Progressive Duplicate Detection

Thorsten Papenbrock, Arvid Heise, and Felix Naumann

Abstract—Duplicate detection is the process of identifying multiple representations of same real world entities. Today, duplicate detection methods need to process ever larger datasets in ever shorter time: maintaining the quality of a dataset becomes increasingly difficult. We present two novel, *progressive* duplicate detection algorithms that significantly increase the efficiency of finding duplicates if the execution time is limited: They maximize the gain of the overall process within the time available by reporting most results much earlier than traditional approaches. Comprehensive experiments show that our progressive algorithms can double the efficiency over time of traditional duplicate detection and significantly improve upon related work.

Index Terms—Duplicate Detection, Entity Resolution, Pay-as-you-go, Progressiveness, Data cleaning

1 Introduction

Data are among the most important assets of a company. But due to data changes and sloppy data entry, errors such as duplicate entries might occur, making data cleansing and in particular duplicate detection indispensable. However, the pure size of today's datasets render duplicate detection processes expensive. Online retailers, for example, offer huge catalogs comprising a constantly growing set of items from many different suppliers. As independent persons change the product portfolio, duplicates arise. Although there is an obvious need for deduplication, online shops without downtime cannot afford traditional deduplication.

Progressive duplicate detection identifies most duplicate pairs *early* in the detection process. Instead of reducing the overall time needed to finish the entire process, progressive approaches try to reduce the average time after which a duplicate is found. Early termination, in particular, then yields more complete results on a progressive algorithm than on any traditional approach.

As a preview of Sec. 8.3, Figure 1 depicts the number of duplicates found by three different duplicate detection algorithms in relation to their processing time: The *incremental* algorithm reports new duplicates at an almost constant frequency. This output behavior is common for state-of-the-art duplicate detection algorithms. In this work, however, we focus on *progressive* algorithms, which try to report most matches early on, while possibly slightly increasing their overall runtime. To achieve this, they need to estimate the similarity of all comparison candidates in order to compare most promising record pairs first.

With the pair selection techniques of the duplicate detection process, there exists a trade-off between the amount of time needed to run a duplicate detection algorithm and the completeness of the results. Progressive techniques make this trade-off more beneficial as they deliver more complete results in shorter amounts of time. Furthermore, they make

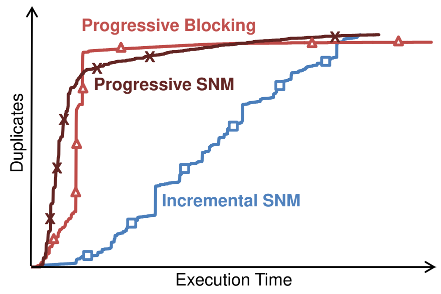


Figure 1. Duplicates pairs found by an incremental and our two progressive algorithms (see Sec. 8.3)

it easier for the user to define this trade-off, because the detection time or result size can directly be specified instead of parameters whose influence on detection time and result size is hard to guess. We present several use cases where this becomes important:

1) A user has only *limited, maybe unknown time* for data cleansing and wants to make best possible use of it. Then, simply start the algorithm and terminate it when needed. The result size will be maximized.
2) A user has *little knowledge* about the given data but still needs to configure the cleansing process. Then, let the progressive algorithm choose window/block sizes and keys automatically.
3) A user needs to do the cleaning *interactively* to, for instance, find good sorting keys by trial and error. Then, run the progressive algorithm repeatedly; each run quickly reports possibly large results.
4) A user has to achieve a certain *recall*. Then, use the result curves of progressive algorithms to estimate how many more duplicates can be found further; in general, the curves asymptotically converge against the real number of duplicates in the dataset.

We propose two novel, progressive duplicate detection algorithms namely *Progressive Sorted Neighborhood Method* (PSNM), which performs best on small and almost clean datasets, and *Progressive Blocking* (PB), which performs best on large and very dirty datasets. Both enhance the

• *The authors are with the Department of Information Systems, Hasso-Plattner-Institute, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany.*
E-mail: firstname.lastname@hpi.de

efficiency of duplicate detection even on very large datasets. In comparison to traditional duplicate detection, progressive duplicate detection satisfies two conditions [1]:

Improved Early Quality. Let *t* be an arbitrary target time at which results are needed. Then the progressive algorithm discovers more duplicate pairs at *t* than the corresponding traditional algorithm. Typically, *t* is smaller than the overall runtime of the traditional algorithm.

Same Eventual Quality. If both a traditional algorithm and its progressive version finish execution, without early termination at *t*, they produce the same results.

Given any fixed-size time slot in which data cleansing is possible, progressive algorithms try to maximize their efficiency for that amount of time. To this end, our algorithms PSNM and PB dynamically adjust their behavior by automatically choosing optimal parameters, e.g., window sizes, block sizes, and sorting keys, rendering their manual specification superfluous. In this way, we significantly ease the parameterization complexity for duplicate detection in general and contribute to the development of more user interactive applications: We can offer fast feedback and alleviate the often difficult parametrization of the algorithms. In summary, our contributions are the following:

- We propose two dynamic *progressive duplicate detection algorithms*, PSNM and PB, which expose different strengths and outperform current approaches.
- We introduce a concurrent progressive approach for the multi-pass method and adapt an incremental transitive closure algorithm that together form the first complete *progressive duplicate detection workflow*.
- We define a novel *quality measure* for progressive duplicate detection to objectively rank the performance of different approaches.
- We exhaustively *evaluate* on several real-world datasets testing our own and previous algorithms.

The *duplicate detection workflow* comprises the three steps pair-selection, pair-wise comparison, and clustering. For a progressive workflow, only the first and last step need to be modified. Therefore, we do not investigate the comparison step and propose algorithms that are independent of the quality of the similarity function. Our approaches build upon the most commonly used methods, sorting and (traditional) blocking, and thus make the same assumptions: duplicates are expected to be sorted close to one another or grouped in same buckets, respectively.

Paper organization. Section 2 examines related work. Sections 3 and 4 introduce the PSNM and the PB algorithm, which progressively find duplicates based on windowing and blocking techniques, respectively. Section 5 contributes the Attribute Concurrency multi-pass strategy, which enables PSNM and PB to automatically choose good key attributes. We discuss the incremental transitive closure calculation in Section 6 and define a novel quality measure for *progressiveness* in Section 7. Section 8 comprehensively evaluates our algorithms, showing that they can double the efficiency of traditional duplicate detection algorithms. Section 9 concludes this paper and discusses future work.

2 Related Work

Much research on duplicate detection [2], [3], also known as entity resolution and by many other names, focuses on pair-selection algorithms that try to maximize recall on the one hand and efficiency on the other hand. The most prominent algorithms in this area are Blocking [4] and the Sorted Neighborhood Method [5].

Adaptive Techniques. Previous publications on duplicate detection often focus on reducing the overall runtime. Thereby, some of the proposed algorithms are already capable of estimating the quality of comparison candidates [6]–[8]. The algorithms use this information to choose the comparison candidates more carefully. For the same reason, other approaches utilize adaptive windowing techniques, which dynamically adjust the window size depending on the amount of recently found duplicates [9], [10]. These adaptive techniques dynamically improve the efficiency of duplicate detection, but in contrast to our progressive techniques, they need to run for certain periods of time and cannot maximize the efficiency for any given time slot.

Progressive Techniques. In the last few years, the economic need for progressive algorithms also initiated some concrete studies in this domain. For instance, pay-as-you-go algorithms for information integration on large scale datasets have been presented [11]. Other works introduced progressive data cleansing algorithms for the analysis of sensor data streams [12]. However, these approaches cannot be applied to duplicate detection.

Xiao et al. proposed a top-k similarity join that uses a special index structure to estimate promising comparison candidates [13]. This approach progressively resolves duplicates and also eases the parameterization problem. Although the result of this approach is similar to our approaches (a list of duplicates almost ordered by similarity), the focus differs: Xiao et al. find the top-k most similar duplicates regardless of how long this takes by weakening the similarity threshold; we find as many duplicates as possible in a given time. That these duplicates are also the most similar ones is a side effect of our approaches.

Pay-As-You-Go Entity Resolution by Whang et al. introduced three kinds of progressive duplicate detection techniques, called “hints” [1]. A hint defines a probably good execution order for the comparisons in order to match promising record pairs earlier than less promising record pairs. However, all presented hints produce static orders for the comparisons and miss the opportunity to dynamically adjust the comparison order at runtime based on intermediate results. Some of our techniques directly address this issue. Furthermore, the presented duplicate detection approaches calculate a hint only for a specific partition, which is a (possibly large) subset of records that fits into main memory. By completing one partition of a large dataset after another, the overall duplicate detection process is no longer progressive. This issue is only partly addressed in [1], which proposes to calculate the hints using all partitions. The algorithms presented in our paper use a global ranking for the comparisons and consider the limited amount of available main memory. The third issue of the algorithms introduced by Whang et al. relates to the proposed pre-partitioning strategy: By using minhash signatures [14] for the partitioning, the partitions do not overlap. However, such an overlap improves the pair-selection [15], and thus our algorithms consider overlapping blocks as well. In contrast to [1], we also progressively solve the multi-pass method and transitive closure calculation, which are essential for a completely progressive workflow. Finally, we provide a more extensive evaluation on considerably larger datasets and employ a novel quality measure to quantify the performance of our progressive algorithms.

