicate pair $\langle t_k, t_l \rangle$. Hence, we do not need to check the window $W(k, k+w-1)$; this window is skipped. Algorithm 2 shows the differences of this variant compared to Algorithm 1. The differences are the performed check, whether a record should be skipped, and the handling of a duplicate.

Algorithm 2 DCS++ ($records$, key, w, ϕ)

```
3.  skipRecords ← null /*records to be skipped*/
4.  /*iterate over all rec. and search for
    duplicates*/
5.  for j = 1 to records.length − 1 do
6.    if win[1] NOT IN skipRecords then
        ... see Algorithm 1
13.     /*iterate over win to find dup. of rec.
        win[1]*/
14.     while k ≤ win.length do
15.       /*check if record pair is a duplicate*/
16.       if isDuplicate(win[1], win[k]) then
17.         emit duplicate pair (win[1], win[k])
18.         skipRecords.add(win[k])
19.         numDuplicates ← numDuplicates + 1
20.         /*increase window size from k by w-1
            records*/
21.         while win.length < k + w − 1
            and j + win.length < records.length do
22.           win.add(records[j + win.length + 1])
23.         end while
24.       end if
25.       ... see Algorithm 1
33.       k ← k + 1
34.     end while
35.   end if
      ... see Algorithm 1
47. end for
48. calculate transitive closure
```

1) Selection of the threshold: If we do not check the window $W(k, k+w-1)$, we might miss some duplicates within this window, if $W(k, k+w-1)$ contains records in addition to those in the window in which t_k was classified as duplicate. In Fig. 3, record t_k was classified as duplicate of t_i. The window of t_i ends with t_j. Let us assume that t_l is also a duplicate of t_i and t_k. If $l \leq j$ (case 1 in Fig. 3), then t_l is detected as duplicate, even if the window of t_k is not considered. On the other hand, if $l > j$ (case 2), we would not classify t_l as duplicate, due to the assumption that we do not have to create the window of t_k. We show that with the right selection of the threshold this case cannot happen.

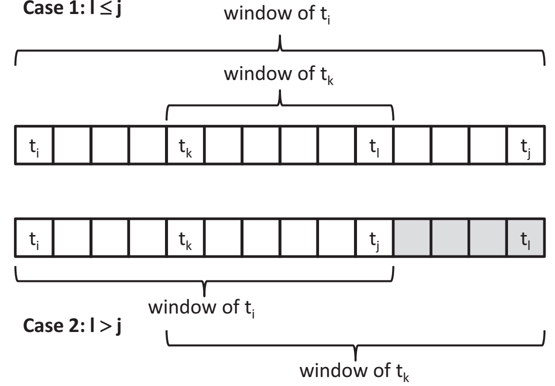


Fig. 3. Illustration of the two cases $l \leq j$ and $l > j$ that have to be considered for the selection of the threshold.

Proposition 3.1: With a threshold value $\phi \leq \frac{1}{w-1}$ no duplicates are missed due to skipping windows.

Proof: We first show that the increase by multiple records cannot cause one window to outrange the other one. Then we show that with $\phi \leq \frac{1}{w-1}$ the skipped windows do not contain additional records, i.e., the window of t_k cannot outrange the window of t_i.

(i) When t_k is detected to be a duplicate of t_i, the window of t_i is increased from t_k by $w-1$ records and thus contains the same records as the beginning window of t_k. Every time a new duplicate is detected, both windows are increased by $w-1$ records from that duplicate.

(ii) Windows are no longer increased, if $\frac{d}{c} < \phi$. Let f be the number of already detected duplicates in window $W(i,k)$, with $f \geq 1$ because at least t_k is a duplicate of t_i, and $k-i$ as the number of comparisons. To ensure $j \geq l$ we need:

$$\frac{f+d}{(k-i)+c} \geq \phi > \frac{d}{c} \tag{1}$$

Due to the assumption of a perfect similarity measure, d is the same for both windows. From (1) we can infer:

$$f + d \geq \phi \cdot (k-i) + \phi \cdot c \tag{2}$$

$$\text{and} \qquad \phi \cdot c > d \tag{3}$$

Inserting (3) in (2) results in:

$$\phi \leq \frac{f}{k-i} \tag{4}$$

Now we want to show which value we have to choose for ϕ, so that (4) is valid for all windows $W(i,k)$. The highest possible value for k is $k = f \cdot (w-1) + i$, which means that all

previously detected duplicates were the last of the respective window. Thus, we have:

$$\phi \le \frac{f}{k-i} = \frac{f}{f\cdot(w-1)+i-i} = \frac{1}{w-1} \tag{5}$$

■

We have shown that if the threshold value is selected $\phi \le \frac{1}{w-1}$, all windows W_i comprise at least all records of a window W_k where t_k is a duplicate of t_i. So leaving out window W_k does not miss a duplicate and thus does not decrease the recall.

2) DCS++ is more efficient than Sorted Neighborhood: In this section we show that DCS++ is at least as efficient as the Sorted Neighborhood Method. Let b be the difference of comparisons between both methods. We have $b > 0$ if DCS++ has more comparisons, $b = 0$ if it has the same number of comparisons, and $b < 0$ if it has fewer comparisons than SNM. Per detected duplicate, our method saves between 0 and $w-2$ comparisons.

