A Generalization of Blocking and Windowing Algorithms for Duplicate Detection

Uwe Draisbach
Hasso Plattner Institute
14482 Potsdam, Germany
uwe.draisbach@hpi.uni-potsdam.de

Felix Naumann
Hasso Plattner Institute
14482 Potsdam, Germany
naumann@hpi.uni-potsdam.de

***Abstract*—Duplicate detection is the process of finding multiple records in a dataset that represent the same real-world entity. Due to the enormous costs of an exhaustive comparison, typical algorithms select only promising record pairs for comparison. Two competing approaches are *blocking* and *windowing*. Blocking methods partition records into disjoint subsets, while windowing methods, in particular the Sorted Neighborhood Method, slide a window over the sorted records and compare records only within the window. We present a new algorithm called *Sorted Blocks* in several variants, which generalizes both approaches. To evaluate Sorted Blocks, we have conducted extensive experiments with different datasets. These show that our new algorithm needs fewer comparisons to find the same number of duplicates.**

I. Duplicate Detection

Duplicate detection, also known as entity matching or record linkage, is the problem of identifying pairs of records that represent the same real-world entity [8], [13]. An exhaustive duplicate detection process involves computing the similarities of all record pairs, which can be very expensive for large datasets. Therefore, the challenge is to effectively and efficiently search for duplicates.

The performance bottleneck for duplicate detection is typically the expensive attribute comparison with similarity measures between record pairs [4]. To avoid these prohibitively expensive comparisons of all pairs of records, a common technique is to carefully partition the records into smaller subsets and search for duplicates only within these partitions. Two competing approaches are often cited: Blocking methods partition records into disjoint subsets, for instance using `zip_code` as partitioning key. Sorted-neighborhood-based methods sort the data according to some key, such as `last_name`, and then slide a window of fixed size across the sorted data and compare pairs only within the window.

We compare both approaches in Sec. II and present a new generalized algorithm in Sec. III. Please note that our intention is to generalize Blocking and Windowing; there exist further approaches that cluster records for duplicate detection. Finally, we evaluate the new method experimentally in Sec. IV using real-world and artificial datasets and conclude in Sec. V.

Note that this paper is an extended version of our workshop short-paper [6] (no formal proceedings). The extensions include new and improved variants of the original algorithm and several additional experiments on a broader scope of datasets.

II. Blocking and Windowing

In the following sections, we briefly introduce the blocking and windowing methods and then compare them.

A. Blocking

(Standard) Blocking algorithms use some blocking key to partition a set of records into disjoint partitions (blocks) [4], [8]. The comparison of record pairs is then limited to records within the same partition. Thus, the overall number of comparisons is greatly reduced [1], [2], [3].

An important decision for the blocking method is the choice of a good partitioning predicate, which determines the number and the size of the partitions. It should be chosen in a manner that potential duplicates are grouped in the same partition. E.g, for CRM applications a typical partitioning is by `zip_code` or by their first few digits. If two duplicate records have the same zip code, they appear in the same partition and thus can be recognized as duplicates. Other partitionings might be by `last_name` or some fixed-sized prefix of them, by `employer`, etc. The frequency distribution of the blocking keys influences the overall execution time, which is dominated by the largest blocks [4]. If the attributes used to create the key have erroneous values, the records of a duplicate pair might be assigned to different blocks and therefore cannot be classified as duplicate.

To detect duplicates that differ in the partitioning attribute, a multi-pass method is employed. Blocking methods perform multiple runs, each time with a different partitioning predicate, followed by a transitive closure over all discovered duplicate pairs. A new approach dealing with different blocking keys – iterative blocking – is presented in [15].

Christen compares six blocking methods [4]. Next to standard blocking with disjoint blocks, he also considers blocking techniques with overlapping blocks, such as q-gram based blocking, canopy clustering, or suffix array based blocking. Other blocking approaches, although called 'adaptive sorted neighborhood methods', are presented by Yan et al. [16]. They first sort the records and then create non-overlapping 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 hand creates windows with a single overlapping record. By considering transitivity, multiple adjacent windows can then be grouped to one block, if the last record of a window is a potential duplicate of the last record in the next adjacent window. Both algorithms have after the enlargement of the windows a retrenchment phase, in which the window is decreased until all records within the block are potential duplicates. We have re-implemented and evaluated both algorithms.