Additive Techniques. By combining the Sorted Neighborhood Method with blocking techniques, pair-selection algorithms can be built that choose the comparison candidates much more precisely. The Sorted Blocks algorithm [15], for instance, applies blocking techniques on a set of input records and then slides a small window between the different blocks to select additional comparison candidates. Our progressive PB algorithm also utilizes sorting and blocking techniques; but instead of sliding a window between blocks, PB uses a progressive block-combination technique, with which it dynamically chooses promising comparison candidates by their likelihood of matching.

The recall of blocking and windowing techniques can further be improved by using multi-pass variants [5]. These techniques use different blocking or sorting keys in multiple, successive executions of the pair-selection algorithm. Accordingly, we present progressive multi-pass approaches that interleave the passes of different keys.

3 Progressive SNM

The *Progressive Sorted Neighborhood Method* (PSNM) is based on the traditional *Sorted Neighborhood Method* [5]: PSNM sorts the input data using a predefined sorting key and only compares records that are within a window of records in the sorted order. The intuition is that records that are close in the sorted order are more likely to be duplicates than records that are far apart, because they are already similar with respect to their sorting key. More specifically, the distance of two records in their sort ranks (rank-distance) gives PSNM an estimate of their matching likelihood. The PSNM algorithm uses this intuition to iteratively vary the window size, starting with a small window of size two that quickly finds the most promising records. This static approach has already been proposed as the Sorted List of Record Pairs hint [1]. The PSNM algorithm differs by dynamically changing the execution order of the comparisons based on intermediate results (Look-Ahead). Furthermore, PSNM integrates a progressive sorting phase (MagpieSort) and can progressively process significantly larger datasets.

3.1 PSNM algorithm

Algorithm 1 depicts our implementation of PSNM. The algorithm takes five input parameters: D is a reference to the data, which has not been loaded from disk yet. The sorting key K defines the attribute or attribute combination that should be used in the sorting step. W specifies the maximum window size, which corresponds to the window size of the traditional Sorted Neighborhood Method. When using early termination, this parameter can be set to an optimistically high default value. Parameter I defines the enlargement interval for the progressive iterations. Sec. 3.2 describes this parameter in more detail. For now, assume it has the default value 1. The last parameter N specifies the number of records in the dataset. This number can be gleaned in the sorting step, but we list it as a parameter for presentation purposes.

Algorithm 1 Progressive Sorted Neighborhood

```
Require: dataset reference $D$, sorting key $K$, window size
    $W$, enlargement interval size $I$, number of records $N$
 1: procedure PSNM($D$, $K$, $W$, $I$, $N$)
 2:     $pSize \leftarrow$ calcPartitionSize($D$)
 3:     $pNum \leftarrow \lceil N/(pSize - W + 1) \rceil$
 4:     array $order$ size $N$ as Integer
 5:     array $recs$ size $pSize$ as Record
 6:     $order \leftarrow$ sortProgressive($D$, $K$, $I$, $pSize$, $pNum$)
 7:     for $currentI \leftarrow 2$ to $\lceil W/I \rceil$ do
 8:         for $currentP \leftarrow 1$ to $pNum$ do
 9:             $recs \leftarrow$ loadPartition($D$, $currentP$)
10:             for $dist \in$ range($currentI$, $I$, $W$) do
11:                 for $i \leftarrow 0$ to $|recs| - dist$ do
12:                     $pair \leftarrow \langle recs[i], recs[i + dist] \rangle$
13:                     if compare($pair$) then
14:                         emit($pair$)
15:                         lookAhead($pair$)
```

In many practical scenarios, the entire dataset will not fit in main memory. To address this, PSNM operates on a *partition* of the dataset at a time. The PSNM algorithm calculates an appropriate partition size *pSize*, i.e. the maximum number of records that fit in memory, using the pessimistic sampling function calcPartitionSize(D) in Line 2: If the data is read from a database, the function can calculate the size of a record from the data types and match this to the available main memory. Otherwise, it takes a sample of records and estimates the size of a record with the largest values for each field. In Line 3, the algorithm calculates the number of necessary partitions *pNum*, while considering a partition overlap of $W - 1$ records to slide the window across their boundaries. Line 4 defines the *order*-array, which stores the order of records with regard to the given key K. By storing only record IDs in this array, we assume that it can be kept in memory. To hold the actual records of a current partition, PSNM declares the *recs*-array in Line 5.

In Line 6, PSNM sorts the dataset D by key K. The sorting is done by applying our progressive sorting algorithm *Magpie*, which we explain in Sec. 3.2. Afterwards, PSNM linearly increases the window size from 2 to the maximum window size W in steps of I (Line 7). In this way, promising close neighbors are selected first and less promising far-away neighbors later on. For each of these *progressive* *iterations*, PSNM reads the entire dataset once. Since the load process is done partition-wise, PSNM sequentially iterates (Line 8) and loads (Line 9) all partitions. To process a loaded partition, PSNM first iterates over all record rank-distances *dist* that are within the current window interval *currentI*. For $I = 1$ this is only one distance, namely the record rank-distance of the current main-iteration. In Line 11, PSNM then iterates all records in the current partition to compare them to their *dist*-neighbor. The comparison is executed using the compare(*pair*) function in Line 13. If this function returns "true", a duplicate has been found and can be emitted. Furthermore, PSNM evokes the lookAhead(*pair*) method, which we explain later, to progressively search for more duplicates in the current neighborhood. If not terminated early by the user, PSNM finishes when all intervals have been processed and the maximum window size W has been reached.

3.2 Progressiveness Techniques

Window Interval. PSNM needs to load all records in each progressive iteration and loading partitions from disk is expensive. Therefore, we introduced the *window enlargement interval* I in Line 7 and 10. It defines how many *dist*-iterations PSNM should execute on each loaded partition. For instance, if we set $I = 3$, the algorithm loads the first partition to sequentially execute the rank-distances 1 to 3, then it loads the second partition to execute the same interval and so on until all partitions have been loaded once. Afterwards, all partitions are loaded again to run *dist* 4 to 6 and so forth. This strategy reduces the number of load processes. However, the theoretical progressiveness decreases as well, because we execute comparisons with a lower probability of matching earlier. So I constitutes a trade-off parameter that balances progressiveness and overall runtime.

Partition Caching. As we cannot assume the input to be physically sorted, the algorithm needs to repeatedly re-iterate the entire file searching for the records of the next partition, which contains the currently most promising comparison candidates. So, all records need to be read when loading the next partition. To overcome this issue, we implemented *Partition Caching* within the loadPartition(*D*, *currentP*) function in Line 9: If a partition is read for the first time, the function collects the requested records from the input dataset and materializes them to a new, dedicated cache file on disk. When the partition is later requested again, the function loads it from this cache file, reducing the costs for PSNM's additional I/O operations (and for possible parsing efforts on the file-input).

Look-Ahead. After sorting the input dataset, we find areas of high and low duplicate density, particularly if duplicates occur in larger clusters, i.e., groups of records that are all pair-wise duplicates. The *Look-Ahead* strategy uses this observation to adjust the ranking of comparison candidates at runtime: If record pair (i,j) has been identified as a duplicate, then the pairs $(i+1,j)$ and $(i,j+1)$ have a high chance of being duplicates of the same cluster. Therefore, PSNM immediately compares them instead of waiting for the next progressive iteration. If one of the look-ahead comparisons detects another duplicate, a further look-ahead is recursively executed. In this way, PSNM iterates larger neighborhoods around duplicates to progressively reveal entire clusters. To avoid redundant comparisons in different look-aheads or in a following progressive iteration, PSNM maintains all executed comparisons in a temporary data structure. This behavior is implemented by the lookAhead(*pair*) function in Line 15 of our PSNM implementation. Since the look-ahead works recursively, it may perform comparisons that are beyond the given maximum window size *W*. Hence, it can find duplicates that cannot be found by the traditional Sorted Neighborhood Method. For easier comparison, we limited the maximum look-ahead rank-distance to *W* in our evaluation. In summary, PSNM automatically prefers locally promising comparisons in the otherwise static execution order by adaptively comparing record pairs in the neighborhood of previously detected duplicates.

MagpieSort. The sorting of records is a blocking preprocessing step that we can already use to (progressively) execute some first comparisons. *MagpieSort* is a naïve sorting algorithm that works similar to *SelectionSort*. The name of this algorithm is inspired by the larcenous bird that collects beautiful things while only being able to carry a few of them at once. MagpieSort repeatedly iterates over all records to find the currently top-*x* smallest ones. Thereby, it inserts each record into a sorted buffer of length *x*. If the buffer is full, each newly inserted record displaces the largest record from the list. After each iteration, the final order can be supplemented by the next top *x* records from the buffer. A record that has been emitted once will not be emitted again. So for *N* records, the algorithm terminates after $\lceil \frac{N}{x} \rceil$ iterations yielding the final order of records. As each pass over the input dataset delivers a partition of appropriately sorted records, we can directly execute some promising comparisons on them. In fact, MagpieSort integrates the entire first progressive iteration of PSNM. Overall, this sorting strategy generates only a small overhead, because the algorithm needs to iterate over the entire dataset anyway whenever a partition needs to be read from disk.

