Effective Incremental Clustering for Duplicate Detection in Large Databases

Francesco Folino
ICAR-CNR
Via Bucci 41c
87036 Rende (CS) - Italy
ffolino@icar.cnr.it

Giuseppe Manco
ICAR-CNR
Via Bucci 41c
87036 Rende (CS) - Italy
manco@icar.cnr.it

Luigi Pontieri
ICAR-CNR
Via Bucci 41c
87036 Rende (CS) - Italy
pontieri@icar.cnr.it

Abstract

We propose an incremental algorithm for discovering clusters of duplicate tuples in large databases. The core of the approach is the usage of an indexing technique which, for any newly arrived tuple μ, allows to efficiently retrieve a set of tuples in the database which are mostly similar to μ, and which are likely to refer to the same real-world entity which is associated with μ. The proposed index is based on a hashing approach which tends to assign similar objects to the same buckets. Empirical and analytical evaluation demonstrates that the proposed approach achieves satisfactory efficiency results, at the cost of low accuracy loss.

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, in that they refer to the same real-world entity. There are several application scenarios involving this important task. A typical example consists in the reconciliation of demographic data sources in a data warehousing setting. Names and addresses can be stored in rather different formats, thus raising the need for an effective reconciliation strategy which could be crucial to effective decision making. In such cases the problem is the analysis of a (typically large) volume of small strings, in order to reconstruct the semantic information on the basis of the few syntactic data available. By assuming that each possible string represents a dimension along which the information contained in a tuple can projected, tuples represent a small informative content in a high-dimensional space. This issue has given rise to a large body of work in different research communities, and under a variety of names (e.g., De-duplication [15], Merge/Purge [10], Entity-Name Matching [6]).

Traditionally, the problem of tuple de-duplication has been addressed mainly from an accuracy viewpoint, by focusing on the minimization of incorrect matchings. However, efficiency and scalability issues do play a predominant role in many application contexts where large data volumes are involved, especially when the object-identification task is part of an interactive application, calling for short response times. In general, the large volume of involved data imposes severe restrictions on the design of data structures and algorithms for data de-duplication, and disqualifies any approach requiring quadratic time in the database size or producing many random disk accesses and continuous paging activities. Thus, in this paper our main objective is to devise a scalable method for duplicate detection that can be profitably applied to large databases, in an incremental way.

More specifically, many de-duplication approaches in the literature essentially attempt to match or cluster duplicated records (e.g., [6, 14]), and do not guarantee an adequate level of scalability. On the other hand, the usage of traditional clustering algorithms is made unviable by the high number of object clusters expected in a typical deduplication scenario, which can be of the same order as the size of the database. Under a scalability perspective, to the best of our knowledge, the only suitable approach is the one proposed in [13], where objects are first grouped in "canopies", i.e., subsets containing objects suspected to be similar, and pairwise similarities are computed only inside each canopy. However this approach does not cope with incrementality issues. Recently, some approaches have been proposed [9, 1, 5] which exploit efficient indexing schemes based on the extraction of relevant features from the tuples under consideration. Such approaches can be adapted to deal with the de-duplication problem, even though they are not specifically designed to approach the problem from an incremental clustering perspective.

The solution we propose essentially relies on an efficient and incremental clustering technique that allows to discover all clusters containing duplicate tuples. The core of the approach is the usage of a suitable indexing technique which, for any newly arrived tuple μ, allows to efficiently retrieve a set of tuples in the database which are likely mostly similar μ, and hence are expected to refer to the same real-world entity associated with μ. The proposed indexing technique is based on a hashing scheme, which tends to assign objects with high similarity to the same buckets. We exploit a refined key-generation technique which, for each tuple under consideration, guarantees a controlled level of approximation in the search for the nearest neighbors of such a tuple. To this purpose, we resort to a family of *Locality-Sensitive* hashing functions [11, 3, 8], which are guaranteed to assign any two objects to the same buckets with a probability which is directly related to their mutual similarity. Notably, with the help of an empirical evaluation performed on synthesized and real data, we can asses that the hashing-based method ensures effective on-line matching, at the expense of limited accuracy loss.

Notice that the approach proposed in this paper is a substantial improvement of the proposal in [4]. There, we adopted an indexing scheme tailored to a set-based distance function, and the management of each tuple was faced at a coarser granularity. This paper extends and improves that proposal in both effectiveness and efficiency, by allowing for a direct control on the degree of granularity needed to discover the actual neighbors (duplicates) of a tuple.