B. Windowing

Windowing methods are slightly more elaborate than blocking methods. In [9] and [10] the authors describe the Sorted Neighborhood Method (SNM), which is divided into three phases. First, a sorting key is assigned to each record. As for the blocking methods, the key does not have to be unique and can be generated by concatenating values (or substrings of values) from different attributes. In the second phase, all records are sorted according to that key. As in the blocking method, the assumption is that duplicates have similar keys and are thus close to each other after sorting. The first two phases are comparable to the selection of a partitioning predicate in the blocking method. The final phase of SNM slides a window of fixed size across the sorted list of records. All pairs of records that appear in the same window are compared. The size of the window (typically between 10 and 30) represents the trade-off between efficiency and effectiveness; larger windows yield longer runtimes but detect more duplicates. To reduce the number of comparisons and the overall execution time, the records can be clustered first which means that as for blocking, the records are assigned to disjoint clusters. The Sorted Neighborhood Method is then applied to each cluster in parallel.

A drawback of the Sorted Neighborhood Method is the fixed window size, especially for datasets with very different cluster sizes. If the window size is selected too small, some duplicates might be missed. On the other hand, if the window size is selected large enough to find all duplicates even for the largest cluster, then there are a lot of unnecessary comparisons in the area of the smaller clusters.

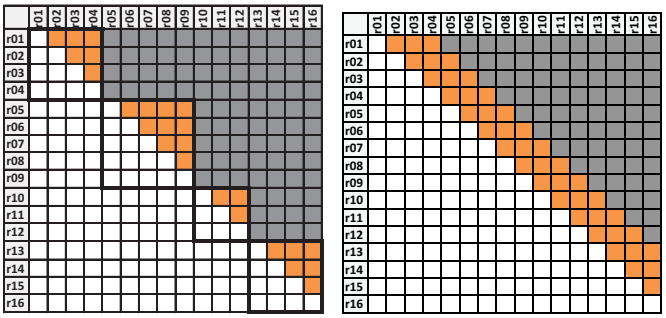
To avoid mis-sorts due to errors in the attributes that are used to generate the key, again, multi-pass variants of SNM produce multiple keys and perform the sorting and windowing multiple times. As with the blocking method, the transitive closure is finally calculated. Research has produced many variants of SNM, including one that avoids the choice of keys [12], and a variant for nested XML data [14].

C. Comparison of blocking and windowing approaches

Blocking and windowing have much in common: Both aim at reducing the number of comparisons by making intelligent guesses about which pairs of records have a chance of being duplicates. Both rely on some intrinsic orderings of the data and the assumption that records that are close to each other with respect to that order have a higher chance of being duplicates than other pairs of records [13].

Figure 1 shows the space of duplicate candidates for a database of 16 records. Each field $c_{i,j}$ in the matrix represents a comparison of the two corresponding candidates r_i and r_j. Assuming that the similarity measure is symmetric, i.e. $sim(c, c') = sim(c', c)$, the number of pairwise comparisons is $\frac{n \times (n-1)}{2}$. The diagonal fields $c_{i,i}$ need not be compared, nor do the fields in the lower, non-shaded part $c_{i,j}$ with $i > j$. Thus, $\frac{16(16-1)}{2}$ = 120 comparisons remain, opposed to 256 comparisons for the complete matrix.

In Fig. 1(a), we assume that records 1-16 are sorted by the partitioning key, both horizontally and vertically. The candidate pairs after partitioning are shaded. Clearly, the number of candidates is reduced, namely to only 25 comparisons in this made-up case. Similarly, Fig. 1(b) shows the candidate pairs for a Sorted Neighborhood algorithm with a window size $w = 4$, resulting in 42 comparisons.



(a) Blocking (b) Windowing

Fig. 1. Duplicate candidates for different algorithms

Although the two blocking and windowing algorithms in this example perform nearly the same number of comparisons, the sets of actual comparisons differ. For instance, records r_5 and r_9 are compared only by the blocking method, because they lie in the same first block. On the other hand, records r_{11} and r_{13} are compared only by the windowing method, because they are in the same window, but not in the same block. In [6], we show that both approaches can be adapted so that they encompass the same comparisons as the other approach. For windowing approaches this can be achieved by increasing the size of the window, while blocking approaches have to allow overlapping blocks to adapt windowing.