Load-Compare Parallelism. The PSNM algorithm consists of two continuously alternating phases: A load phase, in which PSNM reads a partition of records from disk into main memory, and a compare phase, in which PSNM executes comparisons on the current partition. The load phase frequently blocks the algorithm's progress and reduces its progressiveness. To avoid this blocking behavior, we propose to parallelize the two phases and then use double buffering for the partitions. In this way, PSNM can hide data access latencies by simultaneously executing comparisons. Our implementation of this idea, which we call *Load-Compare Parallelism*, uses two worker-threads: a Loader and a Comparator. It also requires one partition for each worker. Since both partitions need to reside in memory at the same time, each of them can only be half the size of the overall available memory. So we define the *recs*-array twice with half of its original size. The PSNM algorithm then runs Lines 2 to 9 in the Loader thread and Lines 10 to 15 in the Comparator thread.

4 Progressive Blocking

In contrast to windowing algorithms, blocking algorithms assign each record to a fixed group of similar records (the blocks) and then compare all pairs of records within these groups. *Progressive Blocking* (PB) is a novel approach that builds upon an equidistant blocking technique and the successive enlargement of blocks. Like PSNM, it also pre-sorts the records to use their rank-distance in this sorting for similarity estimation. Based on the sorting, PB first creates and then progressively extends a fine-grained blocking. These block extensions are specifically executed on neighborhoods around already identified duplicates, which enables PB to expose clusters earlier than PSNM. Sections 8.3 and 8.4 directly compare the performance of PB and PSNM showing that PB is indeed preferable for datasets containing many large duplicate clusters.

4.1 PB intuition

Figure 2 illustrates how PB chooses comparison candidates using the block comparison matrix. To create this matrix, a preprocessing step has already sorted the records that form the Blocks 1-8 (depicted as vertical and horizontal axes). Each block within the block comparison matrix represents the comparisons of all records in one block with all records in another block. For instance, the field in the 4th row and the 5th column represents the comparisons of all records in Block 4 with all records in Block 5. Assuming a symmetric similarity measure, we can ignore the bottom left part of the matrix. The exemplary number of found duplicates is depicted in the according fields. In this example, the block comparison $(4, 5)$ delivered nine duplicates. Because of the equidistant blocking, all blocks have the same size. This eases the progressive extension process that we describe in the following. Only the last block might be smaller, if the dataset is not divisible by the desired block size.

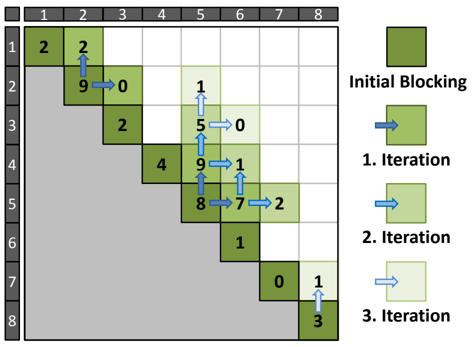


Figure 2. PB in a block comparison matrix

In the initial run, PB defines the blocking and executes all comparisons within each block. For the first progressive iteration, the algorithm then selects those block pairs that delivered the most duplicates in the initial run. In the example, these are the block pairs $(2, 2)$ and $(5, 5)$. Because these two block pairs represent the areas with the currently highest duplicate density, the PB algorithm chooses $(1, 2)$ and $(2, 3)$ to progressively extend the first block pair and $(4, 5)$ and $(5, 6)$ to extend the second block pair. Having compared the four new block pairs, PB starts the second iteration. In this iteration, $(4, 5)$ and $(5, 6)$ are the best block pairs and, hence, extended. The results of this iteration then influences the third iteration and so on. In this way, PB dynamically processes those neighborhoods that are expected to contain most new duplicates. In case of ties, the algorithm prefers block pairs with a smaller rank-distance, because the distance in the sort rank still defines the expected similarity of the records. The extensions continue until all blocks have been compared or a distance threshold for all remaining block pairs has been reached.

4.2 PB algorithm

Algorithm 2 lists our implementation of PB. The algorithm accepts five input parameters: The dataset reference D specifies the dataset to be cleaned and the key attribute or key attribute combination K defines the sorting. The parameter R limits the maximum block range, which is the maximum rank-distance of two blocks in a block pair, and S specifies the size of the blocks. We discuss appropriate values for R and S in the next section. Finally, N is the size of the input dataset.

At first, PB calculates the number of records per partition *pSize* by using a pessimistic sampling function in Line 2. The algorithm also calculates the number of loadable blocks per partition *bPerP*, the total number of blocks *bNum*, and the total number of partitions *pNum*. In the Lines 6 to 8, PB then defines the three main data structures: the *order*-array, which stores the ordered list of record IDs, the *blocks*-array, which holds the current partition of blocked records, and the *bPairs*-list, which stores all recently evaluated block pairs. Thereby, a block pair is represented as a triple of $\langle blockNr1, blockNr2, duplicatesPerComparison\rangle$. We implemented the *bPairs*-list as a priority queue, because the algorithm frequently reads the top elements from this list. In the following Line 10, the PB algorithm sorts the dataset using the progressive *MagpieSort* algorithm. Afterwards, the Lines 11 to 14 load all blocks partition-wise from disk to execute the comparisons within each block.

After the preprocessing, the PB algorithm starts progressively extending the most promising block pairs (Lines 15 to 23). In each loop, PB first takes those block pairs *bestBPs* from the *bPairs*-list that reported the highest duplicate density. Thereby, at most $bPerP/4$ block pairs can be taken, because the algorithm needs to load two blocks per *bestBP* and each extension of a *bestBP* delivers two partition block pairs *pBPs* in Line 20. However, if such an extension exceeds the maximum block range R, the last *bestBP* is discarded. Having successfully defined the most promising block pairs, Line 21 loads the corresponding blocks from disk to compare the *pBPs* in Line 22. The compare(*blocks*,

Algorithm 2 Progressive Blocking

```
Require: dataset reference $D$, key attribute $K$, maximum
    block range $R$, block size $S$ and record number $N$
procedure PB($D$, $K$, $R$, $S$, $N$)
    $pSize \leftarrow$ calcPartitionSize($D$)
    $bPerP \leftarrow \lfloor pSize/S \rfloor$
    $bNum \leftarrow \lceil N/S \rceil$
    $pNum \leftarrow \lceil bNum/bPerP \rceil$
    array order size $N$ as Integer
    array blocks size $bPerP$ as $\langle$Integer, Record[ ]$\rangle$
    priority queue bPairs as $\langle$Integer,Integer,Integer$\rangle$
    $bPairs \leftarrow \{\langle 1,1,\_\rangle, \ldots, \langle bNum, bNum, \_\rangle\}$
    $order \leftarrow$ sortProgressive($D$, $K$, $S$, $bPerP$, $bPairs$)
    for $i \leftarrow 0$ to $pNum - 1$ do
        $pBPs \leftarrow$ get($bPairs$, $i \cdot bPerP$, (i+1) $\cdot$ $bPerP$)
        $blocks \leftarrow$ loadBlocks($pBPs$, $S$, $order$)
        compare($blocks$, $pBPs$, $order$)
    while $bPairs$ is not empty do
        $pBPs \leftarrow \{\}$
        $bestBPs \leftarrow$ takeBest($\lfloor bPerP/4 \rfloor$, $bPairs$, $R$)
        for $bestBP \in bestBPs$ do
            if $bestBP[1] - bestBP[0] < R$ then
                $pBPs \leftarrow pBPs \cup$ extend($bestBP$)
        $blocks \leftarrow$ loadBlocks($pBPs$, $S$, $order$)
        compare($blocks$, $pBPs$, $order$)
        $bPairs \leftarrow bPairs \cup pBPs$
procedure COMPARE($blocks$, $pBPs$, $order$)
    for $pBP \in pBPs$ do
        $\langle dPairs, cNum \rangle \leftarrow$ comp($pBP$, $blocks$, $order$)
        emit($dPairs$)
        $pBP[2] \leftarrow |dPairs|$ / $cNum$
```

pBPs, *order*)-procedure is listed in Lines 24 to 28. For all partition block pairs *pBP*, the procedure compares each record of the first block to all records of the second block. The identified duplicate pairs *dPairs* are then emitted in Line 27. Furthermore, Line 28 assigns the duplicate pairs to the current *pBP* to later rank the duplicate density of this block pair with the density in other block pairs. Thereby, the amount of duplicates is normalized by the number of comparisons, because the last block is usually smaller than all other blocks. In Line 23, the algorithm adds the previously compared *pBPs* to the *bPairs*-list to use them in the next progressive iteration. If the PB algorithm is not terminated prematurely, it automatically finishes when the list of *bPairs* is empty, e.g., no new block pairs within the maximum block range R can be found.