2 Problem statement and overview of the approach

In the following we introduce some basic notation and preliminary definitions. An item domain $\mathcal{M} = \{a_1, a_2, \ldots, a_m\}$ is a collection of items. We assume m to be very large: typically, $\mathcal{M}$ represents the set of all possible strings available from a given alphabet. Moreover, we assume that $\mathcal{M}$ is equipped with 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 tuple μ is a subset of $\mathcal{M}$. An example tuple is

{`Alfred, Whilem, Salisbury, Hill, 3001, London`}

which represent registry information about a subject. Notice that a more appropriate representation [4] can take into account a relational schema in which each tuple fits. For example, in the above schema, a more informative setting requires to separate the tuples into the fields `NAME`, `ADDRESS`, `CITY`, and to associate an itemset with each field: μ[`NAME`] = {`Alfred,Whilem`}, μ[`ADDRESS`] = {`Salisbury, Hill, 3001`}, μ[`CITY`] = {`London`}. For ease of presentation, we shall omit such details: the results which follow can be easily generalized to such a context.

We assume that the set of all tuples is equipped with a distance function, $dist(\mu, \nu) \in [0, 1]$, which can be defined for comparing any two tuples μ and ν, by suitably combining the distance values computed through $dist_{\mathcal{M}}$ on the values of matching fields. In the following, we assume that both $dist(\mu, \nu)$ and $dist_{\mathcal{M}}$ are defined in terms of the *Jaccard* coefficient.

The core of the *Entity Resolution* problem [4] can be roughly stated as the problem of detecting, within a database $\mathcal{DB} = \{\mu_1, \ldots, \mu_N\}$ of tuples, a suitable partitioning $\mathcal{C}_1, \ldots, \mathcal{C}_K$ of the tuples, such that for each group $\mathcal{C}_i$, intra-group similarity is high and extra-group similarity is low. For example, the dataset

μ_1	`Jeff, Lynch, Maverick, Road, 181, Woodstock`
μ_2	`Anne, Talung, 307, East, 53rd, Street, NYC`
μ_3	`Jeff, Alf., Lynch, Maverick, Rd, Woodstock, NY`
μ_4	`Anne, Talug, 53rd, Street, NYC`
μ_5	`Mary, Anne, Talung, 307, East, 53rd, Street, NYC`

can be partitioned into $\mathcal{C}_1 = \{\mu_1, \mu_3\}$ and $\mathcal{C}_2 = \{\mu_2, \mu_4, \mu_5\}$.

This is essentially a clustering problem, but it is formulated in a specific situation, where there are several pairs of tuples in $\mathcal{DB}$ that are quite dissimilar from each other. This can be formalized by assuming that the size of the set $\{\langle \mu_i, \mu_j \rangle \mid dist(\mu_i, \mu_j) \simeq 1\}$ is $O(N^2)$: thus, we can expect the number K of clusters to be very high – typically, $O(N)$.

A key intuition is that, in such a situation, cluster membership can be detected by means of a minimal number of comparisons, by considering only some relevant neighbors for each new tuple, efficiently extracted from the current database through a proper retrieval method. Moreover, we intend to cope with the clustering problem in an incremental setting, where a new database $\mathcal{DB}_\triangle$ must be integrated with a previously reconciled one $\mathcal{DB}$. Practically speaking, the cost of clustering tuples in $\mathcal{DB}_\triangle$ must be (almost) independent of the size N of $\mathcal{DB}$. To this purpose, each tuple in $\mathcal{DB}_\triangle$ is associated with a cluster in $\mathcal{P}$, which is detected through a sort of *nearest-neighbor* classification scheme.

The algorithm in Figure 1 summarizes our solution to the data reconciliation problem. Notably, the clustering method is parametric w.r.t. the distance function used to compare any two tuples, and is defined in an incremental way, for it allows to integrate a new set of tuples into a previously computed partition. In fact, the algorithm receives a database $\mathcal{DB}$ and an associated partition $\mathcal{P}$, besides the set of new tuples $\mathcal{DB}_\triangle$; as a result, it will produce a new partition $\mathcal{P}'$ of $\mathcal{DB} \cup \mathcal{DB}_\triangle$, obtained by adapting $\mathcal{P}$ with the tuples from $\mathcal{DB}_\triangle$.