III. Sorted Blocks

Sorted Blocks is a generalization of blocking and windowing algorithms for duplicate detection. In this section, we first describe the basic algorithm as in [6] and then introduce two new variants.

A. Sorted Blocks

Sorted Blocks first sorts the records based on a sorting key. Like for the Sorted Neighborhood Method and the adaptive sorted neighborhood methods by Yan et al. [16], the assumption is that records close after sorting have a higher probability of being duplicates. But instead of sliding a fixed size window over all records, we create disjoint partitions and compare all records within these partitions.

It is desirable that the sorting keys are unique to obtain an unambiguous sorting order. To this end, more attributes can be included for sorting (e. g. `zip_code` and `name`) than for actually partitioning the data (e. g. only `zip_code`). Nevertheless, uniqueness is not strictly necessary; in case of a tie, we use the input order of the records.

To ensure that also such duplicates can be found that are close in the sorting order, but for any reason were assigned to different partitions, an additional partition overlap is used. This overlap is defined by a manually selected overlap parameter o. It describes the number of records in one partition to be compared with records of the adjacent partition. Within the overlap a fixed size window with size $o+1$ is slid across the sorted data and all records within the window are compared. In this way, the additional complexity of the overlap is linear. Note that this windowing technique is used only in the overlapping part; within a partition all record pairs are compared.

In [6], we used fixed size partitions in the experiments, slicing the entire sorted list into partitions independent of the attribute values of the records. A better approach is using a partition predicate to determine the partitions. The partition predicate should make use of the sorting key, e.g., using its first few characters. This adaptive partition size is advantageous in comparison to the fixed window size of the Sorted Neighborhood Method. A fixed partition size could result in missed duplicates, if the size is selected too small. On the other hand, if the size is selected large enough for the duplicate pair with the largest distance in the sorting order, many unnecessary comparisons are conducted in windows where the duplicates are closer to each other.

Figure 2 is an illustration of Sorted Blocks. The 14 records $r_1 - r_{14}$ are sorted based on a sorting key and then divided into four partitions $P_1 - P_4$ based on a partition predicate. Within each partition we perform a complete comparison of all record pairs as illustrated for P_1.

The overlap was selected as $o = 2$. So between P_1 and P_2 we have overlap O_{P_1,P_2} with $2 \times o = 4$ records ($r_3 - r_6$). Within this overlap are two windows $W_{(P_1,P_2).1}$ and $W_{(P_1,P_2).2}$, each comprising $o + 1 = 3$ records. The records within the windows are also compared pairwise. Of course, the algorithm compares each pair only once.

A special case arises, if a partition is smaller than the overlap. In this case, the windows can comprise more than two partitions, as illustrated for partition P_3. The only impact on the Sorted Blocks method is that in this case there are two identical windows (e.g., $W_{(P_2,P_3).2}$ and $W_{(P_3,P_4).1}$), which are folded in the implementation.

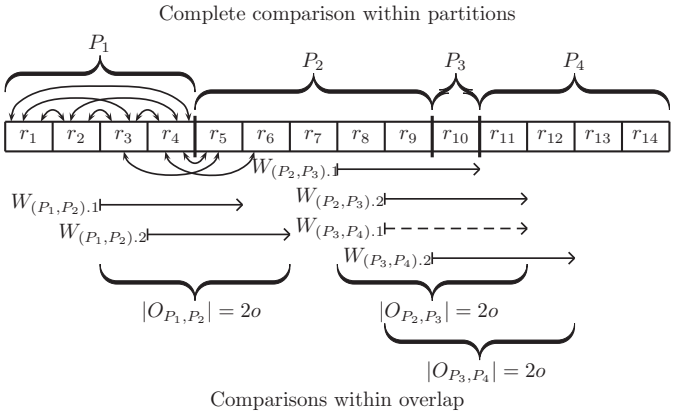


Fig. 2. Illustration Sorted Blocks

Algorithm 1 Sorted Blocks ($records$, key, overlap o)

```
1: sort records on key
2: /* initialization */
3: listComparisonRecords ← [] // List of records that are compared with the currently processed record
4: windowNr ← o+1 // Number of the window in the overlapping area
5: i ← 1
6: /* iterate over all records and search for duplicates */
7: while i ≤ records.length do
8:    if records[i] is 1st element of new partition and i > 1 then
9:       while listComparisonRecords.length > o do
10:         listComparisonRecords.remove[1]
11:      end while
12:      windowNr ← 1
13:   else if windowNr ≤ o then
14:      listComparisonRecords.remove[1]
15:      windowNr ← windowNr + 1
16:   end if
17:   /* compare current record with all records in listComparisonRecords */
18:   for j = 1 to listComparisonRecords.length do
19:      compare records[i] with listComparisonRecords[j]
20:   end for
24:   listComparisonRecords.append(records[i])
25:   i ← i + 1
26: end while
27: calculate transitive closure
```