To compare DCS++ with SNM, we have to examine the additional comparisons due to the window size increase and the saved comparisons due to skipped windows. Figure 4 shows the initial situation. In window W_i, we have d detected duplicates and it is increased up to t_j. The number of comparisons within $W(i,j)$ is $c = j-i$. In any case, we have the comparisons within the beginning window of t_i. The number of additional comparisons compared to SNM can be defined as $a = j-i-(w-1)$. With s as the number of saved comparisons, because we do not create windows for the duplicates, we have $s = d\cdot(w-1)$. We show that $a - s \le 0$.

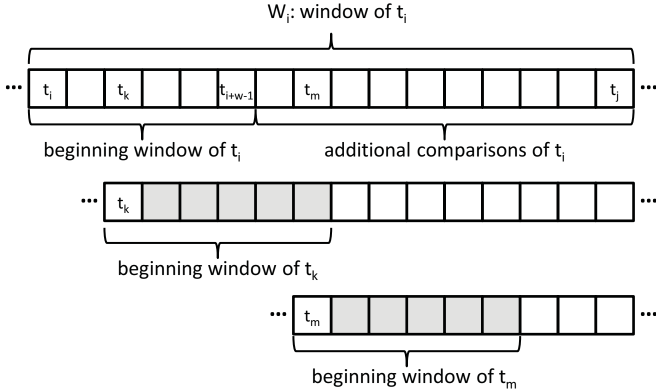


Fig. 4. Initial situation

Proposition 3.2: With a threshold value $\phi \le \frac{1}{w-1}$ DCS++ is at least as efficient as SNM with an equivalent window size ($w_{\text{SNM}} = w_{\text{DCS++}}$).

Proof: We have to distinguish two cases: In case (i) the beginning window of t_i contains no duplicate and in case (ii) it contains at least one duplicate.

(i) If there is no duplicate of t_i within the beginning window $W(i, i+w-1)$, then we have no additional comparisons due to a window size increase, but we also do not save any comparisons due to skipping windows. It therefore holds:

$$b = a - s = 0 - 0 = 0 \tag{6}$$

(ii) In the second case we have $d > 1$ duplicates within the beginning window. Then it holds:

$$\begin{align} b &= a - s \tag{7}\\ &= [j-i-(w-1)] - [d\cdot(w-1)] \tag{8}\\ &= j-i-(d+1)\cdot(w-1) \tag{9}\end{align}$$

As the window size is increased until $\frac{d}{c} < \phi$ and the last record is not a duplicate, we need with $\phi \le \frac{1}{w-1}$ at least $c = d\cdot(w-1)+1$ comparisons to stop the window increase.

In the most unfavorable case (see Fig. 5), we find in the last comparison the duplicate t_k and therefore increase the window by $w-1$ additional records. Then for $W(i,k)$ we have $\frac{d}{d\cdot(w-1)} = \phi$ and for $W(i,k+w-1)$ we have $\frac{d}{c} = \frac{d}{d\cdot(w-1)+(w-1)}$ and thus $c = d\cdot(w-1)+(w-1)$. We then have for $c = j-i$:

$$\begin{align} b &= j-i & &-(d+1)\cdot(w-1) \tag{10}\\ &= d\cdot(w-1)+(w-1) & &-(d+1)\cdot(w-1) \tag{11}\\ &= (d+1)\cdot(w-1) & &-(d+1)\cdot(w-1) \tag{12}\\ &= 0 \tag{13}\end{align}$$

So in this case we have $b = 0$, which means the same number of comparisons as the SNM with a window size of w.

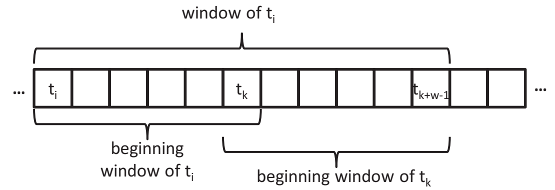


Fig. 5. Unfavorable case. Duplicate t_k is the last record in the window of t_i. Thus, due to the window increase there are $w-1$ additional comparisons.

We now show that for all other cases $b > 0$. In these cases, we have fewer than $w-1$ comparisons after $\frac{d}{c}$ falls under the threshold. The best is, when there is just a single comparison (see Fig. 6).

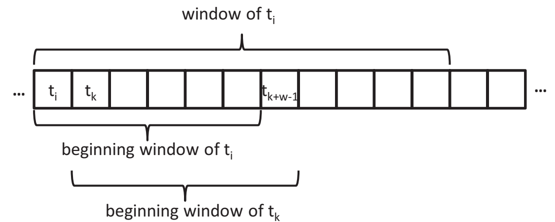


Fig. 6. Favorable case. Duplicate t_k is the first record next to t_i. Thus, due to the window increase there is just one additional comparison.

It holds for the number of comparisons c:

$$d\cdot(w-1)+1 \quad \le \quad c \quad \le \quad d\cdot(w-1)+(w-1) \tag{14}$$

So in the most favorable case $c = j-i = d\cdot(w-1)+1$ we

have:

$$b = j - i - (d+1) \cdot (w-1) \quad (15)$$
$$= d \cdot (w-1) + 1 - (d+1) \cdot (w-1) \quad (16)$$
$$= 1 - (w-1) \quad (17)$$
$$= 2 - w \quad (18)$$

As window size w is at least 2, we have $b \leq 0$. So in comparison to SNM we find the same number of duplicates but can save up to $w-2$ comparisons per duplicate. ■

Thus, we have shown that DCS++ with $\phi \leq \frac{1}{w-1}$ needs in the worst case the same number of comparisons and in the best case saves $w-2$ comparisons per duplicate compared to the Sorted Neighborhood Method.