4.3 Blocking Techniques

Block Size. A block pair consisting of two *small* blocks defines only few comparisons. Using such small blocks, the PB algorithm carefully selects the most promising comparisons and avoids many less promising comparisons from a wider neighborhood. However, block pairs based on small blocks cannot characterize the duplicate density in their neighborhood well, because they represent a too small sample. A block pair consisting of *large* blocks, in contrast, may define too many, less promising comparisons, but produce better samples for the extension step. The block size parameter S, therefore, trades off the execution of non-promising comparisons and the extension quality. In preliminary experiments, we identified 5 records per block to be a generally good and not sensitive value.

Maximum Block Range. The maximum block range parameter R is superfluous when using early termination. For our evaluation, however, we use this parameter to restrict the PB algorithm to approximately the same comparisons executed by the traditional Sorted Neighborhood Method. We cannot restrict PB to execute exactly the same comparisons, because the selection of comparison candidates is more fine-grained by using a window than by using blocks. Nevertheless, the calculation of R as $R = \lfloor \frac{windowSize}{S} \rfloor$ causes PB to execute only minimally fewer comparisons.

Extension Strategy. The extend(*bestBP*) function in Line 20 of Algorithm 2 returns some block pairs in the neighborhood of the given *bestBP*. In our implementation, the function extends a block pair (i,j) to the block pairs $(i+1,j)$ and $(i,j+1)$ as shown in Figure 2. More eager extension strategies that select more block pairs from the neighborhood increase the progressiveness, if many large duplicate clusters are expected. By using a block size S close to the average duplicate cluster size, more eager extension strategies have, however, not shown a significant impact on PB's performance in our experiments. The benefit of detecting some cluster duplicates earlier was usually as high as the drawback of executing fruitless comparisons.

MagpieSort. To estimate the records' similarities, the PB algorithm uses an order of records. As in the PSNM algorithm, this order can be calculated using the progressive *MagpieSort* algorithm. Since each iteration of this algorithm delivers a perfectly sorted subset of records, the PB algorithm can directly use this to execute the initial comparisons. In this way, the entire initialization loop listed in Lines 11-14 can be integrated into the sorting step.

5 Attribute Concurrency

The best sorting or blocking key for a duplicate detection algorithm is generally unknown or hard to find. Most duplicate detection frameworks tackle this key selection problem by applying the *multi-pass* execution method. This method executes the duplicate detection algorithm multiple times using different keys in each pass. However, the execution order among the different keys is arbitrary. Therefore, favoring good keys over poorer keys already increases the progressiveness of the multi-pass method. In this section, we present two multi-pass algorithms that dynamically interleave the different passes based on intermediate results to execute promising iterations earlier. The first algorithm is the *Attribute Concurrent PSNM* (AC-PSNM), which is the progressive implementation of the multi-pass method for the PSNM algorithm, and the second algorithm is the *Attribute Concurrent PB* (AC-PB), which is the corresponding implementation for the PB algorithm.

Algorithm 3 Attribute Concurrent PSNM

```
Require: dataset reference $D$, sorting keys $Ks$, window size
    $W$, enlargement interval size $I$ and record number $N$
 1: procedure AC-PSNM($D$, $Ks$, $W$, $I$, $N$)
 2:     $pSize \leftarrow$ calcPartitionSize($D$)
 3:     $pNum \leftarrow \lceil N/(pSize - W + 1) \rceil$
 4:     array $orders$ dimension $|Ks| \times N$ as Integer
 5:     array $windows$ size $|Ks|$ as Integer
 6:     array $dCounts$ size $|Ks|$ as Integer
 7:     for $k \leftarrow 0$ to $|Ks| - 1$ do
 8:         $\langle orders[k],\ dCounts[k] \rangle \leftarrow$ sortProgressive($D$, $I$,
                                          $Ks[k]$, $pSize$, $pNum$)
 9:         $windows[k] \leftarrow 2$
10:     while $\exists\ w \in windows : w < W$ do
11:         $k \leftarrow$ findBestKey($dCounts$, $windows$)
12:         $windows[k] \leftarrow windows[k] + 1$
13:         $dPairs \leftarrow$ process($D$, $I$, $N$, $orders[k]$,
                                  $windows[k]$, $pSize$, $pNum$)
14:         $dCounts[k] \leftarrow |dPairs|$
```

Algorithm 4 Attribute Concurrent PB

```
Require: dataset reference $D$, sorting keys $Ks$, maximum
    block range $R$, block size $S$ and record number $N$
 1: procedure AC-PB($D$, $Ks$, $R$, $S$, $N$)
 2:     $pSize \leftarrow$ calcPartitionSize($D$)
 3:     $bPerP \leftarrow \lfloor pSize/S \rfloor$
 4:     $bNum \leftarrow \lceil N/S \rceil$
 5:     $pNum \leftarrow \lceil bNum/bPerP \rceil$
 6:     array $orders$ dimension $|Ks| \times N$ as Integer
 7:     array $blocks$ size $bPerP$ as $\langle$Integer, Record[ ]$\rangle$
 8:     list $bPairs$ as $\langle$Integer, Integer, Integer, Integer$\rangle$
 9:     for $k \leftarrow 0$ to $|Ks| - 1$ do
10:         $pairs \leftarrow \{\langle 1, 1, \_, k \rangle\ ,\ ..., \langle bNum, bNum, \_, k \rangle\}$
11:         $orders[k] \leftarrow$ sortProgressive($D$, $Ks[k]$, $S$, $bPerP$,
                                                      $pairs$)
12:         $bPairs \leftarrow bPairs \cup pairs$
13:     « see Algorithm 2 Lines 15 to 23 »
```

5.1 Attribute Concurrent PSNM

The basic idea of AC-PSNM is to weight and re-weight all given keys at runtime and to dynamically switch between the keys based on intermediate results. Thereto, the algorithm precalculates the sorting for each key attribute. The precalculation also executes the first progressive iteration for every key to count the number of results. Afterwards, the algorithm ranks the different keys by their result counts. The best key is then selected to process its next iteration. The number of results of this iteration can change the ranking of the current key so that another key might be chosen to execute its next iteration. In this way, the algorithm prefers the most promising key in each iteration.

Algorithm 3 depicts our implementation of AC-PSNM. It takes the same five parameters as the basic PSNM algorithm but a set of keys *Ks* instead of a single key.

First, AC-PSNM calculates the partition size *pSize* and the overall number of partitions *pNum*. During execution, each key is assigned an own state. To encode these states, the algorithm defines three basic data structures in Lines 4 to 6: an *orders*-array, which stores the different orders, a *windows*-array, which stores the current window range for each key, and a *dCounts*-array, which stores the keys' current duplicate counts. To initialize these data structures, Line 7 iterates all given keys. For each key, the algorithm uses *MagpieSort* in Line 8 to create the corresponding order. Simultaneously, it calculates and counts the duplicates of the key's first progressive iteration. In Line 9, AC-PSNM then stores the number 2 as the recently used window range for the current key.

After initialization, AC-PSNM enters the main loop in Line 10. This loop continues until the maximum window size *W* has been reached with all keys. In the loop's body, the algorithm first selects the key *k* that delivered the most duplicates in the last iteration by consulting the *dCounts*-array in Line 11. To execute the next progressive iteration for *k*, the algorithm first increases the corresponding window range by one. Then, it calls the process(...) function that runs the PSNM algorithm with only the specified rank-distance. Afterwards, Line 14 updates the duplicate count of the current key with the amount of newly found duplicates. Due to the update, AC-PSNM might select another best key in the next iteration. In this way, the algorithm dynamically re-ranks the sorting keys.

Note that the process(...) function in Line 13 handles record comparisons slightly different than *MagpieSort* in Line 8. Since the initialization uses the keys in arbitrary order, *MagpieSort* counts *all* duplicates that are found in the first iterations to treat all keys equally. Afterwards, the process(...) function reports only *new* duplicates that have not been found before with a different key. This change in behavior guarantees that the progressive main loop always chooses the currently most promising key. Counting only new duplicates also causes the algorithm to automatically rank those keys last, whose orders are subsumed by other keys' orders. For instance, "postcode" might displace "city" as a key in an address dataset, because it usually generates a similar but more fine-grained order.

5.2 Attribute Concurrent PB

Instead of scheduling progressive iterations of different keys, AC-PB directly schedules the *bPair*-comparisons of all keys: AC-PB first calculates the initial block pairs and their duplicate counts for all keys (see Figure 2 in Section 4.1); then, it takes all block pairs together and ranks them regardless of the key, with which the individual blocks have initially been created. This approach lets AC-PB rank the comparisons even more precisely than AC-PSNM.

Algorithm 4 shows the implementation of our AC-PB algorithm. Basically, AC-PB works like the already presented PB algorithm with only a few changes: It takes the same five input parameters as the PB algorithm, except that it now takes a *set* of sorting keys *Ks*. Furthermore, AC-PSNM needs to allocate an *array* of orders holding one order for each given sorting key (Line 6). This key-separation is not needed for the *bPairs*-list in Line 8, because AC-PB merges all block pairs based on any order in this list. To match a block pair with its corresponding order, AC-PB implements the block pairs as quadruples containing their sorting key's number in the fourth field. Lines 9 to 11 initialize the three data structures *orders*, *blocks*, and *bPairs* by iterating all sorting keys. Line 10 creates the initial block pairs and directly assigns the corresponding key *k* to them. Afterwards, the AC-PSNM algorithm uses *MagpieSort* to calculate the order for the current key. As in the PB algorithm, the progressive sorting also evaluates the initial block pairs and stores the resulting duplicate counts within them. Having finished the initialization, AC-PSNM holds the orders of all sorting keys and one list containing all block pairs. In Line 13, the algorithm then starts to progressively process the block pairs by simply executing the PB algorithm.