In more detail, for each tuple μ_i in $\mathcal{DB}$ to be clustered, the k most prominent neighbors of μ_i are retrieved through procedure KNEARESTNEIGHBOR. The cluster membership for μ_i is then determined by calling the MOSTLIKELYCLASS procedure, estimating the *most likely* cluster among those associated with the neighbors of μ_i. If such a cluster does not exist, μ_i is estimated not to belong to any of the

```
GENERATE-CLUSTERS($\mathcal{P}$,$\mathcal{DB}_\Delta$,$k$)
  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 $\mathcal{P}' = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ and $\mathcal{DB}_\Delta = \{\mu_1, \dots, \mu_n\}$;
  3: for $i = 1 \dots n$ do
  4:     $neighbors \leftarrow$ kNEARESTNEIGHBOR($\mathcal{DB}', \mu_i, k$);
  5:     $\mathcal{C}_j \leftarrow$ MOSTLIKELYCLASS($neighbors, \mathcal{P}'$);
  6:     $\mathcal{DB}' \leftarrow \mathcal{DB}' \cup \{\mu_i\}$;
  7:     if $\mathcal{C}_j = \emptyset$ then
  8:         create a new cluster $\mathcal{C}_{m+1} = \{\mu_i\}$;
  9:         $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathcal{C}_{m+1}\}$;
  10:    else
  11:        $\mathcal{C}_j \leftarrow \mathcal{C}_j \cup \{\mu_i\}$;
  12:        PROPAGATE($neighbors, \mathcal{P}'$);
  13:    end if
  14: end for

PROPAGATE($S$,$\mathcal{P}$)
  P1: for all $\mu \in S$ do
  P2:     $neighbors \leftarrow$ kNEARESTNEIGHBOR($\mathcal{DB}, \mu, k$);
  P3:     $\mathcal{C} \leftarrow$ MOSTLIKELYCLASS($neighbors, \mathcal{P}$);
  P4:     if $\mu \notin \mathcal{C}$ then
  P5:         $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mu\}$;
  P6:         PROPAGATE($neighbors, \mathcal{P}$);
  P7:     end if
  P8: end for
```

Figure 1. Clustering algorithm

existing clusters with a sufficient degree of certainty, and hence it is assigned to a newly generated cluster.

Finally, procedure PROPAGATE is meant to scan the neighbors of μ_i in order to possibly revise their cluster memberships, since in principle they could be affected by the insertion of μ_i. Notice that, in typical Entity Resolution settings, where clusters are quite far from each other, the propagation affects only a reduced number of tuples, and ends in a few iterations. Further details on the clustering scheme sketched above can be found in [4].

It is worth remarking that the complexity of the algorithm in Figure 1, given the size N of $\mathcal{DB}$ and M of $\mathcal{DB}_\Delta$, depends on the three major tasks: *(i)* the search for neighbors (line 4, having cost n), *(ii)* the voting procedure (line 5, with a cost proportional to k), and *(iii)* the propagation of cluster labels (line 12, having a cost proportional to n, based on the discussion above). As they are performed for each tuple in $\mathcal{DB}_\Delta$, the overall complexity is $O(M(n+k))$. Since k is $O(1)$, it follows that the main contribution to the complexity of the clustering procedure is due to the cost $O(n)$ of the kNEARESTNEIGHBOR procedure. Therefore, the main efforts towards computational savings are to be addressed in designing an efficient method for neighbor search. 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.

In [4], we proposed an hashing scheme which maps any tuple into a proper set of features, so that the similarity between two tuples can be evaluated by simply looking at their respective features. To this purpose, a hash-based index structure, simply called *Hash Index*, was introduced, which consists of a pair $H = \langle FI, ES \rangle$, where:

– ES, referred to as *External Store*, is a storage structure devoted to manage a set of tuple buckets by an optimized usage of disk pages: each bucket gathers tuples that are estimated to be similar to each other, in that they share a relevant set of properly defined features;

– FI, referred to as *Feature Index*, is an indexing structure which, for each given feature s, allows to efficiently recognize all the buckets in ES that contain tuples exhibiting s.

Figure 2 shows how such an index can be used to perform nearest-neighbor searches, so supporting the whole clustering approach previously described. The algorithm works according to the number k of desired neighbors.

```
kNEARESTNEIGHBORS($\mathcal{DB}$,$\mu$,$k$)
  1: Let $S = \{s \mid s$ is a relevant feature of $\mu\}$;
  2: $\mathcal{N} \leftarrow \emptyset$;
  3: while $S \neq \emptyset$ do
  4:     $x = S.Extract()$;
  5:     $h \leftarrow FI.Search(x)$;
  6:     if ($h = 0$) then
  7:         $h \leftarrow FI.Insert(x)$;
  8:     else
  9:         while $\nu = ES.Read(h)$ do
  10:            if $\mathcal{N}.size < k$ or $dist(\mu,\nu) < \mathcal{N}.MaxDist()$ then
  11:                $\mathcal{N}.Insert(\nu, dist(\mu,\nu))$;
  12:            end if
  13:        end while
  14:    end if
  15:    $ES.Insert(\mu, h)$;
  16: end while
  17: return $\mathcal{N}$;
```

Figure 2. The kNEARESTNEIGHBOR procedure.