IV. Experimental Evaluation

In this section, we evaluate the Duplicate Count strategy. Sec. IV-A describes the data sets and experiment settings and Section IV-B presents the results.

A. Data sets and Configuration

The experiments were executed with the DuDe toolkit [12], which is implemented in Java. To calculate the transitive closure, we use Warshall's algorithm [21]; additional duplicate pairs created by the transitive closure do not count as comparison, because for these pairs no comparison function is executed. Our evaluation is primarily based on the number of comparisons, because complex similarity measures are the main cost driver for entity resolution. The transitive closure is calculated for both, the Duplicate Count strategies (DCS and DCS++) and the Sorted Neighborhood Method (SNM). As we show later, the costs for the transitive closure depend on the number of duplicate pairs and hardly differ for the different algorithms.

To evaluate duplicate detection results, a variety of evaluation metrics exists [22], [23]. As we want to evaluate algorithms that select candidate pairs, we do not use a similarity function in Sec. IV-B. Instead, we assume a perfect similarity function by using a look-up in the gold standard to decide whether a record pair is a duplicate or not. Thus, all candidate pairs are classified correctly as duplicate or non-duplicate. For the evaluation, we measure the recall (fraction of detected duplicate pairs and the overall number of existing duplicate pairs) in relation to the number of executed comparisons. As in real world scenarios the assumption of a perfect classifier does not hold, we examine in Sec. IV-C the effects of an imperfect classifier. We use precision (fraction of correctly detected duplicates and all detected duplicates) and recall as quality indicators for the used classifiers and the F-Measure (harmonic mean of precision and recall) as measure to compare the different algorithms.

We chose three data sets for the evaluation. The **Cora Citation Matching** data set has already been described in Sec. I and we use the attribute `newreference` (typically the concatenation of the first author's last name and the year of publication) as sorting key. The second data set was generated with the **Febrl data generator** [24] and contains personal data. Using the Zipf distribution, 30,000 duplicates were added. Figure 7 shows the distribution of cluster sizes within the Febrl data set. We use a complex sorting key, created of the first 3 letters of `culture`, and the first 2 letters of `title`, `social security ID`, `postcode`, `phone number`, `address`, `surname`, and `given name`, always without spaces.

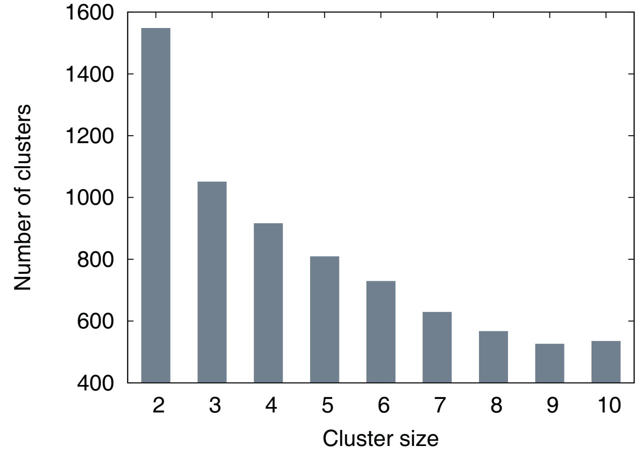


Fig. 7. Distribution of the cluster sizes for the Febrl data set

The third data set is artificially polluted real-world data and contains about 1 million records with **persons** and their addresses. It was created by an industry partner who uses this data set to evaluate duplicate detection methods and is thus a good benchmark. Our sorting key is the concatenation of the first three letters of the `zip code`, two letters of `street` and `last name`, and one letter of `street number`, `city`, and `first name`. Table I gives an overview of the three data sets.

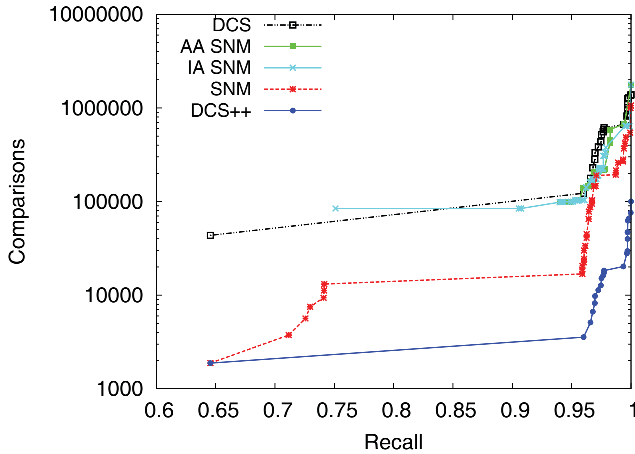
TABLE I
Overview evaluation data sets

Data set	Provenance	# of records	# of dupl. pairs
Cora	real-world	1,879	64,578
Febrl	synthetic	300,009	101,153
Persons	synthetic	1,039,776	89,784