Algorithm 1 shows the pseudo code for the basic Sorted Blocks approach. First, the records are sorted on the sorting key. Then we iterate over the sorted records and check for each record whether it is the first record of a new partition. This check is based on the sorting key, which as a whole or parts of it are used as partition predicate (e.g., records are sorted on `zip_code` and each new partition begins if the first digit of the current record is different to the first digit of the previous one). If the record is not the first record of a new partition, it is compared with all other records that are already in that partition and if applicable with records in the overlap of the previous partition. But if the record is the first element of a new partition, then all records of the previous partition that are not in the overlap are removed and the current record is

compared with the remaining records. After iterating over all records, the transitive closure is calculated.

The Sorted Blocks algorithm can be configured to create the same record pairs for comparison as either Blocking or the Sorted Neighborhood Method. If the overlap parameter is selected to be 0, we have the standard blocking algorithm. For the Sorted Neighborhood Method, the decision in line 8 whether the current record is the first of a new partition, has to be always true and additionally, the overlap parameter o needs to be equal to the window size w.

B. Sorted Blocks variants

The number of comparisons in the basic Sorted Blocks approach depends on the sizes of the partitions. As mentioned for blocking in Sec II-A, the overall execution time for Sorted Blocks is dominated by the largest blocks. To prevent too large partitions, we suggest two variants of Sorted Blocks, which have a maximum partition size. An example is a customer database in which we partition the records by area code to compare customers within the same city. The number of comparisons might be feasible for smaller cities, but would exceed the resources for cities like Berlin, Paris, or London with millions of citizens. An upper bound for the partition size can reduce the number of comparisons, but requires that the sorting key uses additional attributes so that there is a higher chance that possible duplicates are close together within the sorting order (e.g., area code as partition predicate, but area code and name as sorting key).

1) Sorted Blocks creating partition when max. partition size is reached: This first variant creates a new partition if the maximum partition size is reached. This means, that the new partition is created independently of the partitioning key. Although records have the same partition predicate, they are grouped in different partitions. But due to the overlap between the partitions, it is ensured that all records are compared with its predecessors and successors in the sorting order. To implement this variant, the Sorted Blocks algorithm just needs an additional condition in the If-statement, which is shown in Algorithm 2 (Algorithms 2 and 3 extend Algorithm 1 by replacing the corresponding code lines).

Algorithm 2 Sorted Blocks new partition($records$, key, $maxPartitionSize$)

8: **if** ($records[i]$ is 1st element of new partition **and** $i > 1$) **or** ($listComparisonRecords.length$ = $maxPartitionSize$) **then**
9: ... // code like in Algorithm 1
16: **end if**

2) Sorted Blocks using window when max. partition size is reached: This second variant uses the maximum partition size as window size to slide a window over the records within a partition. If the maximum number of records is reached, for each new record in the partition, the first element in the current window is removed. This iterative process runs until the end of the partition is reached. Thus, this variant is very similar to the Sorted Neighborhood Method. The only additional lines of code compared to the basic Sorted Blocks algorithm (Algorithm 1) are shown in Algorithm 3.

Algorithm 3 Sorted Blocks ($records$, key, $maxPartitionSize$)

21: **if** $listComparisonRecords.length$ = $maxPartitionSize$ **then**
22: $listComparisonRecords.remove[1]$
23: **end if**

IV. Experimental Evaluation

In this section we compare the Sorted Blocks algorithm and its variants with both Blocking and the Sorted Neighborhood Method and additionally with IA-SNM and AA-SNM from [16]. The experiments were conducted with our duplicate detection toolkit DuDe [7]. Next to the basic Sorted Blocks method from Section III-A and the two variants from Section III-B, we use also a variant with fixed partition sizes as presented in [6].