A major point in the proposed approach is the choice of features for indexing tuples, which will strongly impact on the effectiveness of the whole method, and should be carefully tailored to the criterion adopted for comparing tuples. In [4] we described an indexing scheme which is meant to retrieve similar tuples, according to a set-based dissimilarity function, namely the *Jaccard distance* – for any two tuples $\mu, \nu \subseteq \mathcal{M}$, $dist(\mu,\nu) = 1 - |\mu \cap \nu|/|\mu \cup \nu|$. In practice, we assume that $dist_\mathcal{M}$ corresponds to the Dirichlet function, and that, consequently, the dissimilarity between two itemsets is measured by evaluating their degree of overlap.

In this case, a possible way of indexing a tuple μ simply consists in extracting a number of non-empty subsets of μ, named *subkeys* of μ, as indexing features. As the number of all subkeys is exponential in the cardinality of the given tuple, the method is tuned to produce a minimal set of "significant" subkeys. In particular, a subkey s of μ is said δ-significant if $\lfloor |\mu| \times (1-\delta) \rfloor \leq |s| \leq |\mu|$.

Notably, any tuple ν such that $dist_J(\mu,\nu) \leq \delta$ must contain at least one of the δ-significant subkeys of μ [4]. Therefore, searching for tuples that exhibit at least one of the δ-significant subkeys derived from a tuple μ constitutes a strategy for retrieving all the neighbors of μ without scanning the whole database. This strategy also guarantees an adequate level of selectivity: indeed, if μ and ν contain a sensible number of different items, then their δ-significant subkeys do not overlap. As a consequence, the probability that μ is retrieved for comparison with ν is low.

Despite its simplicity, this indexing scheme was proven to work quite well in practical cases [4]. Notwithstanding, two main drawbacks can be observed:

1. The cost of the approach critically depends on the number of δ-relevant subkeys: the larger is the set of subkeys, the higher is the number of writes needed to update the index.

2. More importantly, the proposed key-generation technique suffers from a coarse grain dissimilarity between itemsets, which does not take into account a more refined definition of $dist_{\mathcal{M}}$. Indeed, the proposed approach is subject to fail, in principle, in cases where the likeliness among single tokens has to be detected as well. As an example, the tuples

μ_1	`Jeff, Lynch, Maverick, Road, 181, Woodstock`
μ_2	`Jef, Lync, Maverik, Rd, 181, Woodstock`

   are not reckoned as similar in the proposed approach (even though they clearly refer to the same entity), due to slight differences between some semantically equivalent tokens. Notice that lowering the degree δ of dissimilarity partially alleviates this effect, but at the cost of worsening the performances of the index notably.

3 Hierarchical Approximate hashing based on q-grams

Our objective in this section is to define a hash-based index which is capable of overcoming the above described drawbacks. In particular, we aim at defining a key-generation scheme which only requires a constant number of disk writes and reads, yet guaranteeing a fixed (low) rate of false negatives.

To overcome these limitations, we have to generate a fixed number of subkeys, which however are capable of reflecting both the differences among itemsets, and those among tokens. To this purpose, we define a key-generation scheme by combining two different techniques:

– the adoption of hash functions based on the notion of *minwise independent permutation* [8, 2], for bounding the probability of collisions;

– the use of q-grams (i.e., contiguous substrings of size q) for a proper approximation of the similarity among string tokens [9].

A *locally sensitive* hash function H for a set S, equipped with a distance function D, is a function which bounds the probability of collisions to the distance between elements. Formally, for each pair $p, q \in S$ and value ϵ, there exists values P_1^ϵ and P_2^ϵ such that: *(i)* if $D(p,q) \leq \epsilon$ then $\Pr[H(p) = H(q)] \geq P_1^\epsilon$, and *(ii)* if $D(p,q) > \epsilon$ then $\Pr(H(p) = H(q)] > P_2^\epsilon$.

Clearly, such a function H provides a simple solution to the problem of false negatives described in the previous section. Indeed, for each μ, we can define a representation $rep(\mu) = \{H(a) | a \in \mu\}$, and fill the hash-based index by exploiting δ-significant subkeys from such a representation. To this aim, we can exploit the theory of minwise independent permutations [2]. A minwise independent permutation is a coding function π of a set X of generic items such that, for each $x \in X$, the probability of the code associated with x being the minimum is uniformly distributed, i.e., $\Pr[\min(\pi(X)) = \pi(x)] = \frac{1}{|X|}$.

A minwise independent permutation π naturally defines a locally sensitive hash function H over an itemset X, defined as $H(X) = \min(\pi(x))$. Indeed, for each two itemsets X and Y, it can be easily verified that $\Pr[\min(\pi(X)) = \min(\pi(Y))] = \frac{|X \cap Y|}{|X \cup Y|}$. This suggests that, by approximating $dist_{\mathcal{M}}(a_i, a_j)$ with the Jaccard similarity among some given features of a_i and a_j, we can adopt the above envisaged solution to the problem of false negatives. When $\mathcal{M}$ contains string tokens (as it usually happens in a typical entity resolution setting), the features of interest of a given token a can be represented by the q-grams of a. It has been shown [9, 16] that the comparison of the q-grams provides a suitable approximation of the Edit distance, which is typically adopted as a classical tool for comparing strings.

