An Incremental Clustering Scheme for Duplicate Detection in Large Databases

Eugenio Cesario, Francesco Folino, Giuseppe Manco, Luigi Pontieri

RT-ICAR-CS-05-01 **Gennaio 2005**

An Incremental Clustering Scheme for Duplicate Detection in Large Databases

Eugenio Cesario, Francesco Folino, Giuseppe Manco, Luigi Pontieri

ICAR-CNR
Via Bucci 41c, I87036 Rende (CS), Italy
{cesario,ffolino,manco,pontieri}@icar.cnr.it

Abstract. We propose an incremental algorithm for clustering duplicate tuples in large databases, which allows to assign any new tuple t to the cluster containing the database tuples which are most similar to t (and hence likely refer to the same real-world entity t is associated with). The core of the approach is a hash-based indexing technique that tends to assign highly similar objects to the same buckets. Empirical evaluation proves that the proposed method allows to gain considerable efficiency improvement over a state-of-art index structure for searches in metric spaces.

1 Introduction

Recognizing similarities in large collections of data is a major issue in the context of information integration systems. An important challenge in such a setting is to discover and properly manage duplicate tuples, i.e., syntactically different tuples which are actually identical from a semantical viewpoint, for they referring to the same real-world entity. The problem is particularly challenging when information on object identity is carried by textual fields, which are usually subject to various kinds of heterogeneity and mismatches across different information sources.

In the literature, this problem was addressed by different research communities, under a variety of names (such as, e.g., Merge/Purge, Record Linkage, Deduplication, Entity-Name Matching, Object Identification). Most of the proposed approaches mainly attempt to devise effective matching or clustering methods for recognizing duplicated tuples, based on some similarity functions (see, e.g., [1–7, 1]), while paying minor attention to scalability. However, efficiency and scalability issues do play a predominant role in many application contexts where large data volumes are involved, specially when the object-identification task is part of an interactive application, requiring short response times. In such a case, any approach requiring quadratic time in the database size or producing many random disk accesses ends up being unsuitable.

In this paper our main objective is the analysis of techniques for duplicate detection in large databases. We approach the problem from a *clustering* perspective: given a set of tuples, our goal is to recognize subsets of tuples that, with a high level of certainty, correspond to the same real-world entity. Three main features make the problem at hand significantly different from previous approaches: *(i)*tuples are represented as (small) sequences of tokens, where the set of possible tokens is high; *(ii)* the number of clusters is too high to allow the adoption of traditional clustering techniques, and *(iii)* the streaming (constantly increasing) nature of the data imposes linear-time algorithms for clustering.

In particular, in sec. 2, we first formalize the discovery of duplicate objects as a specific clustering problem. Then, sec. 3 illustrates an efficient technique that is able to discover all clusters containing duplicate tuples in an incremental way. The core of the approach is an effective indexing technique which, for any newly arrived tuple t, efficiently retrieves the tuples in the database which are most similar t (and which likely refer to the same real-world entity t is associated with), based on a hashing technique assigning highly similar objects to the same buckets. Finally, in sec. 5, we show experimental results demonstrating that the hashing-based method allows to obtain considerable improvement in efficiency w.r.t. a state-of-art indexing approach [8], which allow for searching in general metric spaces.

2 Problem Statement

In the following we introduce some basic notation and preliminary definitions, which will be used in the rest of the paper. An item domain $\mathcal{M} = \{a_1, a_2, \ldots, a_m\}$ is a collection of items. In our context m can be very large, or it may even be $m = \infty$. Typically, $\mathcal{M}$ represents the set of all possible strings available from a given alphabet. Moreover, we assume that $\mathcal{M}$ is associated with an ordering relationship $\prec_{\mathcal{M}}$ and a distance function $dist_{\mathcal{M}}(\cdot,\cdot) : \mathcal{M} \times \mathcal{M} \mapsto [0,1]$, expressing the degree of dissimilarity between two generic items a_i and a_j.