A. Experiment configuration and key figures

We used three datasets to evaluate the algorithms: The first dataset is a randomly selected sample from freeDB[1]. It contains information about **CDs** including artist, title, and songs. The sorting key is created by concatenating the first three letters of each the artist, the CD title, and the name of the first track.

The **restaurant dataset** is from the RIDDLE repository[2]. It comprises names and addresses of restaurants from two restaurant guides. As sorting key, we use the concatenation of the first three letters of the restaurant name and the first two letters of the city. Both the CD and the restaurant dataset are available on our web page[3].

The third dataset is artificially polluted real-world data and contains about 1 Mio. records with **persons**. It was created by an industry partner who uses this dataset to evaluate duplicate detection methods and is thus a good benchmark. The sorting key is again the concatenation of several attribute values. For this dataset we use 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 used datasets.

Dataset	Nr. records	Nr. dup. pairs
CD (real-world)	9,763	299
Restaurant (real-world)	864	112
Person data (artificial)	1,039,776	89,784

TABLE I
Overview of datasets

For experimentation, we use the first few characters of the sorting key as partition predicate. If these characters are

[1]http://www.freedb.org/
[2]http://www.cs.utexas.edu/users/ml/riddle/data.html
[3]http://tinyurl.com/dude-toolkit

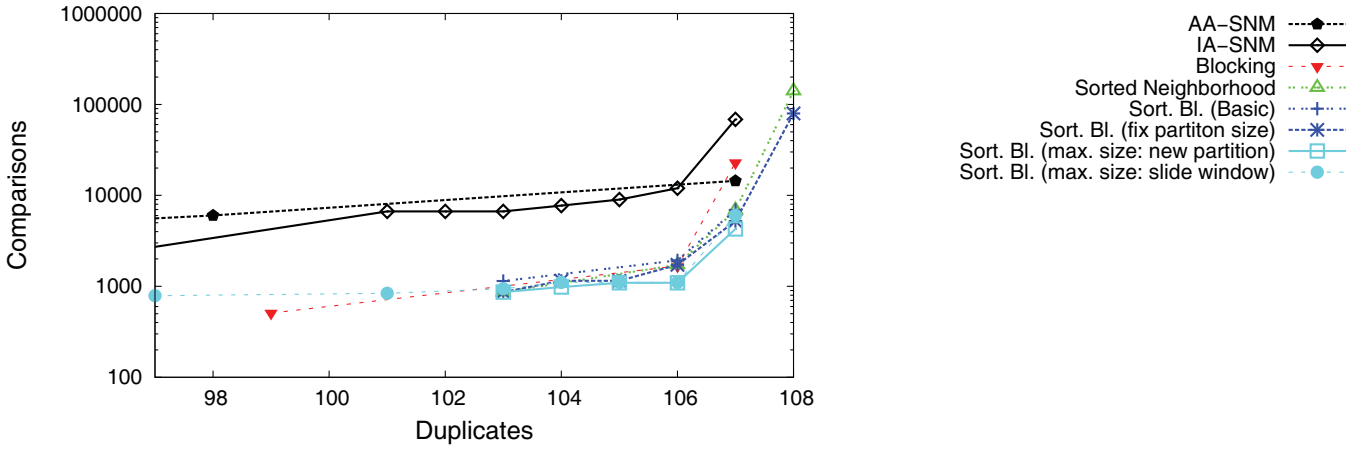


Fig. 3. Experiment results – restaurant dataset

different to those of the previous record, then we create a new partition. On the one hand, we vary the number of characters for the partition predicate, on the other hand we also use different values for the overlap parameter o and the maximum partition size.

To evaluate the performance of duplicate detection, a variety of indicators exists [5], [11]. As we are comparing algorithms for candidate pair selection, we measure the required number of comparisons to select a particular number of duplicates. The configurations can vary concerning the parameters, which are the partition predicate, the overlap parameter, and for the variants the maximum partition size. We interpolate the results to show the minimum number of required comparisons to find at least a specific number of duplicates.

To eliminate effects of a poor quality similarity function, we simply use the gold-standard available for each dataset to decide whether a record pair is a duplicate or a non-duplicate. This means, that all record pairs are classified correctly and the results depend only on the selection of record pairs to be compared.