In the following, we show how minwise functions can be effectively exploited to generate suitable keys. Consider the example tuples

μ_1	`Jeff, Lynch, Maverick, Road, 181, Woodstock`
μ_2	`Jef, Lync, Maverik, Rd, Woodstock`

Clearly, the tuples μ_1 and μ_2 refer to the same entity (and hence should be associated with the same key). The detection of such a similarity can be accomplished by resorting to the following observations:

1. some tokens in μ_1 are strongly similar to tokens in μ_2. In particular, `Jeff` and `Jef`, `Lynch` and `Lync`, `Road` and `Rd`, and `Maverick` and `Maverik`. Thus, denoting by a_1, a_2, a_3, and a_4 the four "approximately common" terms, the tuples can be represented as:

μ_1	a_1, a_2, a_3, a_4, `181,Woodstock`
μ_2	a_1, a_2, a_3, a_4, `Woodstock`

2. the postprocessed tuples exhibit only a single mismatch. If a minwise permutation is applied to both, the resulting key shall very likely be the same.

Thus, a minwise function can be applied over the "purged" representation of a tuple μ, in order to obtain an effective key. The purged version of μ should guarantee that tokens exhibiting high similarity with tokens in other tuples, change their representation towards a "common approximate" token.

Again, the approximate representation of a token, described in point 1 of the example above, can be obtained by resorting to minwise functions. Given two tokens a_i and a_j, recall that their dissimilarity $d_{\mathcal{M}}(a_i, a_j)$ is defined in terms of the Jaccard coefficient. In practice, if $feat(a)$ represents a set of features of token a, then $d_{\mathcal{M}}(a_i, a_j) = 1 - |feat(a_i) \cap feat(a_j)|/|feat(a_i) \cup feat(a_j)|$. The set $feat(a)$ can be defined in terms of q-grams. The latter represent the simplest yet effective information contents of a token and indeed, they have been widely used and demonstrated fruitful in estimating the similarity of strings [9, 16]. Hence, by applying a minwise function to the set of q-grams of a, we again have the guarantee that similar tokens collapse to a unique representation.

Thus, given a tuple μ to encode, the key-generation scheme we propose works in two different levels:

- In the first level, each element $a \in \mu$ is encoded by exploiting a minwise hash function H^l. This guarantees that two similar but different tokens a and b are with high probability associated with a same code. As a side effect, tuples μ and ν sharing "almost similar" tokens are purged into two representations where such tokens converge towards unique representations.

- In the second level, the set $rep(\mu)$ obtained from the first level is encoded by exploiting a further minwise hash function H^u. Again, this guarantees that tuples sharing several codes are likely associated with the same key.

The final code obtained in this way can be effectively adopted in the indexing structure described in section 2.

A key point is the definition of a proper family of minwise independent permutations upon which to define the hash functions. A very simple idea is to randomly map a feature x of a generic set X to a natural number. Then, provided that the mapping is random, the probability of mapping a generic $x \in X$ to a minimum number is uniformly distributed, as required. In practice, it is hard to obtain a truly random mapping. Hence, we exploit a family of "practically" minwise independent permutations [2], i.e., the functions $\pi(x) = (a \cdot c(x) + b) \bmod p$, where $a \neq 0$ and $c(x)$ is a unique numeric code associated with x (such as, e.g. the code obtained concatenating the ASCII characters it includes). Provided that a, b, $c(x)$ and p are large enough, the behavior of π is practically random, as we expect.

We further act on the randomness of the encoding, by combining several alternative functions (obtained by choosing different values of a, b and p). Recall that a hash function on π is defined as $H_\pi(X) = \min(\pi(X))$, and that $\Pr[H_\pi(X) = H_\pi(Y)] = |X \cap Y|/|X \cup Y| = \epsilon$. Notice that the choice of a, b and p in π introduces a probabilistic bias in H_π, which can in principle leverage false negatives. Let us consider the events $A \equiv$"sets X and Y are associated with the same code", and $B = \neg A$. Then, $p_A = \epsilon$ and $p_B = 1 - \epsilon$. By exploiting h different encodings $H^l_1, \ldots, H^l_h$ (which differ in the underlying π permutations), the probability that all the encodings exhibit a different code for X and Y is $(1-\epsilon)^h$. If $\epsilon > 1/2$ represents the average similarity of items, we can exploit the h different encodings for computing h alternative representations $rep_1(\mu), \ldots, rep_h(\mu)$ of a tuple μ. Then, by exploiting all these representations in a disjunctive manner, we lower the probability of false negatives to $(1-\epsilon)^h$.