A *descriptor* $R = \{A_1, \ldots, A_n\}$ is a set of labels. A descriptor corresponds to a database schema, with the simplification that each attribute label A_i, has the same domain $\mathcal{M}$ associated.[1] Then, given a descriptor $R = \{A_1, \ldots, A_n\}$, a tuple τ_R is defined as $\{A_1 : t_1; A_2 : t_2; \ldots; A_n : t_n\}$, where for each $i = 1..n$ $t_i \subseteq \mathcal{M}$, itemset t_i is called *field itemset* of τ_R for attribute A_i, and is denoted by $\tau_R.A_i$. We shall omit the subscript denoting the related descriptor, when it be clear from the context.

Moreover, a distance function $dist(\tau^i_{R_i}, \tau^j_{R_j}) \in [0,1]$ can be defined for comparing two any tuples $\tau^i_{R_i}$ and $\tau^j_{R_j}$, by suitably combining the distance values computed through $dist_{\mathcal{M}}$ on the values of matching fields. Finally, a database is a set of tuples $\mathcal{DB} = \{\tau^1_{R_1}, \ldots, \tau^N_{R_N}\}$, such that $\tau^i_{R_i}$ is a tuple for R_i.

The *Entity Resolution* problem can be formally stated as the problem of transforming a database $\mathcal{DB}$ into a new database $\mathcal{DB}'$, such that the following conditions hold:

[1] This is justified by the fact that we aim at reconstructing semantic similarities from a syntactic viewpoint only.

1. **(Schema reconciliation)** There exist a descriptor R such that, for each tuple $\tau^i_{R_i} \in \mathcal{DB}$, a new tuple $\tau^i_R \in \mathcal{DB}''$ can be defined, with the same contents of $\tau^i_{R_i}$ restructured according to R.
2. **(Data reconciliation)** Given a dissimilarity function $dist(\tau^i_{R_i}, \tau^j_{R_j})$ between two generic tuples, there exists a partition of $\mathcal{DB}$" into groups $\mathcal{P} = \{C_1, \ldots, C_k\}$ such that

$$\sum_{u=1}^{k} \sum_{\tau^i_R, \tau^j_R \in C_u} dist(\tau^i_R, \tau^j_R)$$

is minimized, and

$$\sum_{u=1}^{k} \sum_{v=1, v \neq u}^{k} \sum_{\tau^i_R \in C_u, \tau^j_R \in C_v} dist(\tau^i_R, \tau^j_R)$$

is maximized.
3. **(Identity definition)** Given a partition $\mathcal{P} = \{C_1, \ldots, C_k\}$ of $\mathcal{DB}$", for each C_i there exists a tuple $\overline{\tau}^i_R$ such that

$$\overline{\tau}^i_R = \text{argmin}_{\tau_R} \sum_{\tau^j_R \in C_i} dist(\tau_R, \tau^j_R)$$

and $\mathcal{DB}' = \{\overline{\tau}^1_R, \ldots, \overline{\tau}^k_R\}$.

In the rest of the paper we focus on the *Data Reconciliation* problem: assuming that a database $\mathcal{DB}$ is available which contains only tuples defined according a fixed descriptor, we aim at grouping them in order to recognize duplicate tuples. In practice, this is essentially a clustering problem, but it is formulated in a specific situation, as the clusters to be discovered actually correspond to all the distinct real-world entities which are represented in $\mathcal{DB}$, in a redundant and inconsistent manner. Indeed, a basic assumption characterizing the framework is that there are several pairs of tuples in $\mathcal{DB}$, which are quite dissimilar from each other. This can be formalized by assuming that the size of the set $\{\langle\tau^i_R, \tau^j_R\rangle \mid dist(\tau^i_R, \tau^j_R) \simeq 1\}$ is $O(N^2)$. Thus, we can expect the number k of clusters to be very high – typically, $O(N)$. Moreover, we intend to cope with the clustering problem in an incremental setting, where a new database $\mathcal{DB}_\Delta$ must be integrated with a previously reconciled one $\mathcal{DB}$. Practically speaking, the cost of clustering tuples in $\mathcal{DB}_\Delta$ must be (almost) independent of the size N of $\mathcal{DB}$.