B. Experiment results

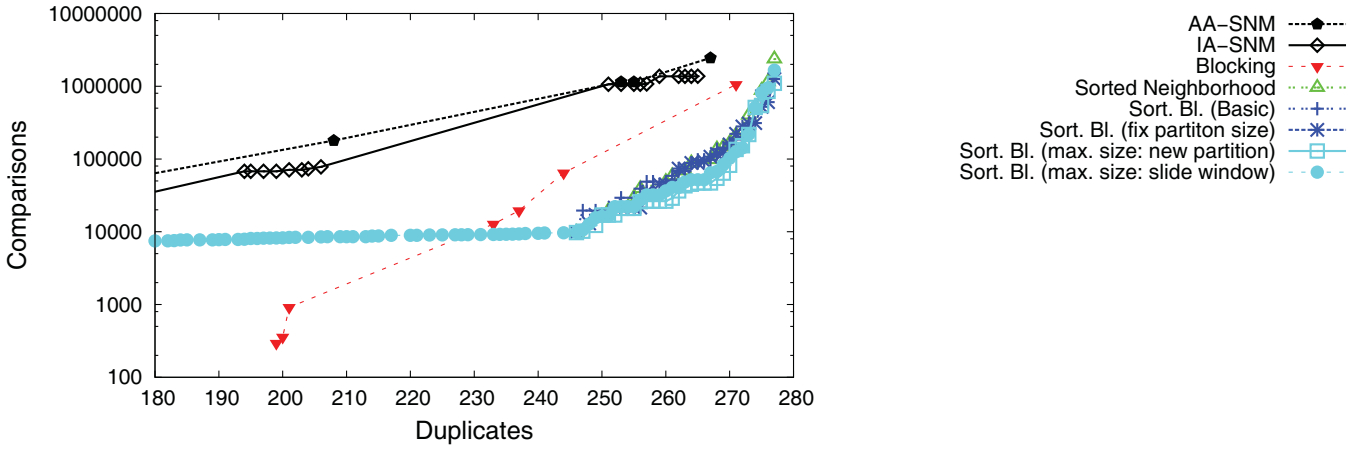
The results of the experiments with the restaurant dataset are shown in Fig. 3, which plots the number of comparisons in a logarithmic scale against the number of detected duplicates. The most relevant results are within the range from 103 up to 108 detected duplicates. Sorted Blocks (Basic) performs to some extent better than Blocking, but worse than the Sorted Neighborhood Method. The Sorted Block variants show the best performance, especially the variant that creates a new partition when the maximum partition size is reached. Overall, IA-SNM and AA-SNM require more comparisons than the other algorithms.

Figure 4 shows the results for the CD dataset. To make the differences between the algorithms more visible, we have divided the chart into parts with different values for the comparison range.

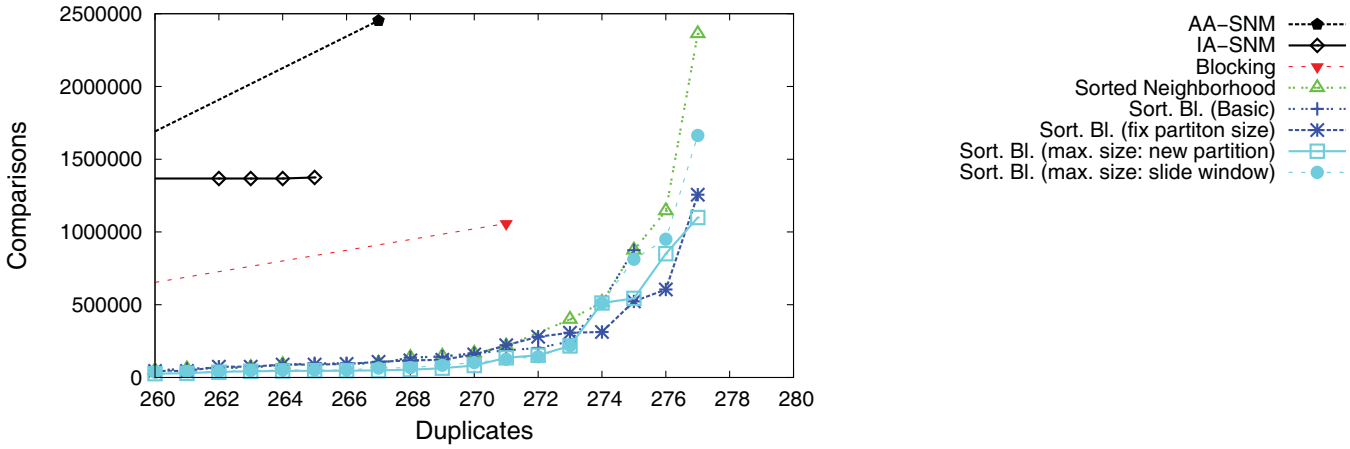
As for the restaurant dataset, IA-SNM, AA-SNM, and Blocking require the most comparisons. Sorted Blocks (Basic) performs slightly better than the Sorted Neighborhood Method, but not as good as the Sorted Blocks variants, which are again superior. This is surprising for the variant with fixed partition sizes, as the partitions are created arbitrarily. Especially for a high number of detected duplicates, the strategy to create a new partition instead of sliding a window to the end of the partition seems to be a promising approach.