The main loop interleaves the enlargements and comparisons of all block pairs by always choosing the most promising block pairs. In this way, the algorithm exploits the different strengths and weaknesses of each key individually. For instance, one key might be good in grouping records of duplicate cluster *A* and another key might group records of cluster *B* more efficiently.

6 Transitive Closure

Due to careful pair-selection and the use of similarity thresholds, the result of a duplicate detection run is usually not transitively closed: the record pairs (a, b) and (b, c) might be recognized as duplicates but (a, c) is (yet) missing in the result. Traditional duplicate detection algorithms, therefore, calculate the transitive closure of all results in the end [16]. As this calculation is blocking in nature, it hinders progressivity. Therefore, we propose to calculate the transitive closure incrementally while the detection algorithm is running.

A suitable incremental transitive closure algorithm has already been introduced by Wallace and Kollias [17]. The proposed algorithm incrementally adds new duplicates, which are given as pairs of record identifiers, to an internal data structure that serves to calculate transitive relations from current results. The proposed data structure comprises two sorted lists of duplicates – one sorted by first records and one sorted by second records. If n is the number of records in the result, the proposed data structure exhibits an insert complexity of $\mathcal{O}(n + log(n))$ and a read complexity of $\mathcal{O}(log(n))$. As these complexities would introduce a significant performance drawback to our progressive workflow, we instead store the duplicates in an index structure: We directly map each record identifier to a set of record identifiers representing a duplicate cluster. To add a new duplicate, we lookup the two contained records and point them to the same cluster, in which we add both records. Because of the map's overhead, this data structure requires approximately 75% more memory. However, inserts and reads can be done in amortized constant time.

7 Measuring Progressiveness

In the previous sections, we presented the two progressive pair-selection algorithms PSNM and PB, complemented them with respective multi-pass methods, and finalized their results by incrementally calculating the transitive closure. To measure their performance in the next section, we now introduce our novel quality measure. As this measure is sensitive to the system running the duplicate detection process, we first discuss four exemplary system types and then lead over to the definition.

7.1 Range of system types

The following system types differ in their availability of computational resources. Duplicate detection in these systems must, hence, serve individual requirements:

Fluctuating System. The load on many systems fluctuates. As data cleansing consumes resources, a fluctuating system has to perform data cleansing tasks at time intervals when its load is low. As the duration of available resources is unpredictable, progressive duplicate detection makes most use of that time.

Pipeline System. Database and ETL systems use pipeline strategies to process their input data. In these systems, data is passed through multiple operators. Since a duplicate detection component executes many complex record comparisons, it might lower the pipeline's execution speed significantly. Progressive duplicate detection algorithms tackle this issue by maximizing the component's output performance especially in the starting phase.

Timeslot System. Sometimes, the operation mode of a system is very strict or follows clear structures. In those systems, we observe well known, fixed sized timeslots of lower and higher system load. A typical timeslot system is the ERP-System of a non-globalized company. At night and on weekends the systems load decreases for a predictable period of time and resources become available for data cleansing. In any of these timeslots, progressive algorithms can maximize the output of duplicate detection processes.

Economic System. From the economic point of view, every IT-System is a cost factor in a company, because the usage of hardware resources must be paid and the system's execution time might prevent other jobs from being done. The quality of these systems is, hence, measured using a cost-benefit calculation. Especially for traditional duplicate detection processes, it is difficult to meet a *budget* limitation, because their runtime is hard to predict. By delivering as many duplicates as possible in a given amount of time, progressive processes optimize the cost-benefit ratio.

7.2 Quality Measure

We now define a novel metric to measure *efficiency over time*. The efficiency of a duplicate detection algorithm is defined by its cost-benefit ratio, where the costs correspond to the algorithm's runtime and the benefit to the number of found duplicates. Hence, the measure focuses on recall and not on precision. Precision is a property of the similarity function, which we do not evaluate in this paper.

Definition 1: Progressive quality: Given the total number of duplicates N in a dataset, a weighting function $\omega(t)$ over time, and the result function $r(t)$ for the number of duplicates found in the time interval $(t-1, t]$, then the progressive quality $Q(T)$ of a duplicate detection algorithm for the measurement time T is defined by the discrete sampling function:

$$Q(T) = \frac{1}{N} \cdot \sum_{t=1}^{T} (\omega(t) \cdot r(t)) \qquad (1)$$

Functions $\omega(t)$ and $r(t)$ are formally defined later. All results that an algorithm delivers later than T are ignored for its evaluation. In particular, once the fastest (progressive or non-progressive) algorithm terminates, further results of any other algorithm are worthless. Hence, we define T as follows:

Definition 2: Measurement time: Given n duplicate detection algorithms with individual overall runtimes T_i on the same dataset and hardware, the measurement time T for the progressive quality measure $Q(T)$ is defined as

$$T = min\{T_1, T_2, ...T_n\}$$

In Definition 1, N is used to normalize the quality values so that $Q(T) \in [0, 1]$. Furthermore, $r(t)$ gives the number of newly found duplicates in the time interval $(t-1, t]$. This function is evaluated in discrete, equidistant intervals. Generally, we can choose any sampling rate for the measurement intervals, but the higher the sampling rate is chosen, the more precise the final quality value is. In Formula (1), each duplicate measurement is also weighted by a system-specific, time-dependent weighting function $\omega(t)$. One may interpret $\omega(t)$ as the probability that the algorithm is still running at time t and that it has not been terminated before. We define this function as follows:

Definition 3: Weighting function: Given a measurement time T, the weighting function $\omega(t)$ for a progressive quality measure can be any function satisfying the following three conditions:

1) $\omega(t): \ \{t \mid 0 < t \leq T\} \longrightarrow \{w \mid 0 \leq w \leq 1\}$
2) $\omega(t) \geq \omega(t+1)$
3) $\omega(1) = 1$

Firstly, $\omega(t)$ has to be defined for the entire measurement time T to be used for the calculation of $Q(T)$. Thereby, $\omega(t)$ weakens the result counts of $r(t)$ by assigning weights between 0 and 1. This condition guarantees a final quality ≤ 1. As $\omega(t)$ is used to weight progressiveness, the second condition states that the weighting function must monotonically decrease, ensuring that early results are never weighted lower than later results. The last condition specifies that the first weight must be 1 for any $\omega(t)$-function: an ideal progressive algorithm, which immediately reports all results right at the beginning, shall achieve a quality of 1, regardless of the concrete weighting function.

The weighting function of choice depends on the given use case. We propose four possible weighting functions for previously introduced system types in Figure 3.

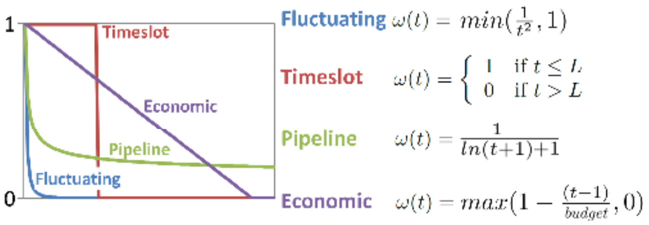


Figure 3. Weighting functions for our system types

The special economic weighting function $\omega(t) = max(1 - \frac{(t-1)}{T}, 0)$ makes $Q(T)$ equivalent to the area under the curve of the result graph. Furthermore, the weighting function $\omega(t) = 1$ leads to $Q(T) = \frac{1}{N} \cdot \sum_{t=1}^{T} r(t)$, which is the definition of recall. These two measures are often used to evaluate the performance of an algorithm, but they are only two possible instances of our more general measure and have not been applied to evaluate progressiveness, yet.

8 Evaluation

In the previous sections, we presented two progressive duplicate detection algorithms namely PSNM and PB, and their Attribute Concurrency techniques. In this section, we first generally evaluate the performance of our approaches and compare them to the traditional *Sorted Neighborhood Method* (SNM) and the *Sorted List of Record Pairs* (SLORP) presented in [1]. Then, we test our algorithms using a much larger dataset and a concrete use case. The graphs used for performance measurements plot the total number of reported duplicates over time. Each duplicate is a positively matched record pair. For better readability, we manually marked some data points from the many hundred measured data points that make up a graph.

8.1 Experimental setup

To evaluate the performance of our algorithms, we chose three real-world datasets with different characteristics (see Table 1). Since only the CD-dataset comes with an own true gold-standard, we computed duplicates in the DBLP- and CSX-dataset by running an exhaustive duplicate detection process using our fixed and reasonable (but for our evaluation irrelevant) similarity measure.