3 A Clustering Approach to Data Reconciliation

The key issue in the problem described above is the capability of detecting cluster membership for an element τ_R by means of a minimal number of comparisons. This can be achieved by exploiting a proper k-NN technique, in which the k nearest neighbors of τ_R are efficiently extracted from the given database. Algorithm 1 summarizes such a clustering procedure, which is parametric with respect to the distance function used for comparing two any tuples. Notably, the clustering method is defined in an incremental way, for it allowing to integrate a new set of tuples into a previously computed partition. Indeed, besides the set of new tuples $\mathcal{DB}_\Delta$, the algorithm receives a a database $\mathcal{DB}$ and an associated partition $\mathcal{P}$. As a result, it will produce a new partition $\mathcal{P}'$ of $\mathcal{DB} \cup \mathcal{DB}_\Delta$, obtained by adapting $\mathcal{P}$ with the tuples from $\mathcal{DB}_\Delta$. To this purpose, each tuple in $\mathcal{DB}_\Delta$ is associated with a cluster in $\mathcal{P}$, detected through a sort of *nearest-neighbor* classification scheme. The basic intuition in the proposed approach is that, since the number of clusters is high (typically $O(N)$), then it suffices to compare few "close" neighbors in order to obtain the appropriate cluster membership. In this respect, the approach is similar to the one proposed in [9], the main difference being that the nearest-neighbor classification scheme detects canopies "on-the-fly", i.e., without building them explicitly. The two remaining input parameters k and δ are meant to rule such a classification task: k is the maximal number of neighbors to be considered when assigning τ to a cluster, and δ is a threshold representing an upper bound for the distance between τ and any of such neighbors.

```
GENERATE-CLUSTERS($\mathcal{P}$,$\mathcal{DB}_\Delta$,$k$,$\delta$)
 Output: A partition $\mathcal{P}'$ of $\mathcal{DB} \cup \mathcal{DB}_\Delta$;
 1:  $\mathcal{P}' \leftarrow \mathcal{P}$; $\mathcal{DB}' \leftarrow \mathcal{DB}$;
 2:  Let $\mathcal{P}' = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ and $\mathcal{DB}_\Delta = \{\tau^1, \ldots, \tau^n\}$;
 3:  for $i = 1 \ldots n$ do
 4:    $neighbors \leftarrow$ KNEARESTNEIGHBOR($\mathcal{DB}', \tau^i, k, \delta$);
 5:    $\mathcal{C}_j \leftarrow$ MOSTLIKELYCLASS($neighbors, \mathcal{P}'$);
 6:    $\mathcal{DB}' \leftarrow \mathcal{DB}' \cup \{\tau^i\}$;
 7:    if $\mathcal{C}_j = \emptyset$ then
 8:      create a new cluster $\mathcal{C}_{m+1} = \{\tau^i\}$;
 9:      $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathcal{C}_{m+1}\}$;
 10:   else
 11:     $\mathcal{C}_j \leftarrow \mathcal{C}_j \cup \{\tau^i\}$;
 12:     PROPAGATE($neighbors, \mathcal{P}'$);
 13:   end if
 14: end for
-------------------------------------------------------------------
PROPAGATE($S$,$\mathcal{P}$)
 P1: for all $\tau \in S$ do
 P2:   $neighbors \leftarrow$ KNEARESTNEIGHBOR($\mathcal{DB}, \tau, k$);
 P3:   $\mathcal{C} \leftarrow$ MOSTLIKELYCLASS($neighbors, \mathcal{P}$);
 P4:   if $\tau \notin \mathcal{C}$ then
 P5:     $\mathcal{C} \leftarrow \mathcal{C} \cup \{\tau\}$;
 P6:     $Propagate(neighbors, \mathcal{P})$;
 P7:   end if
 P8: end for
```

Fig. 1. Clustering algorithm

More in detail, for each tuple τ^i in $\mathcal{DB}$ to be clustered, the neighbors of τ^i are retrieved by means of procedure KNEARESTNEIGHBOR (line 4), which performs a search for the k most prominent neighbors with a bounded range δ and using

τ^i as query object. The cluster membership for τ^i is determined by calling the MostLikelyClass procedure (line 5), which selects a *most likely* cluster among the ones associated with the neighbors of τ^i. If τ^i is estimated not to belong to any existing cluster with a sufficient degree of certainty, τ^i is assigned to a newly generated cluster; otherwise, τ^i is added to the cluster identified. Such an estimation is carried out via a *voting* strategy, where for each neighbor τ^i of τ a contribution $\frac{1}{dist(\tau^i,\tau)}$ is added to the score of the cluster containing τ^i. Tuple τ is eventually assigned to the cluster that gets the highest score, provided that this exceeds a given threshold (0.5, in our usual setting).