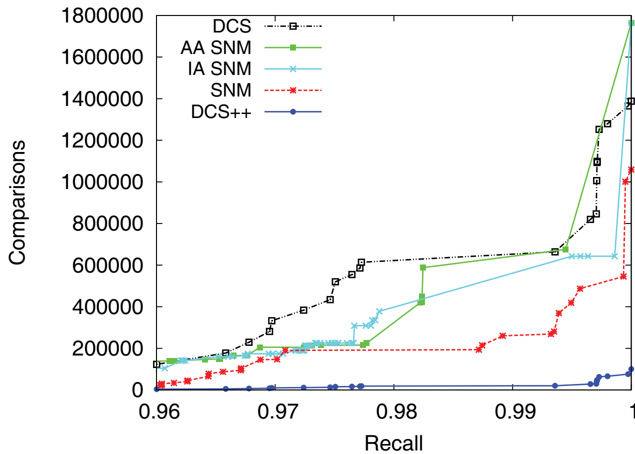
We compare the Duplicate Count strategies on the one hand with SNM and on the other hand with IA-SNM and AA-SNM. As window sizes we use values from 2–1000 for the Cora and the Febrl data set and values between 2–200 for the Persons data set. The threshold ϕ for the Duplicate Count strategies is $\frac{1}{w-1}$ as suggested in the previous section. IA-SNM and AA-SNM use the normalized Edit-Distance for creating the windows with thresholds from 0.1–1.0 for the Cora data set and 0.1–0.75 for the other two data sets. All algorithms use the same classifiers to decide, whether a record pair is a duplicate or not.

B. Experiment Results: Perfect Classifier

For the Cora data set, Fig. 8(a) shows the minimal number of required comparisons to gain the recall value on the x-axis. A comparison means the execution of a (probably complex) similarity function. Please note the logarithmic scale in opposite to Fig. 8(b), where we bring into focus the most relevant recall range from 96% – 100%. Both figures show the monotonic increase of SNM and the Duplicate Count strategies. The results of IA-SNM and AA-SNM are interpolated, which means that they show the minimum number of required comparisons to gain at least the specific recall. The most comparisons are needed for the IA-SNM and AA-SNM algorithms. We see that due to the window size increase, DCS performs worse than SNM. By contrast, DCS++ outperforms SNM, because it omits the creation of windows for already classified duplicates.



(a) Required comparisons (log scale).



(b) Required comparisons in the most relevant recall range.

Fig. 8. Results of a perfect classifier for the Cora data set. The figure shows the minimal number of required comparisons to gain the recall value on the x-axis.

Both SNM and DCS++ make use of calculating the transitive closure to find additional duplicates. Thus, some duplicates are selected as candidate pairs by the pair selection algorithm (e.g. SNM and DCS++) and then classified as duplicates, while other duplicates are detected when calculating the transitive closure later on. Figure 9 shows this origin of the duplicates. The x-axis shows the achieved recall value by summing detected duplicates of executed comparisons and those calculated by the transitive closure. With increasing window size SNM detects more and more duplicates by executing the similarity function, and thus has a decreasing number of duplicates detected by calculating the transitive closure. DCS++ on the other hand has hardly an increase of compared duplicates but makes better use of the transitive closure. As we show later, although there are differences in the origin of the detected duplicates, there are only slight differences in the costs for calculating the transitive closure.

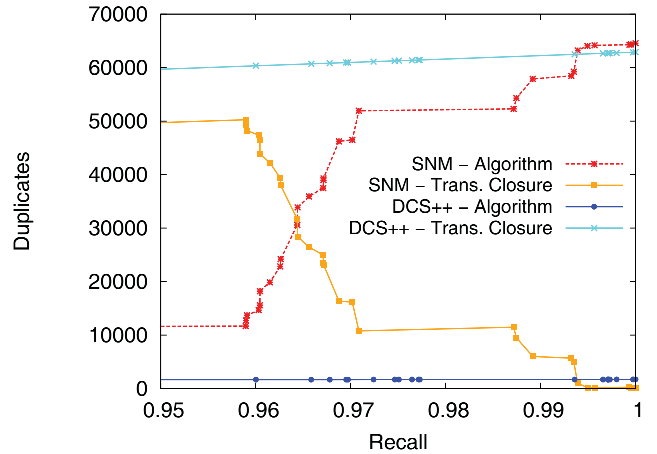


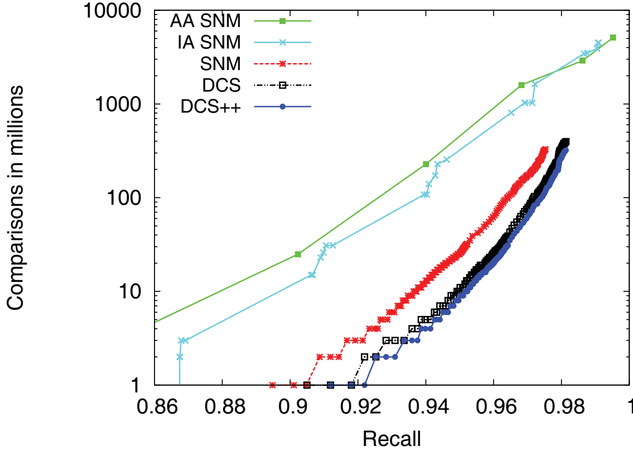
Fig. 9. Comparison of the origin of the detected duplicates for the Cora data set. The figure shows for the recall values on the x-axis the number of duplicates detected by the pair selection algorithm (SNM / DCS++) and the number of duplicates additionally calculated by the transitive closure.

The results for the Febrl data set (cf. Fig. 10(a)) are similar to the results of the Cora data set. Again, the IA-SNM and the AA-SNM algorithm require the most comparisons. But for this data set SNM requires more comparisons than DCS, whereas DCS++ still needs the fewest comparisons.