Table 1
Real-world datasets and their characteristics

Name	CD	DBLP	CSX
Records	9,763	1,268,017	1,385,532
Duplicates	277	67,586	195,042
Threshold	0.7	0.85	0.85
Best Key	`Track01`	`Title`	`Title`

The CD-dataset[1] contains various records about music and audio CDs. The DBLP-dataset[2] is a bibliographic index

1. www.hpi.de/naumann/projects/data-quality-and-cleansing/dude
2. www.informatik.uni-trier.de/~ley/db/

on computer science journals and proceedings. In contrast to the other two datasets, DBLP includes many, large clusters of similar article representations. The CSX-dataset[3] contains bibliographic data used by the CiteSeerX search engine for scientific digital literature. CSX also stores the full abstracts of all its publications in text-format. These abstracts are the largest attributes in our experiments.

Our work focuses on increasing efficiency while keeping the same effectiveness. Hence, we assume a given, correct similarity measure; it is treated as an exchangeable black box. For our experiments, however, we use the Damerau-Levenshtein similarity [18]. This similarity measure achieved an actual precision of 93% on the CD-dataset, for which we have a true gold standard.

The first part of our evaluation is executed on a DELL Optiplex 755 comprising an Intel Core 2 Duo E8400 3 GHz and 4 GB RAM. We use Ubuntu 12.04 32 bit as operating system and Java 1.6 as runtime environment. The evaluation of Sec. 8.6 uses a different machine, explained there.

Memory limitation. We assume that many real-world datasets are considerably larger than the amount of available main memory,e.g., in our use case described in Sec. 8.6. Therefore, we limit the main memory of our machine to 1 GB so that the DBLP- and CSX-dataset do not fit into main memory entirely. 1 GB of memory corresponds to about 100 000 records that can be loaded at once. The artificial limitation actually degrades the performance of our algorithms more than the performance of the non-progressive baseline, because progressive algorithms need to access partitions several times. As our experiments show, using more memory significantly increases the progressiveness of both PSNM and PB. Sec. 8.6 further shows that all results on 1 GB main memory can be extrapolated to larger datasets being processed using more main memory.

Quality measure. To evaluate the progressiveness of our algorithms, we use the quality measure proposed in Sec. 7.2. For the weighting function, we generally choose $\omega(t) = max(1 - \frac{(t-1)}{T}, 0)$, i.e., the area under the curve of the corresponding result graph. In this way, the calculated quality values are visually easy to understand.

Baseline approach. The baseline algorithm, which we use in our tests, is the standard Sorted Neighborhood Method (SNM). This algorithm has been implemented similar to the PSNM algorithm so that it may use load-compare parallelism as well. In our experiments, we always execute SNM and PSNM with the same parameters and optimizations to compare them in a fair way.

8.2 Optimizations in PSNM

Before we compare our PSNM algorithm to the PB algorithm and existing approaches, we *separately* evaluate PSNM's different progressive optimizations. We use a window size of 20 in all these experiments.

Window Interval. The window interval parameter I is a trade-off parameter: Small values close to 1 favor progressiveness at any price while large values close to the window size optimize for a short overall runtime. In all our experiments, $I = 1$ performs best, achiebing, for instance, 67% progressiveness on the DBLP-dataset. On the same dataset, the performance reduces to 65% for $I = 2$, to 62% for $I = 4$ and to 48% for $I = 10$. Hence, we suggest to set $I = 1$ if early termination can be used.

Partition Caching. Although eventually PSNM executes the same comparisons as the traditional SNM approach, the algorithm takes longer to finish. The reason for this observation is the increased number of highly expensive load processes. To reduce their complexity, PSNM implements partition caching. We now evaluate the traditional SNM algorithm, a PSNM algorithm without partition caching and a PSNM algorithm with partition caching on the DBLP-dataset. The results of this experiment are shown in Figure 4 in the left graph. The experiment shows that the benefit of partition caching is significant: The runtime of PSNM decreases by 42% minimizing the runtime difference between PSNM and SNM to only 2%.

Look-Ahead. To optimize the selection of comparison candidates, PSNM's look-ahead strategy dynamically executes comparisons around recently identified duplicates. In the following experiment, we evaluate the gain of this optimization. As in the previous experiment, we compare the look-ahead optimized PSNM to the non-optimized PSNM on the DBLP-dataset. As the results in the right graph of Figure 4 show, the look-ahead strategy clearly improves the progressiveness of the PSNM algorithm: The measured quality increases from 37% to 64%. This is a quality gain of 42%. On the CSX-dataset, however, the performance increases by only 7% from 70% to 75%. The reason is that the benefit of the look-ahead optimization greatly depends on the number and the size of duplicate clusters contained within a dataset. The CSX-dataset contains only few large clusters of similar records and, therefore, exhibits a very homogeneous distribution of duplicates, which is why the look-ahead strategy achieves only a small gain in progressiveness on that dataset.

Load-Compare Parallelism. By parallelizing the load phase and the compare phase, the load time for partitions should ideally no longer affect the performance. The following experiments evaluate this assumption for our PSNM. Since the load-compare parallelism also improves the traditional SNM, the experiment runs SNM with and without parallelization as well. Figure 5 illustrates the results of the experiment.

On the DBLP-dataset, load-compare parallelism performs almost perfectly: the entire load-time is hidden by the compare-time so that the optimized PSNM algorithm and the optimized SNM algorithm finish nearly simultaneously. This is due to the fact that the latency hiding effect reduced the runtime of the PSNM algorithm by 43% but the runtime of the SNM algorithm by only 5%. On the larger CSX-dataset, however, the load-compare parallelism strategy reduces the runtime of the SNM algorithm by 11% and the runtime of the PSNM algorithm by only

3. csxstatic.ist.psu.edu/about/data

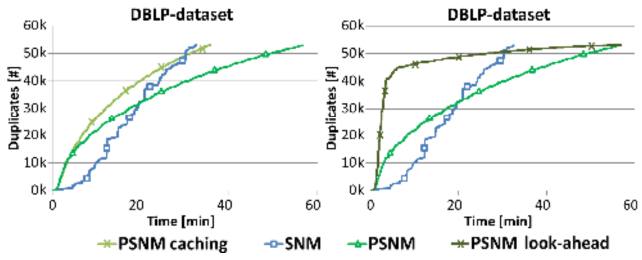


Figure 4. Effect of *partition caching* and *look-ahead*

25%. This is a remarkable gain, but since the load phases are much longer than the compare phases on this dataset, the optimization cannot hide the full data access latency: the CSX-dataset contains many enormously large attribute values that increase the load time a lot.

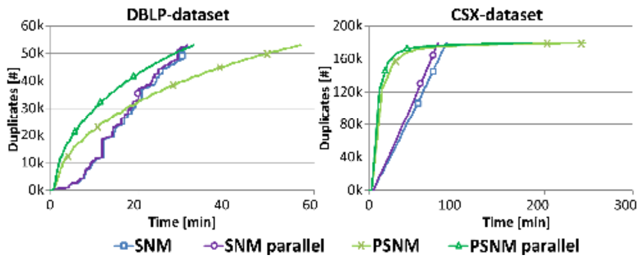


Figure 5. Evaluation of the *Load-Compare Parallelism*

Although the load-compare parallelism improves the PSNM algorithm, all further experiments do *not* use this optimization; the comparisons would become unfair using parallelization for some algorithms and no parallelization on some other algorithms, in particular those of [1].

8.3 Comparison to related work

In the following experiment, we evaluate our algorithms PSNM and PB on all four datasets. We use the traditional, non-progressive SNM algorithm as baseline to measure the real benefit of PSNM and PB. Furthermore, the experiment includes an implementation of the *Sorted List of Record Pairs* (SLORP) hint [1], which we consider to be the best progressive duplicate detection algorithm in related work. For fairness, SLORP also uses partition caching, because text-files had not been considered as input format in that work. The experiment uses a maximum window size of 20 for PSNM, SNM, and SLORP. In accordance with Sec. 4.3, we set both PB's block size and PB's block range to 5. So, the PB algorithm executes 11% fewer comparisons on each dataset than the three other approaches. The results of the experiment are depicted in Figure 6.

Low latency. On all datasets PSNM and PB start reporting first results about 1-2% earlier than SNM and SLORP. This advantage is a result of our progressive *MagpieSort*. For the non-progressive algorithms, we use an implementation of the Two-Phase Multiway Merge Sort (TPMMS), which is a popular approach for external memory sorting. Although TPMMS is highly efficient, Magpie-Sorting slighly outperforms this approach regarding progressiveness.

PSNM. In all three test runs, PSNM achieves the best performance, approximately doubling the progressiveness of the SNM baseline algorithm. PSNM also significantly outperforms the SLORP algorithm. In our experiment, PSNM exhibits a 6% (CSX) to 29% (DBLP) higher progressiveness than SLORP.

PB. The PB algorithm is the second best algorithm in this experiment. As the progressiveness of this algorithm highly benefits from more and larger duplicate clusters, it shows its best performance on the DBLP-dataset. In general, PB reports first duplicates in the starting phase clearly slower than the PSNM, because running a window of size 1 is initially more efficient than running the first block comparisons. In the following phases, however, PB resolves duplicate clusters extremely fast. Overall, PSNM is still 3% more progressive than PB on the DBLP-dataset. Thereby, we need to consider that PB executes 11% fewer comparisons than PSNM and, therefore, finds 4% fewer duplicates. Hence, PB actually competes well with PSNM on skewed datasets but loses on uniformly distributed duplicates in single-pass settings.

