Parallel and Multiple Data Distributed Process using Concurrent 
Progressive Duplicate Detection 
Mahalakshmi.B¹, Hemapriyadharshini. R², Divyadharshini. G³, Ganavel. S⁴ 
Department of Computer Science and Engineering 
K. S. R. College of Technology, India 

Abstract: 
Efficient management of larger data set is an important factor in realizing the Database management vision. Maintaining the quality of the data set over a short period of time becomes increasingly difficult. In this paper, multiple representations of entities which cause data inconsistencies have been removed. For this purpose two basic approaches have been followed: Progressive Sorted Neighborhood Method which performs deduplication best on small and almost clean dataset and Progressive Blocking which is suitable for large and very dirty dataset. Two dynamic progressive duplicate detection algorithms expose different strength and outperform current approaches. However these methods have certain limitations over different attributes. In order to overcome this issue, concurrent progressive approach has been proposed for the multi-pass method along with incremental transitive closure algorithm to form the first complete progressive duplicate detection workflow. This method quickly filters out duplicate data from multiple resources in a concurrent manner. In this way deduplication results in an increased scalability with dynamic behavior. Quality measure has been proposed for progressive duplicate detection to objectively rank the performance of different approaches. Their corresponding accuracy and time efficiency are shown in graphical representation. 

Keywords: Progressive Deduplication, Attribute concurrent Progressive Sorted Neighborhood Method, Attribute concurrent Progressive Blocking 

1. INTRODUCTION 
Data mining, or knowledge discovery, is the computer-assisted process of analyzing enormous sets of data and then extracting the meaning of the data. Data mining tools can 1) answer business questions that traditionally were too time consuming to resolve. 2) They scour databases for hidden patterns 3) predictive information that experts may miss because it lies outside their expectations. Maintaining the large data possess a difficult task due to repeated data, integration of data from multiple resources. Hence, 1) to avoid duplicate entities deduplication method have been proposed in a progressive manner 2) to avoid data integration problem concurrent and parallel approach has been introduced to improve the overall processing of removing duplicates. 

2. RELATED WORK 
Ashwini. V, Lithin. K et al [1] proposed that the presence of duplicates create major flaw in database quality. To detect the duplicity with less time methods like Progressive Neighborhood and Progressive Blocking are used. Progressive Neighborhood method detect duplicate in a parallel approach. Progressive Blocking method works well on larger and dirty database. By these methods efficiency are doubled than the conventional methods. 

Ahmed K. Elmagarmid et al [2] proposed that duplicate records do not share common key and moreover they contain errors that make duplicate matching a difficult task. Hence similarity metrics have been proposed to detect the similar field entities along with the extensive set of duplicate detection algorithm. It has also been covered multiple techniques for improving the efficiency and scalability of approximate duplicate detection algorithm. 

A.Y Halevy et al [3] suggested that in order to manage wide variety of database management problem query should to manage efficiently. Hence a materialized view is needed to avoid problem in answering queries. In query optimization finding a rewriting of a query using a set of materialized view can yield more efficient query execution plan. As a result query execution plan can access storage amounts to solve the problem on answering queries using views and synthesize desperate work into coherent framework. This avoids in integrating data. 

Praveen Kumar. B, Priyanka. M, Subashini. S and Radhika. C et al [4] proposed a concurrent progressive approach for detecting duplicate. This method implements ranking methodology to find the closest similar records. To maintain the data quality, duplicate detection job has been scheduled to check for duplicates for records that match certain criteria. 

C. Xiao, W. Wang and X. Lin et al [5] suggested that for primitive operation similarity join plays an important role. Hence proposed a set of top k similarity join by ranking the similarity. This results an extended application over large-scale dataset for data integration, deduplication, etc. 

V. Gotham, D. Madhuri and T. Thirumani et al [6] proposed an alternative way for deduplication by Progressive duplicate detection algorithm which significantly intensifies the efficiency
of discovering replicas if the execution time is inadequate. Hence, widespread test display that progressive algorithm can doubles the efficiency over time of traditional duplicate detection.