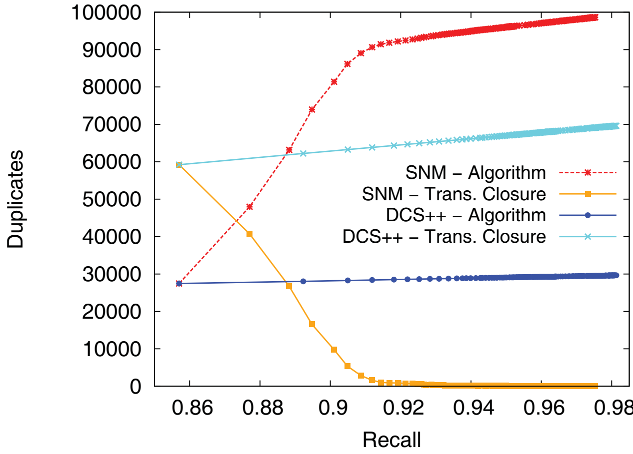
In Fig. 10(b) we can see again that SNM has to find most duplicates within the created windows to gain high recall values. DCS++ on the other hand finds most duplicates due to calculating the transitive closure. It is therefore important, to use an efficient algorithm that calculates the transitive closure.

In contrast to the Cora data set, the Persons data set has only clusters of two records. The Duplicate Count strategy is therefore not able to find additional duplicates by calculating the transitive closure. Fig. 11 shows that DCS++ nevertheless slightly outperforms SNM as explained in Sec. III-B2. The difference between DCS and SNM is very small, but the basic variant needs a few more comparisons.

We also evaluated the performance of the R-Swoosh algorithm [20]. R-Swoosh has no parameters, it merges records until there is just a single record for each real-world entity. The results of R-Swoosh are 345,273 comparisons for the Cora



(a) Minimal number of required comparisons (log scale) to gain the recall value on the x-axis.



(b) Comparison of the origin of the detected duplicates, i.e. whether the duplicate pairs are created by the pair selection algorithm or calculated by the transitive closure later on.

Fig. 10. Results of a perfect classifier for the Febrl data set.

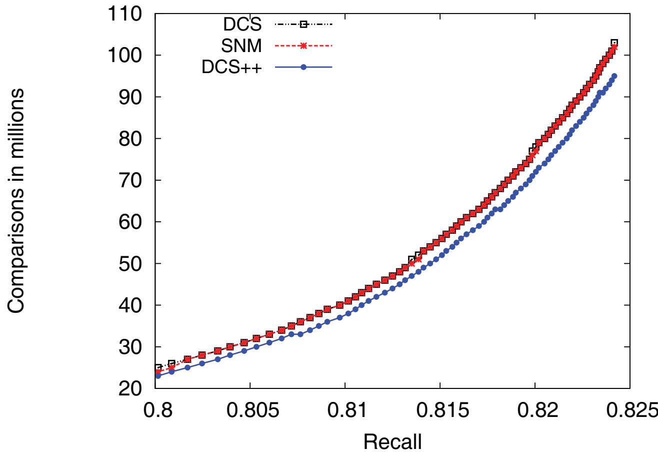


Fig. 11. Results of a perfect classifier for the Person data set. The figure shows the minimal number of required comparisons to gain the recall value on the x-axis.

data set, more than 57 billion comparisons for the Febrl data set, and more than 532 billion comparisons for the Person data set. So for the Cora data set with large clusters and therefore many merge operations, R-Swoosh shows a better performance than SNM or DCS, but it is worse than DCS++. For the Febrl and the Person data sets, R-Swoosh requires significantly more comparisons, but on the other hand returns a "perfect" result (recall is 1).

C. Experiment Results: Imperfect Classifier

So far we have assumed a perfect classifier, which is nearly impossible to develop in practice. In this section we analyze the effects of an imperfect classifier on the results of the DCS++ algorithm.

Figure 12 shows a sequence of sorted records. We first assume that all three labeled records t_i, t_j, and t_k are duplicates. Table II shows all possible combinations of how the classifier could classify these pairs, if the pairs were created as candidates by an algorithm. We further assume that t_j is within the initial window of t_i and t_k is within the initial window of t_j. Additionally, we assume for this example that if $\langle t_i, t_j \rangle$ is classified as duplicate, then the window of t_i is increased until it includes at least t_k. Otherwise, we have to distinguish the two cases whether t_k is included in the window of t_i or not.

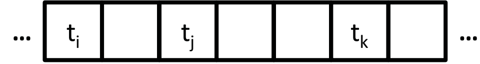


Fig. 12. Sorted records for evaluation of an imperfect classifier

For each combination, Table II describes the effect of the classification on the overall result. Each classified non-duplicate is a false negative. Please note that the stated classification refers to the result of the classifier. If the pair is not created, the classifier result is irrelevant; calculating the transitive closure can yet change the final classification of a record pair. Misclassification is not just a problem of the Duplicate Count Strategy, but also occurs with any other method.

If $\langle t_i, t_j \rangle$ is a duplicate, DCS++ does not create a window for t_j and therefore the classification of $\langle t_j, t_k \rangle$ does not depend on the classifier, but only on the calculation of the transitive closure.

In cases 5–8 of Tab. II $\langle t_i, t_j \rangle$ is classified as non-duplicate, and so there is no guarantee that the window of t_i is large enough to comprise t_k. But if t_k is included and $\langle t_j, t_k \rangle$ is classified correctly (case 5), then the transitive closure also includes $\langle t_i, t_j \rangle$ as a duplicate.