The person data experiment results are shown in Fig. 5, which also divides the chart into parts with different x-axis and y-axis ranges to better analyze the differences between the algorithms. AA-SNM and IA-SNM again show the worst performance. Note that Blocking shows the good performance in Fig. 5(a) only due to the logarithmic scale of the y-axis and because there are only few connected data points. As we can see in Fig. 5(b), Blocking is not superior to the other algorithms. The Sorted Blocks variant that creates a new partition when the maximum partition size is reached requires again the fewest comparisons. Blocking and Sorted Blocks (Basic) detect more duplicates than the Sorted Neighborhood Method and the Sorted Blocks variants due to large partition sizes. Thus, we can see a high increase of the required number of comparisons from about 74,000 to 83,000 detected duplicates.

To summarize the results of the experiments, we could see that the Sorted Blocks variant that creates a new partition when the maximum partition size is reached outperforms the other algorithms. On the one hand, this variant uses the data values to create partitions of variable partition size with records that have a higher chance of being duplicates. This explains the performance gain compared to the Sorted Neighborhood Method with a fixed partition size. Additionally, the maximum partition size prevents too large partitions which result in a high number of comparisons (e.g., for Blocking). The difficulty for all Sorted Blocks variants is that they have more parameters than the Sorted Neighborhood Method or Blocking, which makes configuration slightly more complex.



(a) CD dataset results 1



(b) CD dataset results 2

Fig. 4. Experiment results – CD dataset

V. Conclusion

Efficient duplicate detection is an important task especially in large datasets. In this paper, we have compared two important approaches, blocking and windowing, for reducing the number of comparisons. Additionally, we have introduced Sorted Blocks which is a generalization of blocking and windowing. Experiments with several real-world datasets show that Sorted Blocks outperforms the two other approaches. A challenge for Sorted Blocks is finding the right configuration settings, as it has more parameters than the other two approaches.

An advantage of Sorted Blocks in comparison to the Sorted Neighborhood Method is the variable partition size instead of a fixed size window. This allows more comparisons if several records have similar values, but requires fewer comparisons if only a few records are similar. In the future, one of our research topics will be to evaluate strategies that group records with a high chance of being duplicates in the same partitions.

Acknowledgment

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

References

[1] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2002.

[2] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*, 2003.

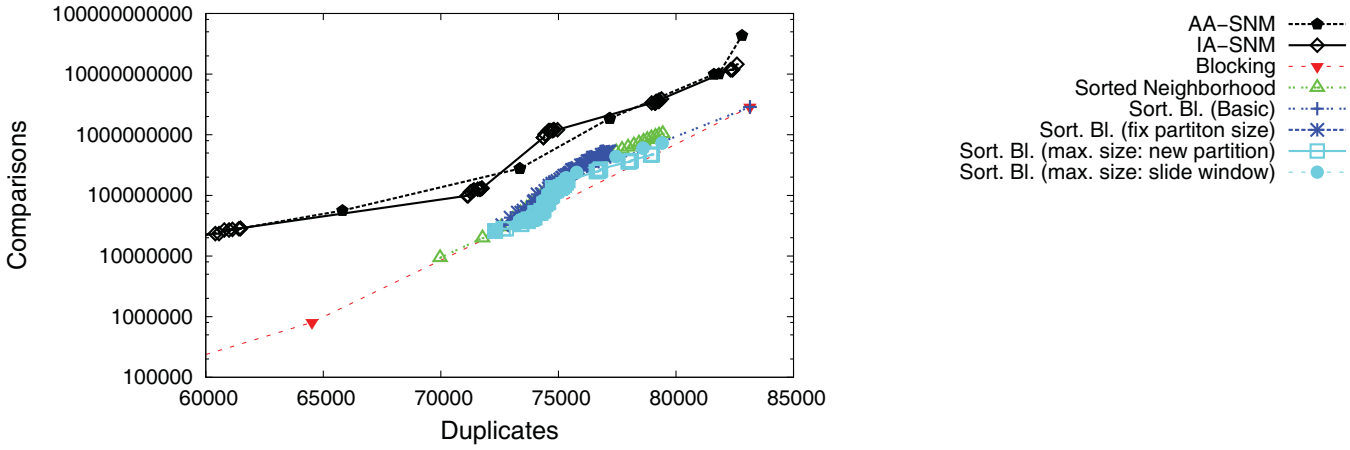
[3] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Industrial Conference on Data Mining (ICDM)*, 2006.