In general, allowing several trials generally flavors high probabilities. Consider the case where $\epsilon < 1/2$. Then, the probability that, in k trials (corresponding to k different choices of a, b and p) at least one trial is B is $1 - \epsilon^k$. We can apply this to the second-level encoding, where, conversely from the previous case, the probabilistic bias can influence false positives. Indeed, two dissimilar tuples μ and ν could in principle be associated with the same token, due to a specific bias in π which affects the computation of minimum random code both in $rep_i(\mu)$ and in $rep_i(\nu)$. If, by the converse a key is computed as a concatenation of k different encodings $H^u_1, \ldots, H^u_k$, the probability of having a different key for μ and ν is $1 - \epsilon^k$, where ϵ is the Jaccard similarity between $rep_i(\mu)$ and $rep_i(\nu)$.

It is worth noting that the application of LSH methods to nearest neighbors searches in high-dimensional spaces has been thoroughly studied: [8, 7], in particular, adopt a hierarchical combination of LSH functions similar to ours. However, their objective is rather different: given an object q from some given domain D and a threshold δ, retrieve the set $S = \{p \in D | d(q,p) < \delta\}$ in $O(|S|)$. By contrast, in a deduplication scenario the value δ is unknown, and is in general parametric to the similarity function to be adopted. Thus, using LSH in this context raises some novel, specific, issues, which have not yet been studied. In particular, since the level of mismatch (e.g., mispelling errors) actually affecting duplicate tuples is an application-dependent parameter, it is not clear how to tune the h and k parameters. On the other hand, the extraction of q-grams from input strings introduces a further degree of freedom in the method, since determining a proper value for q-gram sizes is not a trivial task, in general. Notice that the empirical analysis illustrated in the following section is crucial in this respect, as it allows to get insight on the issue of optimally setting the above parameters.

4 Experimental Results

The above discussion shows that the effectiveness of the approach relies on choosing proper values of h and k. Low values of h leverage false negatives, whereas high values leverage false positives. Analogously, low values of k leverage false positives, whereas high values should, in principle, leverage false negatives.

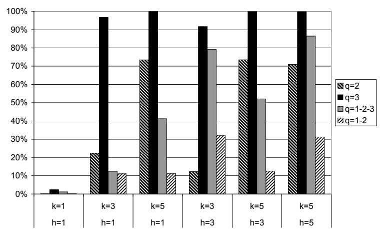
Thus, this section is devoted to studying suitable values of these parameters that fix a high correspondence between the retrieved and the expected neighbors of a tuple. To this purpose, for a generic tuple μ we are interested in evaluating the number TP_μ of *true positives* (i.e., the tuples which are retrieved and that belong to the same cluster of μ), and compare it to the number of *false positives* FP_μ (i.e., tuples retrieved without being neighbors of μ), and *false negatives* FN_μ (i.e., neighbors of μ which are not retrieved). As global indicators we exploit the average precision and recall per tuple, i.e. $\mathit{precision} = \frac{1}{N}\sum_{\mu \in \mathcal{DB}} \frac{TP_\mu}{TP_\mu + FP_\mu}$ and $\mathit{recall} = \frac{1}{N}\sum_{\mu \in \mathcal{DB}} \frac{TP_\mu}{TP_\mu + FN_\mu}$, where N denotes the number of tuples in $\mathcal{DB}$.

The values of such quality indicators influence the effectiveness of the clustering scheme in Figure 1. In general, high values of precision allows for correct de-duplication: indeed, the retrieval of true positives directly influences the MOSTLIKELYCLASS procedure which assigns each tuple to a cluster. When precision is low, the clustering method can be effective only if recall is high.

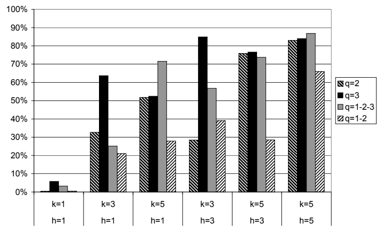
Notice that low precision may cause a degradation of performances, if the number of false positives is not bounded. Thus, we also evaluate the efficiency of the indexing scheme, in terms of the number of tuples retrieved by each search. This value depends on h and k, and is clearly related to the rate of false positives.

Experiments were conducted on both real and synthesized data. For the real data, we exploited a (not publicly available) collection of about 105,140 tuples, representing information about customers of an Italian bank. Synthetic data was produced by generating 50,000 clusters with an average of 20 tuples per cluster, and each tuple containing 20 tokens in the average. Each cluster was obtained by first generating a representative of the cluster, and then producing the desired duplicates as perturbations of the representative. The perturbation was accomplished either by deleting, adding or modifying a token from the cluster representative. The number of perturbations was governed by a gaussian distribution having p as mean value. The parameter p was exploited to study the sensitivity of the proposed approach to the level of noise affecting the data, and due, for example to misspelling errors.