Compared to the Sorted Neighborhood Method (SNM), case 3 is especially interesting, because pair $\langle t_j, t_k \rangle$ is not created and therefore $\langle t_i, t_k \rangle$ is not detected to be a duplicate due to calculation of the transitive closure. SNM does not skip windows and would therefore classify $\langle t_j, t_k \rangle$ and due to the transitive closure also $\langle t_i, t_k \rangle$ correctly as duplicate. So for

TABLE II

ALL THREE PAIRS ARE DUPLICATES. THE TABLE SHOWS FOR THE DCS++ ALGORITHM THE NUMBER OF FALSE NEGATIVES (FN) IF PAIRS ARE MISCLASSIFIED AS NON-DUPLICATES.

No	Pair created / classifier result $\langle t_i, t_j \rangle$		$\langle t_i, t_k \rangle$		$\langle t_j, t_k \rangle$		FN	Explanation
1	y	D	y	D	n	D	0	All pairs classified correctly; pair $\langle t_j, t_k \rangle$ is not created but classified by the TC.
2	y	D	y	D	n	ND	0	All pairs classified correctly; pair $\langle t_j, t_k \rangle$ is not created but classified by the TC.
3	y	D	y	ND	n	D	2	As $\langle t_i, t_j \rangle$ is a duplicate, no window is created for t_j. Thus, $\langle t_j, t_k \rangle$ is not compared and therefore also misclassified as non-duplicate.
4	y	D	y	ND	n	ND	2	Only $\langle t_i, t_j \rangle$ is classified correctly as duplicate.
5	y	ND	y/n	D	y	D	0/2	If the window for t_i comprises t_k, then $\langle t_i, t_j \rangle$ is classified as duplicate by calc. the TC. Otherwise, both $\langle t_i, t_j \rangle$ and $\langle t_j, t_k \rangle$ are misclassified as non-duplicate.
6	y	ND	y/n	D	y	ND	2/3	If the window for t_i comprises t_k, then only $\langle t_i, t_j \rangle$ is classified as duplicate.
7	y	ND	y/n	ND	y	D	2	Only $\langle t_j, t_k \rangle$ is classified correctly as duplicate.
8	y	ND	y/n	ND	y	ND	3	All record pairs are misclassified as non-duplicate

TABLE III

ALL THREE PAIRS ARE NON-DUPLICATES. THE TABLE SHOWS FOR THE DCS++ ALGORITHM THE NUMBER OF FALSE POSITIVES (FP) IF PAIRS ARE MISCLASSIFIED AS DUPLICATES.

No	Pair created / classifier result $\langle t_i, t_j \rangle$		$\langle t_i, t_k \rangle$		$\langle t_j, t_k \rangle$		FP	Explanation
9	y	ND	y/n	ND	y	ND	0	All pairs classified correctly as non-duplicate.
10	y	ND	y/n	ND	y	D	1	Only $\langle t_j, t_k \rangle$ is misclassified as duplicate.
11	y	ND	y/n	D	y	ND	1/0	If the window for t_i comprises t_k, then only $\langle t_i, t_k \rangle$ is misclassified as duplicate.
12	y	ND	y/n	D	y	D	3/1	If the window for t_i comprises t_k, then $\langle t_i, t_j \rangle$ is misclassified by calculating the TC. Otherwise, only $\langle t_j, t_k \rangle$ is misclassified as non-duplicate.
13	y	D	y	ND	n	ND	1	Only $\langle t_i, t_j \rangle$ is misclassified
14	y	D	y	ND	n	D	1	As $\langle t_i, t_j \rangle$ is classified as duplicate, no window is created for t_j. Thus, $\langle t_j, t_k \rangle$ is not compared and therefore also correctly classified as non-duplicate.
15	y	D	y	D	n	ND	3	All pairs are misclassified; pair $\langle t_j, t_k \rangle$ is not created but classified by the TC.
16	y	D	y	D	n	D	3	All pairs are misclassified; pair $\langle t_j, t_k \rangle$ is not created but classified by the TC.

case 3, we have two false negatives for DCS++, but no false negatives for SNM.

Table III also refers to the records t_i, t_j, and t_k in Fig. 12, but we now assume that they are all non-duplicates. The results are similar to those in Table II, but this time classified duplicates are false positives. In cases 13–16, $\langle t_i, t_j \rangle$ is incorrectly classified as duplicate and thus, no window for t_j is created. The classification of $\langle t_j, t_k \rangle$ depends only on the transitive closure. This results in fewer false positives, compared to the SNM, in case 14 because SNM would compare $\langle t_j, t_k \rangle$ and thus misclassify it as duplicate. Additionally, due to the calculation of the transitive closure, also $\langle t_i, t_k \rangle$ would be misclassified as duplicate, resulting in three false positives for SNM opposed to one false positive for DCS++.

Based on these results we can say that a misclassification can but does not necessarily have a negative impact on the overall result. We now experimentally evaluate the effect of misclassification. The experiment uses different classifiers for the CORA data set. Classifiers can be very restrictive, which leads to a high precision, but a low recall value. Such a classifier that does not detect all real duplicates favors SNM, as described before in case 3. On the other hand, classifiers with a lower precision and hence a higher recall value favor DCS++, because misclassified non-duplicates are worse for SNM (see case 14). Thus, the results depend on the precision/recall tradeoff of the classifier and we therefore use the F-Measure (harmonic mean of precision and recall) in our experiments as quality indicator.