Whenever the new tuple is assigned to a preexistent cluster, procedure Propagate, scans the neighbors of τ^i in order to possibly revise their cluster membership, as it could be affected by the insertion of τ^i. In particular, for each tuple τ' in the input set, the membership of τ' is estimated by using again procedure MostLikelyClass; in the case that the selected cluster does not coincide with the one which actually contains τ', the membership of τ' is modified and the procedure Propagate is recursively applied to the neighbors of τ. In principle, since this task might be iterated over each reassigned tuple, it could have a linear complexity w.r.t. the size of $\mathcal{DB}$. However, in a typical Entity Resolution setting, where clusters are quite distant from each others, such a propagation affects only a reduced number of tuples, and terminates in a low number of iterations.

The complexity of Algorithm 1, given the size N of $\mathcal{DB}$ and M of $\mathcal{DB}_\Delta$, depends on the three major tasks: the search for neighbors (line 4, having cost n), the voting procedure (line 5, having a cost proportional to k), and the propagation of cluster labels (line 12, having a cost proportional to n, according to the discussion above). Since they are performed for each tuple in $\mathcal{DB}_\Delta$, the overall complexity is $O(M(n+k))$. Since k is $O(1)$, it is the main contribution to the complexity of the clustering procedure is due to the kNearestNeighbor procedure. Therefore, the efforts towards computational savings must be addressed when designing a method for neighbor searches. Our main goal is doing this task by minimizing the number of accesses to the database, and avoiding the computation of all pair-wise distances. As discussed in next section, this goal can be achieved through a suitable indexing structure, allowing for retrieving neighbor in sub-linear time.

4 Optimizing the Search for Neighbors

As previously described, procedure kNearestNeighbor plays a fundamental role in the performance of the algorithm specified in fig. 1. In order to efficiently retrieve the neighbors of the current tuple, we can resort to an indexing scheme that can support the execution of similarity queries, and can be incrementally populated with new tuples.

To this purpose we introduce a novel hashing-based indexing scheme, which is expected to guarantee, in the average, the execution of neighbor searches in a time independent of the number of database tuples. The proposed index structure, called *Hash Index*, is represented as a pair $H = \langle FI, ES \rangle$, where:

– FI, referred to as *Feature Index*, is a component that associates any tuple with a set of codes, each of them referencing a bucket that contains similar tuples, for they sharing a relevant set of properly defined features;
– ES, referred to as *External Store*, is a component that is responsible for efficiently storing such buckets and ensuring an optimized usage of disk pages.

The basic idea underlying the definition of this indexing scheme is to map any tuple to a set of features, so that the similarity between two tuples can be evaluated by simply looking at their respective features. Under such a perspective, the role of the hashing technique is to maintain the association between tuples and the corresponding features, so that the neighbors of a tuple τ can be efficiently computed, by simply retrieving the tuples that appear in the same buckets which correspond to τ. For the sake of simplicity, we next illustrate how the approach can be defined in the case only a single attribute is used for comparing two tuples. In such a situation, the overall indexing structure has to be built only on the basis on the values of the selected attribute. Let τ be a tuple and t_i be the field itemset representing the value of τ for a chosen attribute A_i, i.e., $t_i = \tau.A_i$. In this context, we assume that the dissimilarity of two tuples τ and τ' is measured by taking the *Jaccard distance* between their associated field itemset for the attribute A_i, i.e. $dist(\tau, \tau') = dist_J(\tau.A_i, \tau'.A_i) = 1 - \frac{\tau.A_i \cap \tau'.A_i}{\tau.A_i \cup \tau'.A_i}$