Figures 3 and 4 illustrate results of some tests we conducted on this synthesized data, in order to analyze the sensitivity to the parameters q, h and k (relative to the indexing



(a) precision vs. h, k and q



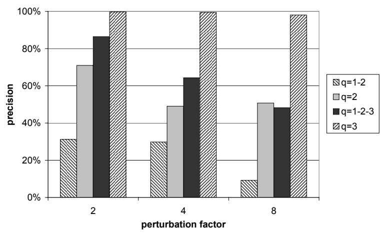
(b) recall vs. h, k and q

Figure 3. Results on synthetic data w.r.t. q-gram size (q) and nr. of hash functions (h,k)

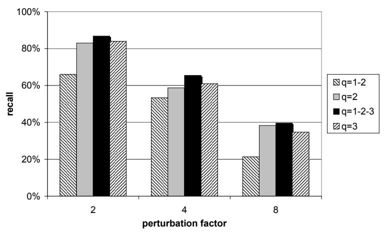
scheme), and p (relative to the noise in the data). In particular, the values of q ranged over 2, 3, 1-2 (both 1-grams and 2-grams) and 1-2-3 (q-grams with sizes 1, 2 and 3).

Figures 3.(a) and 3.(b) show the results of precision and recall for different values of h and k, and $p = 2$. We can notice that precision raises on increasing values of k, and decreases on increasing values of h. The latter statement does not hold when q-grams of size 1 are considered. In general, stabler results are guaranteed by using q-grams of size 3. As to the recall, we can notice that, when k is fixed, increasing values of h correspond to improvements as well. If h is fixed and k is increased, the recall decreases only when $q = 3$. Here, the best results are guaranteed by fixing q=1-2-3. In general, when $h \geq 3$ and $k \geq 3$, the indexing scheme exhibits good performances.

Figures 4.(a) and 4.(b) are useful to check the robustness of the index. As expected, the effectiveness of the approach tends to degrade when higher values of the perturbation factor p are used to increase intra-cluster dissimilarity. However, the proposed retrieval strategy keeps on exhibiting values of precision and recall that can still enable an effective clustering. In more detail, the impact of perturbation on precision is clearly emphasized when tuples are encoded by using also 1-grams, whereas using only either 2-grams

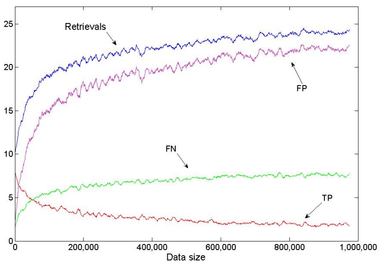


(a) precision vs. perturbation and q

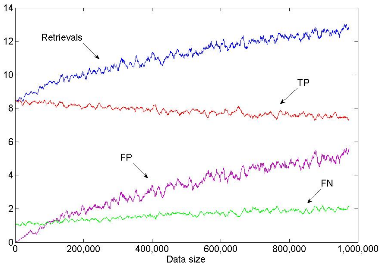


(b) recall vs. perturbation and q

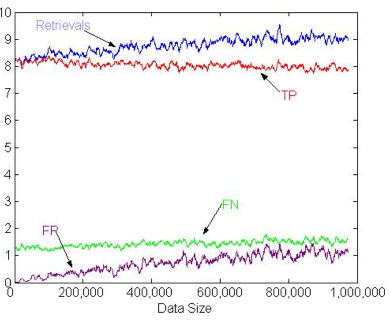
Figure 4. Results on synthetic data w.r.t. q-gram size (q) and perturbation



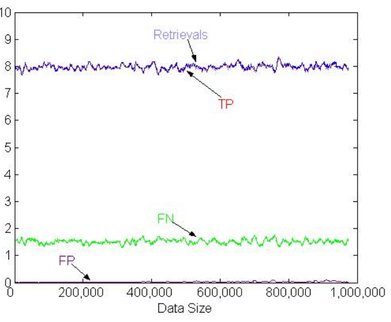
(a) $q = 2, h = 3, k = 3$



(b) $q = 2, h = 5, k = 5$



(c) $q = 3, h = 3, k = 3$



(d) $q = 3, h = 5, k = 5$

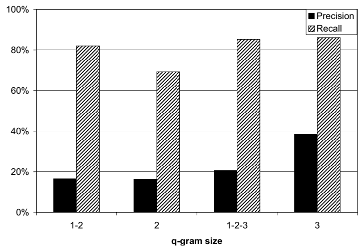
Figure 5. Scalability w.r.t. the data size

or 3-grams allows for making precision results stabler. Notice that for $q = 3$ a nearly maximum value of precision is achieved, even when a quite perturbed data set is used.