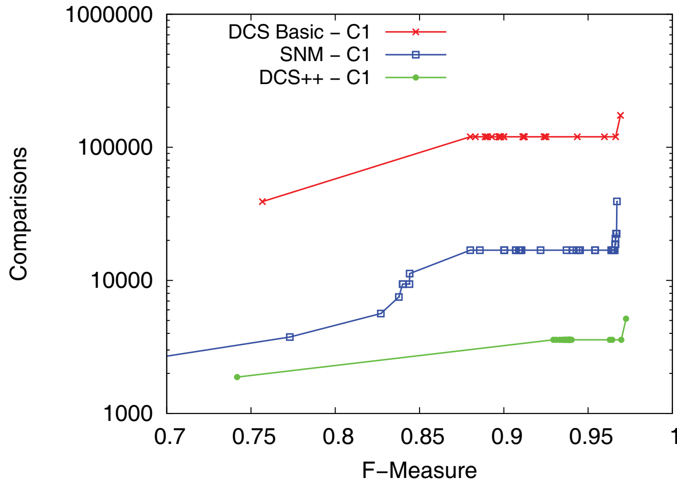
Table IV gives an overview of the used classifiers. The values are the results of an exhaustive comparison without calculating the transitive closure. We have selected one classifier with both a high precision and a high recall value (C1). The other two classifiers have either a high recall (C2) or high precision (C3) value.

TABLE IV

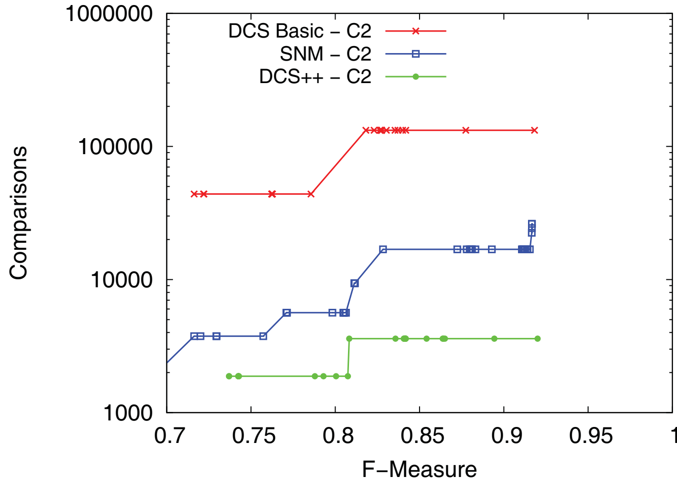
THREE CLASSIFIERS FOR THE CORA DATA SET. NUMBERS ARE BASED ON AN EXHAUSTIVE COMPARISON WITHOUT CALCULATING THE TRANSITIVE CLOSURE.

Classifier	Precision	Recall	F-Measure
C1	98.12 %	97.17 %	97.64 %
C2	83.27 %	99.16 %	90.52 %
C3	99.78 %	84.13 %	91.23 %

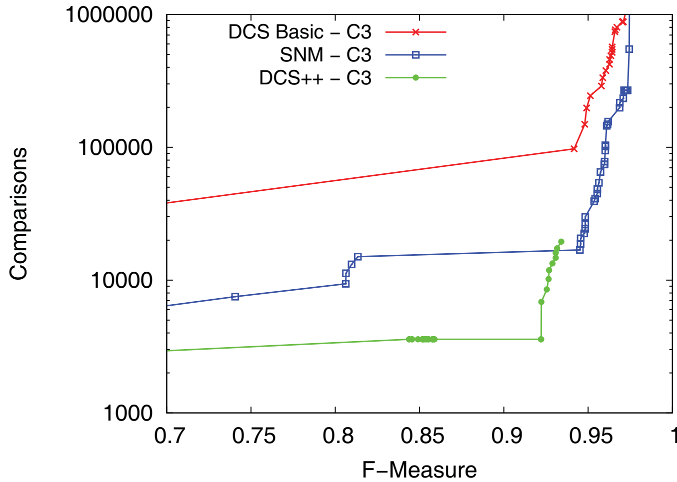
Figure 13 shows the interpolated results of our experiments – one chart for each of the classifiers. We see for all three classifiers that DCS requires the most and DCS++ the least number of comparisons, while SNM is in between. Figure 13(a) shows the results for classifier C1 with both a high recall and a high precision value. The best F-Measure value is nearly the same for all three algorithms and the same is true for classifier C2 with a high recall, but low precision value, as shown in Fig. 13(b). However, we can see that the F-Measure value for C2 is not as high as for classifiers C1 or C3. Classifier C3 with a high precision but low recall value shows a slightly lower F-Measure value for DCS++ than for DCS or SNM. This classifier shows the effect of case 3 from Table II. Due to skipping of windows, some duplicates are missed. However, the number of required comparisons is



(a) Required comparisons (log scale) classifier C1



(b) Required comparisons (log scale) classifier C2



(c) Required comparisons (log scale) classifier C3

Fig. 13. Interpolated results of the three imperfect classifiers C1-C3 for the Cora data set.

significantly lower than for the other two algorithms. Here, the DCS++ algorithm shows its full potential for classifiers that especially emphasize the recall value.

So far we have considered only the number of comparisons to evaluate the different algorithms. As described before, DCS++ and SNM differ in the number of detected duplicates by using a classifier and by calculating the transitive closure. We have measured the execution time for the three classifiers, divided into classification and transitive closure (see Fig. 14). As expected, the required time for the transitive closure is significantly lower than for the classification, which uses complex similarity measures. The time for the classification is proportional to the number of comparisons. All three classifiers require about 0.2 ms per comparison.

The time to calculate the transitive closure is nearly the same for all three algorithms and all three classifiers. SNM requires less time than DCS or DCS++, but the difference is less than 1 second. Please note that the proportion of time for classification and for calculating the transitive closure depends on the one hand on the data set size (more records lead to more comparisons of the classifier) and on the other hand on the number of detected duplicates (more duplicates require more time for calculating the transitive closure).