A simple strategy for mapping a tuple τ to a set of features consists in extracting a set of "subkeys" from t_i, and associating each of these subkeys with a bucket, which will actually contain all the tuples sharing the given subkey with τ. More precisely, a *subkey* of t_i is any non-empty subset of t_i. Moreover, an *m-subkey* of t_i, with $0 < m \leq |t_i|$, is a subkey of t_i consisting of exactly m items. The indexing scheme works as follows. For a given tuple τ, a significant set of subkeys is extracted for an itemset t_i corresponding to the focusing attribute A_i. Then, each selected subkey s is exploited in searching for tuples sharing s in H. Figure 4 illustrates, in more details, how the hash-based indexing scheme can be exploited to support nearest-neighbor searches. The algorithm works on the basis of two main parameters: the number k of desired neighbors, and the maximum allowed distance δ from the query object τ. As noticed above, we restrict ourselves to a case where the distance between tuples is defined on just one attribute, which we indicate as an additional parameter A within the algorithm.

The algorithm uses two auxiliary structures, namely the set S of subkeys to be generated, and the set *neighbors* of neighbor tuples to return as an answer. For convenience, both structures are organized as priority queues: in particular, keys in S are sorted according to their size, whereas tuples in *neighbors* are sorted according to their distance from the query tuple τ. Moreover, we fix the capacity of *neighbors* to k, as we are only interested in the most prominent k neighbors.

Lines 4-17 specify how the set *neighbors* is filled. First, a subkey x is extracted (line 5), and the associated bucket is identified by employing the *FI.Search* method, which actually returns the logical address h of this bucket. If a bucket is associated with s, then lines 10-14 iteratively extract the tuples stored in the bucket (using *ES.Read*) and try to insert them within the *neighbors* structure. In particular, a tuple τ_e can be inserted within *neighbors* in two cases: *(a)* either

```
kNearestNeighbors($\mathcal{DB}$,$\tau$,$k$,$\delta$)
 1: Let $t = \tau.A$, where $A$ is the attribute indexed by $FI$ and $ES$;
 2: $S \leftarrow \{s \mid s$ is a $\delta$-significant subkey of $t\}$;
 3: $neighbors \leftarrow \emptyset$; $neighbors.capacity \leftarrow k$;
 4: while $S \neq \emptyset$ do
 5:   $x = S.Extract()$;
 6:   $h \leftarrow FI.Search(x)$;
 7:   if $(h = 0)$ then
 8:     $h \leftarrow FI.Insert(x)$;
 9:   else
10:     while $\tau_e = ES.Read(h)$ do
11:       if $neighbors.size < k$ or $dist(\tau_e, \tau) < neighbors.Max()$ then
12:         $neighbors.Insert(\tau_e, dist(\tau_e, \tau))$;
13:       end if
14:     end while
15:   end if
16:   $ES.Insert(\tau, h)$;
17: end while
18: return $neighbors$;
```

Fig. 2. The kNearestNeighbor procedure.

the size of *neighbors* does not exceed its capacity, or *(b) neighbors* capacity is k, but it contains an element whose distance from τ is higher than the distance between τ_e and τ. In both cases, τ_e is inserted in *neighbors* (line 12). If needed, the element with highest distance from τ is removed from *neighbors*, in order to make room for τ_e. As a side effect, the algorithm updates the index structure (i.e., FI and ES), in order to correctly refer to the novel tuple τ.

A major point is that both the indexing of τ and the retrieval of its neighbors are based on generating δ-*significant* subkeys of the field itemset t of τ corresponding to the attribute A used for indexing purposes. Clearly, the number of subkeys for a given itemset is exponential in the cardinality of the itemset itself. Therefore, the generation of subkeys for an itemset t should be bounded to produce only a minimal set of "significant" subkeys, which yet allow to retrieve all the itemsets whose distance from t is lower than a specified threshold δ. Notice that, if we restrict ourselves to 1-subkeys, then the indexing schemes behaves exactly like a inverted-lists index. However, inverted indexes do not guarantee that a minimal set of candidate tuples is retrieved. In particular, we say that a subkey s of a given itemset t is δ-*significant* if $dist_J(t, s) \leq \delta$. Clearly enough, all the δ-significant subkeys of t coincide with the m-subkeys of t, for $\lfloor |t| \times (1 - \delta) \rfloor \leq m \leq |t|$.