I/O-Overhead. For a given dataset, the tasks of sorting, candidate generation, and record comparison all have the same runtime in both progressive and non-progressive algorithms. However, the progressive algorithms require more I/O operations if the data does not fit into main memory. This causes their overall runtimes to increase, which then reduces their progressivity. Figure 6 shows these runtime differences especially for the large CSX-dataset. If the data fits into main memory, e.g., for the CD-dataset, this effect cannot be observed.

Pairs Quality. To show how precise comparison candidates are chosen, we evaluated the pairs quality *PQ* [19] of PSNM, PB, and SNM over time. The *PQ* of a duplicate detection algorithm at time *t* is the number of identified duplicates at *t* divided by the number of comparisons that were executed to find these duplicates. So the perfect duplicate detection algorithm comparing only those record pairs that in fact are duplicates yields *PQ*=1. Figure 7 depicts the *PQ*-value curves for the CSX-dataset (left chart). As the curves show, the two progressive approaches choose their comparison candidates much more carefully: The PSNM algorithm detects a new duplicate with every 12th and PB with every 20th comparison in the first few minutes. The baseline approach, in contrast, reports fewer than one duplicate in 100 comparisons. In the end, all algorithms have executed (almost) the same comparisons, so that their *PQ* curves converge to the same value.

Precision and Recall. The proposed progressive algorithms enhance the efficiency and usability of duplicate detection processes, but do not change their effectiveness. Of course, the similarity function used to determine duplicates must match the characteristics of the used sorting key(s). But both similarity function and keys are irrelevant for the progressiveness of our algorithms. In other words: If the similarity function is poor, we obtain the same poor results from progressive and non-progressive algorithms.

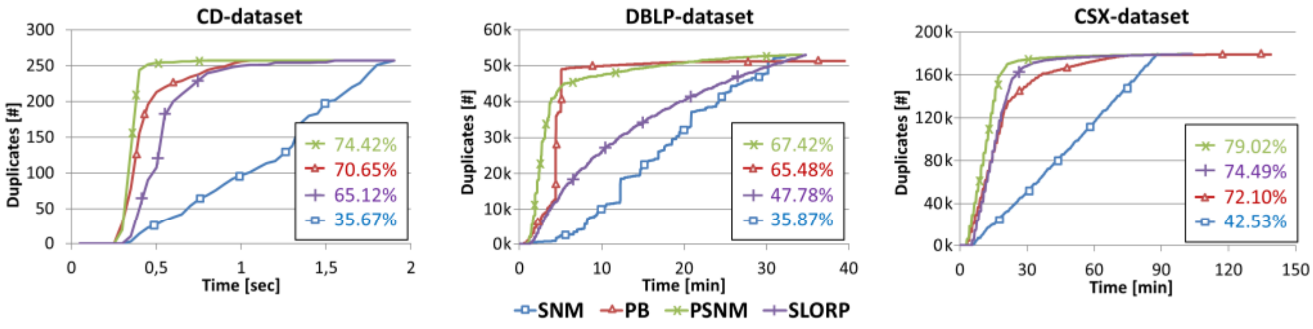


Figure 6. Performance comparison of the traditional SNM and the progressive PB, PSNM, and PB algorithms

To illustrate this behavior, we evaluated the change in precision and recall on the CD-dataset, which is the only dataset for which a true gold-standard is given. As the right chart in Figure 7 shows, the recall curves correspond to the previous duplicate curves. The precision curves, on the other hand, give the following insights: First, the final precision of 93% is relatively high, which underlines the suitability of the used similarity function. Second, both SNM and PSNM have very similar values in precision, which verifies the irrelevance of the similarity measure for progressiveness. Third, the progressive algorithms find fewer false positive matches in relation to true positive matches in the beginning, as the precision graphs show.

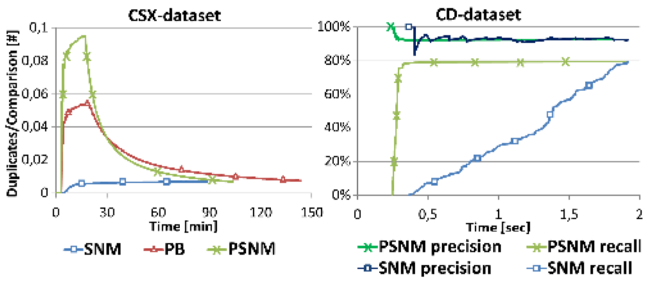


Figure 7. Evaluation on pairs quality *PQ* (left) and precision and recall (right)

8.4 Attribute Concurrency

Our Attribute Concurrency algorithms AC-PSNM and AC-PB progressively execute the multi-pass method for the PSNM algorithm and PB algorithm, respectively, favoring good keys over poor keys by dynamically ranking different passes using their intermediate results. In the following, we compare AC-PSNM and AC-PB to the common multi-pass execution model, which resolves the different keys sequentially in random order. The experiment uses three different keys, which are {Title}, {Authors}, and {Description}. Since a common multi-pass algorithm can execute the different passes in any order, it might accidentally choose the best or worst order of keys. Therefore, we run the traditional, sequential multi-pass algorithm with the optimal key Sequence 1, two mediocre key Sequences 2 and 3 and the worst key Sequence 4. The corresponding

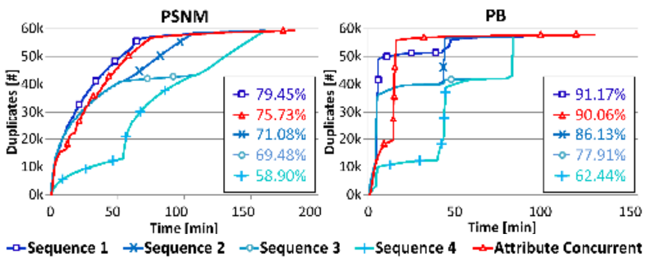


Figure 8. *Attribute Concurrency* on the DBLP-dataset

graphs are depicted in Figure 8. The fifth graph in both charts shows the AC-strategy for the respective algorithm.

First of all, both charts show that the AC-approaches need about 10% more time to finish. This is because the ranking of intermediate results and the scheduling of different keys takes some additional time. Moreover, both approaches need to store all orders simultaneously in main memory, which decreases the size of their partitions.

We first evaluate the results for the AC-PSNM algorithm. With a progressiveness of 79%, Sequence 1 is the best approach. Our AC-PSNM algorithm, then, delivers the second best result with 76% followed by all other results. Thereby, the worst sequence achieves a progressive quality of only 59%.

Due to the overhead of creating all orders and lots of initial block pairs, the PB approach loses much time early on. But after 18 minutes runtime, the attribute concurrent PB algorithm outperforms all other multi-pass approaches, because it has finished the initial runs and can now simultaneously use the benefits of all orders. Therefore, its overall progressiveness of 90% is almost as good as the progressiveness of the best sequence, which is 91%. The worst sequence of sorting keys, in contrast, achieves only 62% progressive performance, which is about $\frac{1}{3}$ less than the best two approaches.

In summary, both attribute concurrent approaches offer a good progressive quality. Although they might not find the most progressive multi-pass configurations, they always produce reliable execution orders for the different passes. We also see that PB outperforms PSNM in multi-pass settings. Finally, it is worth noting that due to dynamically generated execution orders only little expert knowledge is needed in creating good sorting or blocking keys.

8.5 Incremental Transitive Closure

In this experiment, we evaluate the computational overhead caused by the incremental calculation of the transitive closure. We take a result set of one million duplicates (a subset of duplicates found in the use case of Sec. 8.6), submit it to the transitive closure algorithm and measure the time after each insert. Figure 9 plots the resulting curve.

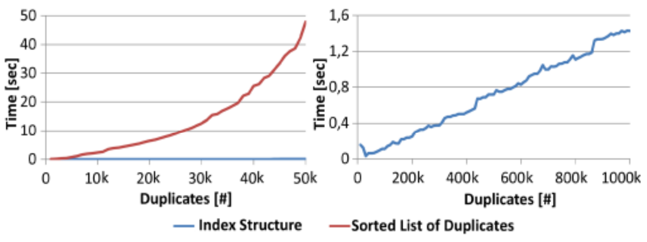


Figure 9. The incremental transitve closure overhead

The left chart shows that the proposed sorted lists of duplicates data structure does not scale well with the result set's size. However, the incremental transitive closure algorithm by Wallace and Kollias [17] scales linearly with the number of identified duplicates if we use an index structure on the identified duplicates. The measurements further show that the overhead of calculating the transitive closure is negligible: Identifying one million duplicates took more than 30 minutes, but calculating the transitive closure on them takes only 1.4 seconds.