[4] Peter Christen. Towards parameter-free blocking for scalable record linkage. Technical Report TR-CS-07-03, The Australian National University, August 2007.

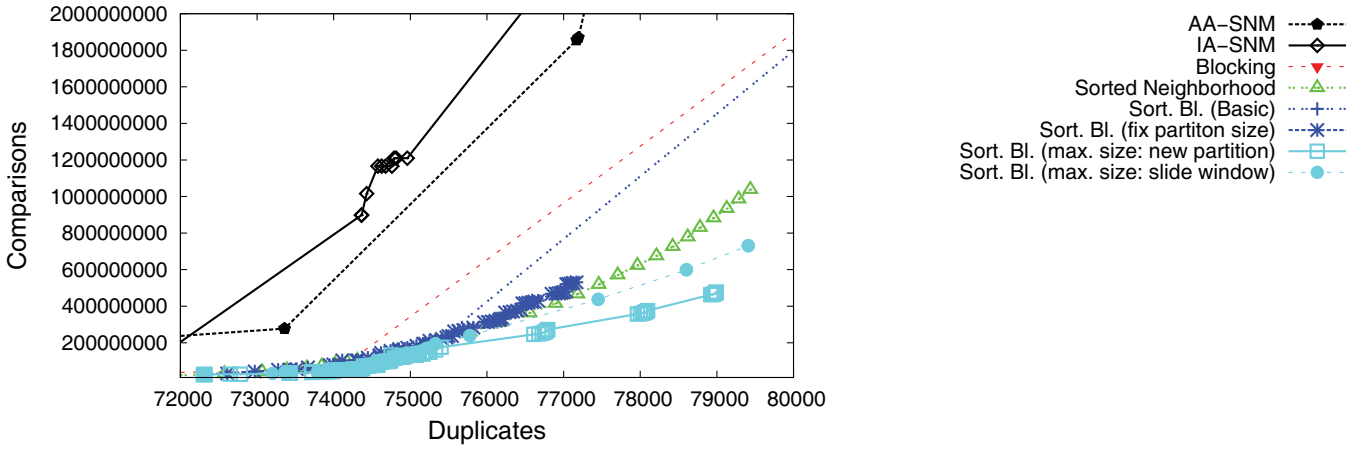
[5] Peter Christen and Karl Goiser. Quality and complexity measures for data linkage and deduplication. In *Quality Measures in Data Mining*, volume 43 of *Studies in Computational Intelligence*. Springer, 2007.

[6] Uwe Draisbach and Felix 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] Uwe Draisbach and Felix Naumann. DuDe: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2010.



(a) Person dataset results 1



(b) Person dataset results 2

Fig. 5. Experiment results – person dataset

[8] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19, 2007.

[9] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995.

[10] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1), 1998.

[11] David Menestrina, Steven Euijong Whang, and Hector Garcia-Molina. Evaluating entity resolution results. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2010.

[12] Alvaro E. Monge and Charles Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[13] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection (Synthesis Lectures on Data Management)*. Morgan and Claypool Publishers, 2010.

[14] Sven Puhlmann, Melanie Weis, and Felix Naumann. XML duplicate detection using sorted neighborhoods. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2006.

[15] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative blocking. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009.

[16] Su Yan, Dongwon Lee, Min-Yen Kan, and Lee C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, 2007.