U. Draisbach and F. Naumann et al [7] suggested that for finding duplicate elements cost of comparison is higher in traditional method. Hence, proposed two approaches: Blocking method which partitions the records into disjoint subset and Windowing method which slide a window over the sorted windows and compare records within the window. By generalizing the two approaches proposed sorted block in several variants.

Szott. S and Wonneberg O et al [8] proposed an alternative way from Sorted Neighborhood Method for finding duplicates by Adaptive Window technique in which window size can be adapted according to the range of similarity comparison of records. This method is suitable for both large and small dataset.

E. Rahm and A. Philip et al [9] suggested that schema matching is a basic problem in many database application domains. Hence proposed a taxonomy that covers many existing domains. Based on our classification we review some previous match implementations thereby indicating which part of the solution space they cover. Thus this approach will be useful for many schemas matching application.

Ivax P. Fellegi et al [10] proposed that a mathematical model is developed to provide a theoretical framework for a computer-oriented solution to the problem of recognizing those records in two files which represent identical persons, objects or events. A comparison is to be made between the recorded characteristics and values in two records and a decision made as to whether or not the members of the comparison-pair represent the same person or event, or whether there is insufficient evidence to justify either of these decisions at stipulated levels of error. This technique a Advances in electronic data processing equipment and techniques which make it appear technically and economically feasible to carry out the huge amount of operational work in comparing records between even medium-sized files.

L. Jin and C. Li et al [11] proposed an efficient approach to record linkage. Given two lists of records, the record-linkage problem consists of determining all pairs that are similar to each other, where the overall similarity between two records is defined based on domain-specific similarities over individual attributes constituting the record. For each attribute of records, it first maps values to a multidimensional Euclidean space that preserves domain-specific similarity.

Manju V J and Chinchu Krishna S et al [12] proposed Windowing method for deduplication process. One among the popular method is Sorted Neighborhood Method. But this method is not straight forward method over several variants. Hence a comparative approach has been made among traditional SNM and Progressive Sorted Method. As a result, PSNM yields better efficiency than SNM.

M. Halkidi and Y. Batistakis et al [13] suggested that Cluster analysis is made to identify similar group of objects in order to discover distribution of pattern and interesting correlations in a large dataset. Hence a survey is made to compare various clustering algorithm. As a result clustering validity measure is proposed for each algorithm and represents important issues in clustering process.

O Benjelloun, D Menestrina and J Widom et al [14] proposed on considering the entity resolution (ER) problem in which records determined to represent the same real-world entity are successively located and merged. It has been formalized the generic ER problem, treating the functions for comparing and merging records as black-boxes, which permits expressive and extensible ER solutions.

O Hassanzadeh, F Chiang and H Chul Lee et al [15] suggested that in order to detect duplicates, Entity Resolution or Record linkage is used as a part of data cleaning process. For this purpose Stringer system is used as an evaluation framework for understanding the barrier towards truly scalable duplicate detection. Hence proposed an approximate join technique to evaluate the various clustering algorithms.

Peter Christen et al [16] proposed that record linkage is the process of matching records from several databases that refer to the same entities. When applied on a single database, this process is known as deduplication. This paper aimed at reducing the number of record pairs to be compared in the matching process by removing obvious non matching pairs, while at the same time maintaining high matching quality.

R. Ananthakrishna, S. Chaudhuri, and V. Ganti et al [17] suggested that for deduplication previous domain provide independent solutions to this problem relied on standard textual similarity functions between multi-attribute tuples. However, such approaches result in large numbers of false positives if we want to identify domain-specific abbreviations and conventions. Hence proposed an algorithm for eliminating duplicates in dimensional tables in a data warehouse, which are usually associated with hierarchies for high quality and scalable duplicate detection algorithm.

R J Miller et al [18] proposed alternative approach is to keep duplicates when the correct cleaning strategy is not certain, and utilize an efficient probabilistic query answering technique to return query results along with probabilities of each answer being correct. In this paper, focused on presenting a flexible modular framework for scalable creating a probabilistic database out of a dirty relation of duplicated data and overview the challenges raised in utilizing this framework for large relations of string data.

Whang S.E, Marmaros D and Garcia-Molina H et al [19] proposed a potential approach to identify entity resolution problem in a larger dataset based on hints, this maximize the progress of ER within limited amount of time. By constructing the hints efficiently the maximum number of matching pairs can be identified in a less amount of work.