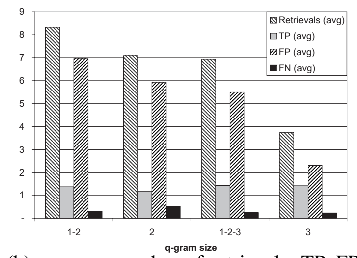
Figure 5 provides some details on the progress of the number of retrieved neighbors, TP, FP and FN, when an increasing number of tuples, up to 1,000,000, is inserted in the index. For space reasons, only some selected combinations of h and k, and q are considered, which were deemed as quite effective in previous analysis. Anyway, we pinpoint that some general results of the analysis illustrated here also apply to other cases. The values are averaged on a window of 5,000 tuples. In general, it is interesting to observe that the number of retrievals for each tuple is always bounded, although for increasing values of the data size the index grows. This general behavior, which we verified for all configurations of h, k and q, clearly demonstrates the scalability of the approach. In particular, we observe that for $q = 3$ the number of retrievals is always very low and nearly independent of the number of tuples inserted (see figures 5.(c) and 5.(d)). More in general, the figures confirm the conceptual analysis that the number of I/O operations directly depends on the parameter h, the latter determining the number of searches and updates against the index.

All these figures also agree with the main outcomes of the effectiveness analysis previously conducted with the help of Figure 3. In particular, notice the quick decrease of FP and FN when both k and h turn from 3 to 5, in the case of $q = 2$ (Figure 5.(a) and 5.(b)), that motivates the improvement in both precision and recall observed in these cases. Moreover, the high precision guaranteed by our approach with q-grams of size 3, is substantiated by Figures 5.(c) and 5.(d), where the number of retrieved tuples is very close to TP; in particular, for $k = 5$ (Figure 5.(d)) the FP curve definitely flatten on the horizontal axis.

The above considerations are confirmed by experiments on real data. Figure 6.(a) shows the results obtained for precision and recall by using different values of q, whereas figure 6.(b) summarizes the average number of retrievals and quality indices. As we can see, recall is quite high even if precision is low (thus allowing for a still effective clustering). Notice that the average number of retrievals is low, thus guaranteeing a good scalability of the approach.



(a) precision and recall



(b) average number of retrievals, TP, FP and FN

Figure 6. Results on real data using different q-gram sizes

5 Conclusions and Future Works

In this paper, we addressed the problem of recognizing duplicate data, 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. To this purpose, we studied a refined key-generation technique, which allows a controlled level of approximation in the search for the nearest neighbors of a tuple. An empirical analysis, on both synthesized and real data, showed the validity of the approach.

Clearly, when strings are too small or too different to contain enough informative content, the de-duplication task cannot be properly accomplished by the proposed clustering algorithm. To this purpose, we plan to study the extension of the proposed approach to different scenarios, where more informative similarity functions can be exploited. An example is the adoption of link-based similarity: recently, some techniques were proposed [12] which have been proved effective but still suffer from the incrementality issues which are the focus of this paper.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. VLDB Conf.*, pages 586–597, 2002.

[2] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Minwise independent permutations. In *Proc. STOC Conf.*, pages 327–336, 1998.

[3] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering on the web. In *Proc. WWW Conf.*, pages 1157–1166, 1997.

[4] E. Cesario, F. Folino, G. Manco, and L. Pontieri. An incremental clustering scheme for duplicate detection in large databases. In *Proc. IDEAS Conf.*, 2005.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. SIGMOD Conf.*, pages 313–324, 2003.

[6] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proc. SIGKDD Conf.*, pages 475–480, 2002.

[7] A. Gionis, D. Gunopulos, and N. Koudas. Efficient and tunable similar set retrieval. In *Proc. SIGMOD Conf.*, 2001.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. VLDB Conf.*, pages 518–529, 1999.

[9] L. Gravano et al. Approximate string joins in a database (almost) for free. In *Proc. VLDB Conf.*, pages 518–529, 2001.

[10] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. SIGMOD Conf.*, pages 127–138, 1995.

[11] P. Indyk and R. Motwani. Approximate nearest neighbor - towards removing the curse of dimensionality. In *Proc. STOC Conf.*, pages 604–613, 1998.

[12] D. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting relationships for domain independent data cleaning. In *Proc. SIAM Conf.*, pages 262–273, 2005.

[13] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. SIGKDD Conf.*, pages 169–178, 2000.

[14] A. E. Monge and C. P. Elkan. The field matching problem: Algorithms and applications. In *Proc. SIGKDD Conf.*, pages 267–270, 1996.

[15] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. SIGKDD Conf.*, pages 269–278, 2002.

[16] E. Ukkonen. Approximate string matching using q-grams and maximal matches. *TCS*, 92(1):191–211, 1992.