It is easy to see that any itemset t' such that $dist_J(t, t') \leq \delta$ must contain at least one of the δ-significant subkeys of t. Indeed, $dist_J(t, t') \leq \delta$ implies that:

$$|t \cap t'| \geq |t \cup t'| \times (1 - \delta) \geq |t| \times (1 - \delta)$$

thus requiring that $|t \cap t'|$ is a δ-significant subkey of t. In other words, searching for tuples that exhibit at least one of the δ-significant subkeys derived from a tuple τ represents a strategy for retrieving all the neighbors of τ without scanning the whole database. Notice that the above strategy guarantees the minimality

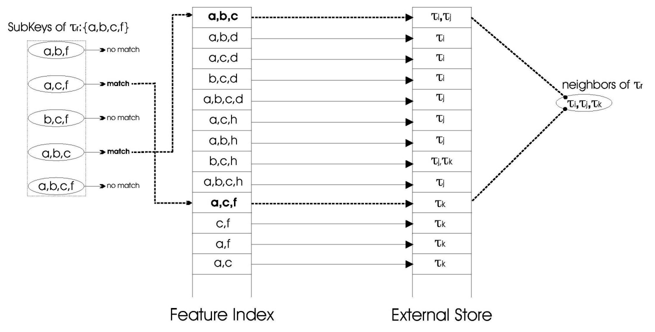


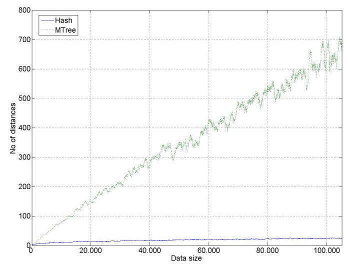
Fig. 3. Hash Index structure

condition: indeed, if $\tau_e.A$ and $\tau.A$ contain a sensible number of different items, then their δ-significant subkeys do not overlap. As a consequence, the probability that τ_e is retrieved for comparison with τ is low.

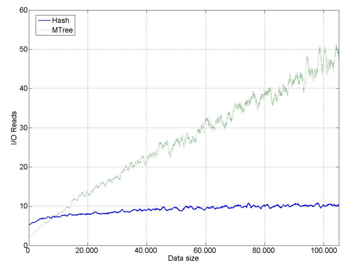
As an example, let us consider the index structure exemplified in fig. 3. The index contains the tuples $\tau^i = \{A : \{a, b, c, d\}, \ldots\}$, $\tau^j = \{A : \{a, b, c, h\}, \ldots\}$ and $\tau_k = \{A : \{a, c, f\}, \ldots\}$, which are inserted by fixing $\delta = 0.1$ as distance threshold. Observe that each subkey is associated with a bucket containing the tuples that exhibit the subkey: for example, the bucket linked by the $\{a, b, c\}$ subkey contains both τ^i and τ^j whereas the bucket linked by $\{a, c, f\}$ contains τ_k. When searching for the neighbors of a new tuple $\tau_r = \{A : \{a, b, c, f\}, \ldots\}$, the algorithm generates the δ-significant subkeys of τ_r, listed in the left of the figure. By querying the index structure with the subkeys, we obtain τ^i, τ^j (by means of $\{a, b, c\}$), and τ_k (by means of $\{a, c, f\}$).