X Wang, Angeles Maria del Pilar and M Lachlan et al [20] suggested that Data integration is the process of extracting and merging data from multiple heterogeneous sources to be loaded
into an integrated information resource. Hence this project proposed the development of a Data Quality Manager to establish communication between the process of integration of information, the user and the application, to deal with data inconsistencies and information integration.

3. THE PROGRESSIVE SNM

Method is based on the traditional Sorted Neighborhood Method. 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 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. 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 Algorithm 1 Progressive Sorted Neighborhood Require: dataset reference D, sorting key K, window size W, enlargement interval size I, number of records N sorting key K defines the attribute or attribute combination that should be used in the sorting step. 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. Furthermore, PSNM integrates a progressive sorting phase (MagpieSort) and can progressively process significantly larger datasets.

Algorithm 1

Progressive Sorted Neighborhood method

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 ← calcPartitionSize (D)
3: pNum ← [n/pSize-W+1]
4: array order size N as integer
5: array recs size pSize as record
6: order ← sort Progressive (D, K, I, pSize, pNum)
7: for currentI ← 2 to W/1 do
8: for current ← 1 to pNum do
9: recs ← load Partition (D, currentP)
10: for dist ∈ range (currentI, W) do
11: for i ← 0 to [recs]-dist do
12: pair ← (recs[i], recs [i+dist])
13: if compare (pair) then
14: emit (pair)
15: look Ahead (pair)

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 disintegrations 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 N 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 it constitutes a trade-off parameter that balances progressiveness and overall runtime.

http://ijesc.org/
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 load Partition (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.

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-ahead or in a following progressive iteration, PSNM maintains all executed comparisons in a temporary data structure. This behavior is implemented by the look Ahead (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 execute some first comparisons. MagpieSort is a naïve sorting algorithm that works similar to Selection Sort. 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 MM from the buffer. A record that has been emitted once will not be emitted again. So for _N_ records, the algorithm terminates after _N/x_ 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. 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

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. 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.

![Figure 2: PB in block comparison matrix](http://ijesc.org/)
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 influence 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 data block 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 hblockNr1; blockNr2; duplicates Per Comparison. 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.

Algorithm 2 Progressive Blocking

Requires: dataset reference D, key attribute K, maximum block range R, block size S and number of records N

1: procedure PB (D, K, R, S, N)
2: pSize ← calcPartitionSize (D)
3: bPerP ← [pSize/S]
4: bNum ← [N/S]
5: pNum ← [bNum/bPerP]
6: array order size N as Integer
7: array blocks size bPerP as (Integer, Record [ ])
8: priority queue bPairs as (Integer, Integer, Integer)
9: bPairs ← {<1, 1, _>, <bNum, bNum>}
10: order ← sort Progressive (D, K, S, bPerP, bPairs)
11: for i ← 0 to pNum – 1 do
12: pBPs ← get (bPairs, i.bPerP, (i+1).bPairs)
13: blocks ← loadBlocks (pBPs, S, Order)
14: compare (blocks, pBPs, order)
15: while bPairs is not empty do
16: pBPs ← {}
17: bestBPs ← take Best ([bPerP/4], bPairs, R)
18: for bestBP ∈ bestBPs do
20: pBPs ← pBPsU extend (bestBP)
21: blocks ← loadBlocks (pBPs, S, order)
22: compare (blocks, pBPs, order)
23: bPairs ← bPairsUpBPs
24: procedure compare (blocks, pBPs, Order)
25: for pBPeP bBPs do
26: (dPairs, cNum) ← comp (pBPs, blocks, and order)
27: emit (dPairs)

The 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

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. 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 \( R = \lfloor \text{window size/S} \rfloor \) 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 (ACP-PSNM), which is the progressive implementation of the multi-pass method for the PSNM algorithm, and the second algorithm is the Attribute Concurrent PB (ACP-PB), which is the corresponding implementation for the PB algorithm.