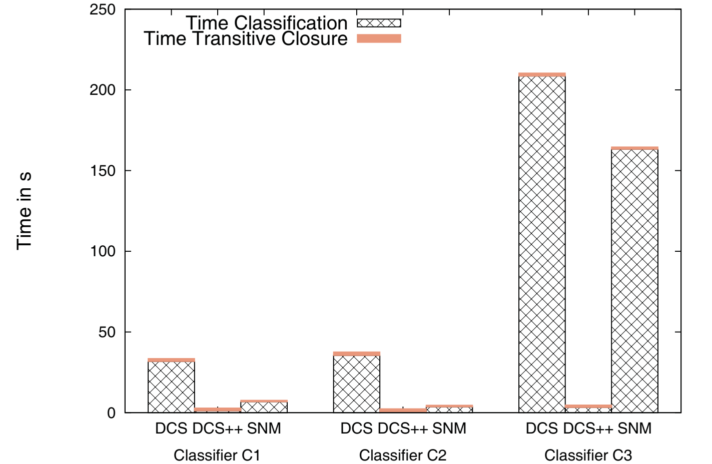


Fig. 14. Required Time for the best F-Measure result of each classifier.

V. Conclusion

With increasing data set sizes, efficient duplicate detection algorithms become more and more important. The Sorted Neighborhood method is a standard algorithm, but due to the fixed window size, it cannot efficiently respond to different cluster sizes within a data set. In this paper we have examined different strategies to adapt the window size, with the Duplicate Count Strategy as the best performing. In Sec. III-B2 we have proven that with a proper (domain- and data-independent!) threshold, DCS++ is more efficient than SNM without loss of effectiveness. Our experiments with real-world and synthetic data sets have validated this proof.

The DCS++ algorithm uses transitive dependencies to save complex comparisons and to find duplicates in larger clusters.

Thus, it is important to use an efficient algorithm to calculate the transitive closure. In contrast to previous works, we consider the costs of the transitive closure separately.

Overall, we believe that DCS++ is a good alternative to SNM. The experiments have shown the potential gains in efficiency, allowing to search for duplicates in very large data sets within reasonable time.

ACKNOWLEDGEMENT

This research was partly supported by the German Research Society (DFG grant no. NA 432).

REFERENCES

[1] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James, "Automatic linkage of vital records." *Science*, vol. 130, pp. 954–959, 1959.

[2] L. Gu and R. Baxter, "Adaptive filtering for efficient record linkage," in *Proceedings of the SIAM International Conference on Data Mining*, 2004, pp. 477–481.

[3] F. Naumann and M. Herschel, *An Introduction to Duplicate Detection (Synthesis Lectures on Data Management)*, 2010.

[4] H. H. Shahri and A. A. Barforush, "A flexible fuzzy expert system for fuzzy duplicate elimination in data cleaning," in *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, 2004, pp. 161–170.

[5] H. Köpcke and E. Rahm, "Frameworks for entity matching: A comparison," *Data & Knowledge Engineering (DKE)*, vol. 69, no. 2, pp. 197–210, 2010.

[6] U. Draisbach and F. Naumann, "A comparison and generalization of blocking and windowing algorithms for duplicate detection," in *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2009.

[7] M. A. Hernández and S. J. Stolfo, "The merge/purge problem for large databases," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995, pp. 127–138.

[8] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data Mining and Knowledge Discovery*, vol. 2(1), pp. 9–37, 1998.

[9] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2003, pp. 39–48.

[10] X. Dong, A. Halevy, and J. Madhavan, "Reference reconciliation in complex information spaces," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005, pp. 85–96.

[11] P. Singla and P. Domingos, "Object identification with attribute-mediated dependences," in *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, 2005, pp. 297–308.

[12] U. Draisbach and F. Naumann, "DuDe: The duplicate detection toolkit," in *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2010.

[13] A. E. Monge and C. Elkan, "An efficient domain-independent algorithm for detecting approximately duplicate database records," in *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, 1997.

[14] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster, "Industry-scale duplicate detection," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1253–1264, 2008.

[15] U. Draisbach, F. Naumann, S. Szott, and O. Wonneberg, "Adaptive windows for duplicate detection," Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Tech. Rep. 49, 2011.

[16] S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles, "Adaptive sorted neighborhood methods for efficient record linkage," in *Proceedings of the ACM/IEEE-CS joint conference on Digital libraries (JCDL)*, 2007, pp. 185–194.

[17] R. Baxter, P. Christen, and T. Churches, "A comparison of fast blocking methods for record linkage," in *Proceedings of the ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003, pp. 25–27.

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

[19] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina, "Entity resolution with iterative blocking," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009, pp. 219–232.

[20] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, "Swoosh: a generic approach to entity resolution," *VLDB Journal*.

[21] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, pp. 11–12, January 1962.

[22] P. Christen and K. Goiser, "Quality and complexity measures for data linkage and deduplication." in *Quality Measures in Data Mining*, ser. Studies in Computational Intelligence, 2007, vol. 43, pp. 127–151.

[23] D. Menestrina, S. Whang, and H. Garcia-Molina, "Evaluating entity resolution results," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, pp. 208–219, 2010.

[24] P. Christen, "Probabilistic data generation for deduplication and data linkage," in *IDEAL, Springer LNCS 3578*, 2005, pp. 109–116.