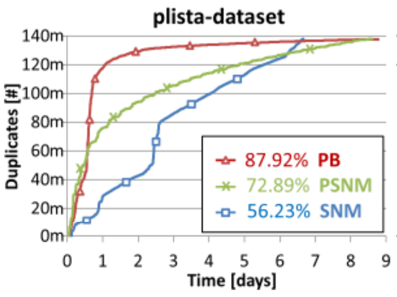
8.6 Examining a concrete use case

Progressive duplicate detection is an efficient and convenient solution for many data cleansing use cases. In cooperation with plista (www.plista.com), a company offering target-oriented online advertisement, we used our progressive algorithms to detect persona in web server log data. A persona is a user with a certain interest area. Hence, the same user is and should be reflected by different persona, if her interests differ. Compared to the number of entity duplicates in traditional data cleansing tasks, we expect many more persona duplicates in this dataset.

To arrange target-oriented advertisements, plista collects anonymized web log data for visitors of their customer's web pages. The huge amount of constantly growing data comprises information about user's software, geographic location, query terms, and categories, to mention only a few attributes. We refer to this dataset as the *plista dataset* [20]. For the task of finding persona, we consider a subset of the *IMPRESSION*-table comprising 100 million records and 63 attributes, which corresponds to 150 GB in total.

Although primarily used to create recommendations for advertisement, plista also analyzes the dataset to identify users. Currently, users are identified by their session ID – not recognizing different users that, for instance, share the same device or same users that maintain multiple sessions. To identify users more accurately, domain experts at plista defined a similarity measure for web log records that deduplicates personas. The similarity measure compares 17 of the 63 attributes by either edit-distance, numerical distance, or exact matching and returns a final similarity as the weighted sum of the individual similarities.

To run the persona detection, we use a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB DDR3-1600 RAM. Note that although the server provides 16 cores, the current implementations of all algorithms are single-threaded and, therefore, utilize only one core. Hence, *all* algorithms can further be improved by parallelization. The server's main memory of 128 GB can hold 15 million records of the given plista-dataset, which leads to seven partitions overall. Due to the size of the dataset and the high number of expected duplicates, we also increase the maximum window size to 50 for the SNM-approaches and the block size to 6 and maximum block range to 8 for the PB algorithm. The results of this experiment are shown in Figure 10.



$\omega(t)$	SNM	PB	PSNM
$min(\frac{1}{t^2}, 1)$	52%	81%	67%
$\begin{matrix} 1 & \text{if } t \leq L \\ 0 & \text{if } t > L \end{matrix}$	65%	96%	79%
$\frac{1}{ln(t+1)+1}$	46%	63%	64%

Figure 10. Duplicates found in the plista-dataset

The traditional Sorted Neighborhood Method takes almost seven days to finish the persona detection. Not only must the user wait this long for results, the algorithm also reserves significant server resources during these days. In combination with early termination, both progressive algorithms significantly reduce this effort. Although the two algorithms require more time to completely finish, they deliver almost same results in a much shorter time: PSNM identifies 71% and PB identifies 93% of all duplicates already in the first two days. So if we accept a slightly less complete result, we can run the deduplication in two instead of seven days.

With 56%, SNM exhibits an above average progressive performance. However, PSNM still outperforms this quality with 73% and PB with even 88%. These results are comparable to the results that we measured in Sec. 8.3 on smaller datasets using less memory. The reason for PB significantly outperforming PSNM on the plista dataset is that the dataset contains many duplicate clusters, which was foreseeable for the use case at hand. We also show the quality for other weighting functions $\omega(t)$ with $L = 1$ and t in days for this experiment: As the first two rank the results similar, the last function puts so much weight on the few very early results that PSNM is ranked highest here. So PSNM might be preferable in a pipeline-scenario.

In the analysis, we found out that the plista dataset contains about 135 million duplicate pairs (wrt. the expert's similarity measure definition of a persona). After merging all these duplicates, we ended up with 61.4 million distinct personas in the 100 million web log records. Among those, 55 million were singletons, i.e., had no duplicate. So each persona visited about 1.6 web-pages containing plista advertisement on average. Furthermore, the average size of a duplicate cluster (excluding the singletons) is 21, which corresponds to seven records for the same persona. So most personas visit only one web-page with plista advertisement (the singletons), but if a persona visits more than one page, then she visits seven pages on average. By further inspecting the identified personas, however, data mining specialists might discover more insights.

In summary, executing a full, traditional duplicate detection run on plista's massive amount of log data turned out to be extremely time and resource consuming. Using progressive duplicate detection techniques, on the contrary, renders this process feasible: As the result of the persona detection must not necessarily be complete, the progressive analysis can be stopped at any point in time and still maximizes the output.

9 Conclusion and Future Work

This paper introduced the *Progressive Sorted Neighborhood Method* and *Progressive Blocking*. Both algorithms increase the efficiency of duplicate detection for situations with limited execution time; they dynamically change the ranking of comparison candidates based on intermediate results to execute promising comparisons first and less promising comparisons later. To determine the performance gain of our algorithms, we proposed a novel quality measure for progressiveness that integrates seamlessly with existing measures. Using this measure, experiments showed that our approaches outperform the traditional SNM by up to 100% and related work by up to 30%.

For the construction of a fully progressive duplicate detection workflow, we proposed a progressive sorting method, *Magpie*, a progressive multi-pass execution model, *Attribute Concurrency*, and an incremental transitive closure algorithm. The adaptations AC-PSNM and AC-PB use multiple sort keys concurrently to interleave their progressive iterations. By analyzing intermediate results, both approaches dynamically rank the different sort keys at runtime, drastically easing the key selection problem.

In future work, we want to combine our progressive approaches with scalable approaches for duplicate detection to deliver results even faster. In particular, Kolb et al. introduced a two phase parallel SNM [21], which executes a traditional SNM on balanced, overlapping partitions. Here, we can instead use our PSNM to progressively find duplicates in parallel.

References

[1] S. E. Whang, D. Marmaros, and H. Garcia-Molina, "Pay-as-you-go entity resolution," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 25, no. 5, 2012.

[2] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 19, no. 1, 2007.

[3] F. Naumann and M. Herschel, *An Introduction to Duplicate Detection.* Morgan & Claypool, 2010.

[4] H. B. Newcombe and J. M. Kennedy, "Record linkage: making maximum use of the discriminating power of identifying information," *Communications of the ACM*, vol. 5, no. 11, 1962.

[5] 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, no. 1, 1998.

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

[7] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller, "Framework for evaluating clustering algorithms in duplicate detection," in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2009.

[8] O. Hassanzadeh and R. J. Miller, "Creating probabilistic databases from duplicated data," *VLDB Journal*, vol. 18, no. 5, 2009.

[9] U. Draisbach, F. Naumann, S. Szott, and O. Wonneberg, "Adaptive windows for duplicate detection," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2012.

[10] S. Yan, D. Lee, M. yen Kan, and C. L. Giles, "Adaptive sorted neighborhood methods for efficient record linkage," in *International Conference on Digital Libraries (ICDL)*, 2007.

[11] J. Madhavan, S. R. Jeffery, S. Cohen, X. Dong, D. Ko, C. Yu, and A. Halevy, "Web-scale data integration: You can only afford to pay as you go," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2007.

[12] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy, "Pay-as-you-go user feedback for dataspace systems," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2008.

[13] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2009.

[14] P. Indyk, "A small approximately min-wise independent family of hash functions," in *Proceedings of the Symposium on Discrete Algorithms*, 1999.

[15] U. Draisbach and F. Naumann, "A generalization of blocking and windowing algorithms for duplicate detection." in *International Conference on Data and Knowledge Engineering (ICDKE)*, 2011.

[16] W. Jr., "A modification of warshall's algorithm for the transitive closure of binary relations," *Communications of the ACM*, vol. 18, no. 4, 1975.

[17] M. Wallace and S. Kollias, "Computationally efficient incremental transitive closure of sparse fuzzy binary relations," in *IEEE International Conference on Fuzzy Systems*, 2004.

[18] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, 1964.

[19] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 9, 2012.

[20] B. Kille, F. Hopfgartner, T. Brodt, and T. Heintz, "The Plista dataset," in *Proceedings of the International Workshop and Challenge on News Recommender Systems*, 2013.

[21] L. Kolb, A. Thor, and E. Rahm, "Parallel sorted neighborhood blocking with mapreduce," in *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, 2011.

Thorsten Papenbrock studied computer science at the Hasso Plattner Institute (HPI) in Potsdam and received his Master degree in 2013. He is continuing his studies as PhD candidate in the Information Systems group at HPI. His primary interests are efficient data cleansing and data profiling techniques.

Arvid Heise studied computer science at the Hasso Plattner Institute (HPI) in Potsdam and received his Master degree in 2010. As a developer of the parallel data processing system Stratosphere, he researches on distributed data cleansing operators during his PhD studies in the Information Systems group at HPI.

Felix Naumann studied mathematics at the University of Technology in Berlin. After receiving his diploma (MA) in 1997 he joined the graduate school "Distributed Information Systems" at Humboldt University of Berlin, completing his PhD thesis in 2000. In 2001 and 2002 he worked at the IBM Almaden Research Center on topics around data integration. From 2003 until 2006 he was assistant professor for information integration at the Humboldt-University of Berlin. Since then he is full professor at the University of Potsdam in Germany, heading the Information Systems group at the Hasso Plattner Institute (HPI).