Algorithm 3 Attribute Concurrent PSNM
\begin{verbatim}
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 ← calcPartitionSize (D)
3: pNum ← dN/(pSize−W + 1)e
4: array orders dimension |Ks| x N as Integer
5: array windows size |Ks| as Integer
6: array dCounts size |Ks| as Integer
7: for k ← 0 to |Ks| −1 do
8: (orders[k], dCounts[k]) ← sort Progressive (D, I, Ks[k], pSize, pNum)
9: windows[k] ← 2
10: while \( \exists w \in \text{windows}: w < W \) do
11: k ← findBestKey (dCounts, windows)
12: windows[k] ← windows[k] + 1
13: dPairs ← process (D, I, N, orders[k], windows[k], pSize, pNum)
14: dCounts[k] ← |dPairs|
\end{verbatim}

5.1 Attribute Concurrent PSNM
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. Algorithm 4 Attribute Concurrent PB Require: dataset reference D, sorting keys Ks, maximum block range R, block size S and record number N

Algorithm 4 Attribute Concurrent PB
\begin{verbatim}
Require: dataset reference D, sorting keys Ks, maximum block range R, maximum block size S and record number N
1: procedure AC-PB (D, Ks, R, S, N)
2: pSize ← calcPartitionSize (D)
3: bPerP ← [bSize/S]
4: bNum ← [N/S]
5: pNum ← [bNum/bPerP]
6: array order dimension |Ks| x N as integer
7: array blocks size bPerP as (Integer, record [])
8: list bPairs as (Integer, Integer, Integer)
9: for k ← 0 to |Ks| −1 do
10: pairs ← \{ (1, 1, ..., k) ... (bNum, bNum, k) \}
11: orders[k] ← sort Progressive (D, Ks[k], S, bPerP, pairs)
12: bPairs ← bPairs U pairs
\end{verbatim}
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.

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. 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 ACPB merges all block pairs based on any order in this list. To match a block pair with its corresponding order, ACPB 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. RESULTS AND DISCUSSIONS

1. ADDITION OF VOTER’S DETAIL

The Figure 8.1 shows the voter’s record addition. The figure contains voter id, voter name, voter address, voter phone, Gender, Age, and Adhaar Number with unique id details.

Figure 1. Record addition

The records are added for the given details about the voters. The records contain the personal details of the voter with the unique identification number. The records are saved in ‘Voter’ table. This voter information dataset contains the duplicate data which will be applied for the duplicate removal process using the progressive shorted neighborhood method and progressive blocking method.

2. ATTRIBUTE SETTING AND PARAMETR SETTING FOR PSNM/PB AND CPSNM/CPB

Figure 2. Attribute Setting

Attribute selection is a process used to set based on which, field needed for the duplicates are detected. This consists of reference data that contain the corresponding database to be referred and the field which act as a key for duplicate detection process.
This module consists of reference data in which the corresponding database can be chosen along with the desired field such as voter ID, name, address, etc. This process possess variation among the progressive approach and concurrent approach. In progressive approach only single field can be selected as a key for duplicate detection mechanism. But in concurrent approach more than one field can be chosen as a key for duplicate detection. In parameter setting, input for the Algorithm PSNM is selected; this parameter can be set to an optimistically high default value.

3. PROGRESSIVE SORTED NEIGHBORHOOD METHOD

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum

7. PROGRESSIVE BLOCKING

In PB at first calculates the number of records per partition pSize. 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. PB then defines three main data structures for duplicate removal process

- The order-array: Stores the ordered list of record IDs
- The blocks-array: Holds the current partition of blocked records.
- The bPairs-list: Stores all recently evaluated block pairs.

Data loads partition wise from disk and starts progressively extending the most promising block pairs. The evaluated best pairs are stored in a list as a priority queue. Then during each iteration among the best pair from the list, the block pairs which contains highest duplicate density are reported

8. CONCURRENT PSNM

The PSNM algorithm calculates maximum number of records that fit into the memory. Initially the algorithm calculates necessary partition and hold actual record of current partition. Sort the dataset according to the key fixed during the parameter setting process. Number of records for comparison has been linearly increase window size to maximum
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 pre-calculates the sorting for each key attribute. The pre-calculation 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.

9. CONCURRENT PB

AC-PB instead of scheduling progressive iterations of different keys, it directly schedules the bPair-comparisons of all keys. AC-PB first calculates the initial block pairs and their duplicate counts for all keys. It then 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.