5 Experimental Results

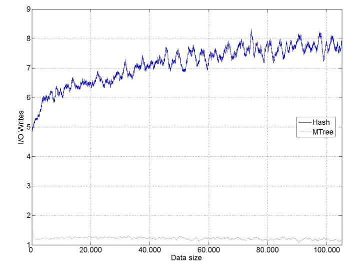
In this section the behavior of the proposed approach is studied with the help of an empirical evaluation, performed on both synthesized and real data. To this purpose, we equip the Generate-Clusters algorithm in fig. 1 with both the index structure illustrated in sec. 4 and a state-of-the-art indexing structure for proximity searches, named *M-Tree* [8]. Experiments are aimed at evaluating whether an appropriate number of clusters is generated, as well as whether the hashing-based indexing method allows for efficiency improvement w.r.t. the approach based on M-Trees. As a matter of fact, the *M-tree* is an index/storage structure, which looks like a n-ary tree, and is capable of indexing generic objects, provided that a suitable distance metric is defined for comparing them. The interesting point of the M-tree structure is that it represents a balanced hierarchical clustering structure, in which each cluster has a fixed size (related to the size of a page to be stored on disk). Thus, a similarity search can be accomplished by traversing the tree, and ignoring subtrees reputed uninteresting for



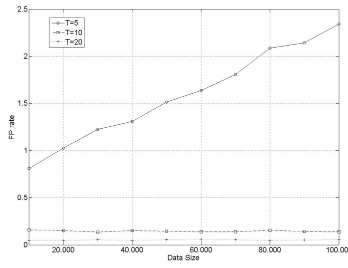
(a) No of distances on real data



(b) No of I/O Reads on real data



(c) No of I/O Writes on real data



(d) `FP-rate` vs. data size on synthesized data

Fig. 4. Summary of evaluation.

the search purpose. However, the benefits of this indexing structure are likely to degrade in a typical entity resolution scenario, where most of the internal nodes in the tree tend to correspond to a quite "heterogeneous" set of tuples, and hence a high number of levels, i.e., nearly linear in the number of distinct entities, is required to suitably partition the whole data set. Further details on M-Trees can be found in [8].

The results of the clustering algorithm in the two cases are analysedby considering three parameters:

– The number of distances computed during the selection of the neighbors. This is an effective evaluation parameter, which represents how many comparisons are performed during an insert/select operation and provides for an estimation of the CPU overhead.
– The number of disk pages read during the selection of objects. In principle, the hash-based approach could cause continuous leaps in the read operations, even if a small number of comparisons is needed.
– The number of disk pages written during the updating of the index. Since the index has to be incrementally maintained, it is important to evaluate the cost of such a maintenance.

The clustering algorithm was applied to the task of de-duplicating a real-world data set consisting of 105,140 tuples, representing information about customers of an Italian bank. Each tuple exhibit an average of 8 relevant neighbors, and, in general, distances between tuples exhibit high values.

The performance of the clustering algorithm with regards to the adoption of both the M-Tree and the Hash index structure are compared in fig. 4, where the efficiency measures have been averaged on 5,000 tuples. We used the M-tree implementation available on the Web, tuned by setting a node size of 4K and a *Random split* policy. In both techniques we fixed $\delta = 0.2$ and $k = 10$.

Fig. 4 shows the efficiency performances of the approaches w.r.t. the data size. In particular, the horizontal axis represents the portion of data examined so far. The evaluation of the incrementality of the approach can be made by observing whether the increase of the measure under consideration is bounded. This clearly does not happen for the number of distances and I/O Reads in the case of the M-tree approach which are roughly linear in the tree size. On the other side, performance metrics for the Hash are bounded by a constant factor, as expected. An opposite trend can be seen in the number of disk writes. This is mainly due to the different philosophy underlying the two structures. The number of disk writes in the Hash method depends on the number of δ-relevant subkeys. The larger is the set of subkeys, the higher is the number of writes needed in order to update the index. On the contrary, the M-tree is a balanced structure whose update causes (at most) a number of writes which is proportional to the depth of the tree. Indeed, in order to update its structure, the M-tree has to select the most appropriate position for the current tuple. After a suitable node has been selected the tree inserts the tuple in the node and writes it back to the disk. Overheads occur only when the insertion causes a node overflow: the node is split and the insertion is propagated upward in the tree.