10. VIEW OF VOTER DETAILS

The changes in voter detail such as record addition, deletion and updation in the field values are reflected in the administrator database. The data of voters are stored in the form of sequential manner in the database so that the duplicate records cannot be overridden. Due to this functionality the duplicate records can be easily identified.

11. PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th>S. N O</th>
<th>No. of Dataset</th>
<th>PSNM Duplicate Size (Count)</th>
<th>PSNM Time (ms)</th>
<th>CC PSNM Duplicate Size (Count)</th>
<th>CC-PSNM Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>8</td>
<td>0.01</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>14</td>
<td>0.01</td>
<td>18</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>25</td>
<td>0.02</td>
<td>31</td>
<td>0.01</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>37</td>
<td>0.02</td>
<td>48</td>
<td>0.02</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>48</td>
<td>0.03</td>
<td>59</td>
<td>0.02</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>59</td>
<td>0.03</td>
<td>65</td>
<td>0.03</td>
</tr>
<tr>
<td>7</td>
<td>175</td>
<td>66</td>
<td>0.04</td>
<td>71</td>
<td>0.03</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>72</td>
<td>0.04</td>
<td>83</td>
<td>0.04</td>
</tr>
</tbody>
</table>

In this paper, we have issued a set of comparison between Progressive Sorted Neighborhood method and Attribute Concurrent Progressive Sorted Neighborhood Method. Based on the performance evaluation of PSNM and concurrent PSNM it has been clearly proved that concurrent PSNM produces enhanced time efficiency than the progressive approach of duplicate detection.

<table>
<thead>
<tr>
<th>S. N O</th>
<th>No. of Block Size</th>
<th>PB Duplicate Size (Count)</th>
<th>PB Time (ms)</th>
<th>CC-PB Duplicate Size (Count)</th>
<th>CC-PB Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0.004</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0.008</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>15</td>
<td>0</td>
<td>28</td>
<td>0.012</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>27</td>
<td>0</td>
<td>36</td>
<td>0.025</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>38</td>
<td>0</td>
<td>42</td>
<td>0.032</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>49</td>
<td>0</td>
<td>53</td>
<td>0.037</td>
</tr>
<tr>
<td>7</td>
<td>175</td>
<td>56</td>
<td>0</td>
<td>62</td>
<td>0.039</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>60</td>
<td>0.1</td>
<td>70</td>
<td>0.045</td>
</tr>
</tbody>
</table>

On comparing the performance evaluation to the Progressive approach of blocking method and concurrent progressive blocking method it has been resulted that output target time t needed for CC-PB is smaller than the traditional PB method. Apart from the time efficiency it is resulted with more number of promising records within the given stipulated time. The respective time efficiency has been calculated by finding the difference between the start and end time of program execution.
12. EXPERIMENTAL RESULT AND DISCUSSIONS

Comparison between PSNM and CC-PSNM Algorithm [Duplicate Record Analysis]

The experimental comparison provides a view towards duplicate record analysis and time analysis of the given data set. I.e., Voter dataset. In duplicate record analysis, the X axis indicates the number of records that took part during execution and Y axis indicates the number of duplicate records that are detected during the execution of the process. From the result concurrent approach has been proved to better in finding large set of duplicates. This is due to adaptive approach which can dynamically adjust the window size based on identified intermediate result. In time analysis of duplicate detection, X axis represents the number of records that are taken under consideration and Y axis represent the time taken to find out the duplicate entities. From the obtained observation it has been proved that the target time t for concurrent approach is typically lower than the traditional progressive approach.

13. CONCLUSION

Need of efficient duplicate detection algorithm becomes important, as the data set increases in a large volume. Through this project, traditional Progressive algorithm has been experimentally compared with the concurrent algorithm. The experiment shows that CC-PSNM method and CC-PB algorithm produces greater efficiency than the PSNM and PB algorithm in terms of time and number of most matching pairs of duplicate records founded during execution. Concurrent progressive methods can reduce the average time after which an arbitrary duplicate is found and thereby detects duplicates earlier. It also finds more results because of comparison of entities with several attributes in a simultaneous manner.

14. REFERENCES