Effectiveness can be evaluated by measuring the overlap between the expected number of clusters and the actual number of clusters computed. Clearly, the latter depends on the k and δ parameters, which directly influence an indexing scheme in performing neighbor searches. Hence, an important issue is whether the neighbors retrieved from the index suffice to perform a correct classification. To this purpose, we introduce the `FP-rate` measure, denoted by `FP-rate`(τ, DB), and representing the rate of (*False Positives*) tuples in DB which are retrieved by the indexing scheme but are not relevant to clustering τ, i.e., they should not belong to the same cluster as τ. A global `FP-rate` measure can be computed by averaging the `FP-rates` locally to each tuple τ^i and the pertaining portion of the database $\mathcal{DB}_{\Delta}$ (i.e., the dataset $DB_i = \{\tau_1, \ldots, \tau_{i-1}\}$).

Effectiveness was measured over synthesized data. Tuples were generated according to the following parameters: *(i)* the average size T of the itemsets associated with each attribute in the tuple; *(ii)* the size of $\mathcal{M}$; *(iii)* the number of clusters C; *(iv)* the number of tuples N. Each cluster was obtained by randomly choosing a subset of the items in $\mathcal{M}$. Then, each tuple in the cluster was generated by choosing items from the subset associated with the cluster. In order to guarantee the right degree of overlap, each new tuple was generated as a variation of a previously generated one.

We tested the hash based approach for increasing values of T. Figure 4(d) summarizes the values of `FP-rate`. As we can see, the rate is constant (fairly low) except in the case $T = 5$. The latter exhibits higher values mainly because the size of the itemsets contained in the tuples causes the generation of 1-subkeys, which cause a large number of false positives.

6 Conclusions and Future Works

In this paper, we addressed the problem of recognizing duplicate information, specifically focusing on scalability and incrementality issues. The core of the proposed approach is an incremental clustering algorithm, which aims at discovering clusters of duplicate tuples, based on a novel hashing-based indexing technique. An empirical analysis, both on synthesized and on real data, showed the validity of the approach, which does exhibit a considerable improvement in performance with respect to a traditional, state-of-the-art, index structure.

The proposed approach is expected to efficiently recognizing duplicate tuples which are similar according to set-based similarity functions. In particular, the effectiveness of the approach is based on the adoption of the Jaccard distance, which does not exploit the dissimilarity $dist_{\mathcal{M}}$ between tokens. Experiments on real data, show that the above dissimilarity works well in practical cases. Nevertheless, the approach could in principle fail in cases where a more refined dissimilarity functions is needed [3, 10].

To this purpose, we are currently studying a more refined key-generation technique for the algorithm in fig. 3, which allows a controlled level of approximation in the search for the nearest neighbors of a tuple. Extremely interesting to this context is the family of *Locality-Sensitive* hashing functions [11–13], which are guaranteed to assign two any objects to the same buckets with a probability which is directly related to their degree of mutual similarity. Preliminary results still show the suitability of the approach proposed in this paper, at the cost of a relatively low number of missed neighbors (`FN`-rate).

References

1. Monge, A.E., Elkan, C.P.: The field matching problem: Algorithms and applications. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. (1996) 267–270
2. Monge, A.E., Elkan, C.P.: An efficient domain-independent algorithm for detecting approximately duplicate database records. In: Proc. SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery. (1997) 23–29
3. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proc. IJCAI Workshop on Information Integration on the Web. (2003) 73–78

4. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2003) 39–48
5. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2002) 269–278
6. Cohen, W., Richman, J.: Learning to match and cluster entity names. In: Proc. ACM SIGIR Workshop on Mathematical/Formal Methods in Information Retrieval. (2001)
7. Cohen, W.W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2002) 475–480
8. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proc. of the 23rd Int. Conf. on Very Large Data Bases. (1997) 426–435
9. McCallum, A.K., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets with application to reference matching. In: Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2000) 169–178
10. Bilenko, M., et al.: Adaptive name matching in information integration. IEEE Intelligent Systems **18**(5) (2003) 16–23
11. Indyk, P., Motwani, R.: Approximate nearest neighbor - towards removing the curse of dimensionality. In: Proc. 30th Symposium on Theory of Computing. (1998) 604–613
12. Broder, A., Glassman, S., Manasse, M., Zweig, G.: Syntactic clustering on the web. In: Proc. 6th Int.WWW Conf. (1997) 1157–1166
13. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proc. 25th Conf. on Very Large Data Bases. (1999) 518–529