An Incremental Clustering Scheme for Data Deduplication

Gianni Costa, Giuseppe Manco, Riccardo Ortale
ICAR-CNR
Via P. Bucci 41c
I87036 Rende (CS), Italy
email {costa, manco, ortale}@icar.cnr.it

July 21, 2009

Abstract

We propose an incremental technique for discovering duplicates in large databases of textual sequences, i.e. syntactically different tuples, that refer to the same real-world entity. The problem is approached from a *clustering* perspective: given a set of tuples, the objective is to partition them into groups of duplicate tuples. Each newly arrived tuple is assigned to an appropriate cluster via *nearest-neighbor* classification. This is achieved by means of a suitable hash-based index, that maps any tuple to a set of indexing keys and assigns tuples with high syntactic similarity to the same buckets. Hence, the neighbors of a query tuple can be efficiently identified by simply retrieving those tuples that appear in the same buckets associated to the query tuple itself, without completely scanning the original database. Two alternative schemes for computing indexing keys are discussed and compared. An extensive experimental evaluation on both synthetic and real data shows the effectiveness of our approach.
Keywords: Clustering - Mining methods & algorithms; Record classification; Indexing methods & structures; Locality sensitive hashing; Min-wise independent permutations; Approximated similarity measures; De-duplication

1 Introduction

Recognizing similarities in large collections of data is a major issue in the context of designing information integration systems. The wide exploitation of new techniques and systems for generating, collecting and storing data has made available a huge amount of information. Large quantities of such data are stored as continuous text, such as personal demographic data, bibliographic information, phone and mailing lists. Often, the integration of such data is a problematic process, that involves dealing with two major issues, namely *structural* and *syntactic* heterogeneity. The former occurs when the data does not explicitly exhibit a common field structure. In this case, *schema reconciliation* techniques [16, 30, 31] allow the segmentation of textual sequences into a uniform schema.

However, whether or not reconciled into a common database schema, data can still be affected by *syntactic heterogeneity*. This is a fundamental issue in the context of information integration systems, that consists in discovering duplicates within the integrated data, i.e. syntactically different records that, as a matter of facts, refer to a same real-world entity.

De-duplication is necessary in several applicative settings. A typical example is the reconciliation of demographic data sources into a data warehousing setting. Names and addresses can be stored in rather different formats, thus raising the need for an effective reconciliation strategy, that could be crucial for effective decision making. In these cases, a (typically large) volume of small strings is analyzed to reconstruct the semantic information on the basis of few syntactic information. Consider, e.g., a banking scenario, where the main interest is to rank the credit risk of a customer, by looking at the past insolvency history. Since information about payments may come from different sources, each of which conforming

to a possibly different encoding-format for the data, de-duplicating demographic tuples is crucial in order to correctly analyze the attitude at insolvency of the customer.

In such applicative scenarios, tuples usually correspond to small sequences of strings, characterized by an inherent segmentation in specific semantical entities. However, such a segmentation is not known in advance and this further exacerbates the difficulties behind the identification of duplicates.

In the current literature, the notion of *Entity Resolution* [35–37] has been used to denote a complex process for data manipulation, that embraces *schema reconciliation*, *data reconciliation* and *identity definition* (i.e. the act of grouping duplicate tuples, in order to extract a representative tuple for each such a group). This paper aims at devising an effective and incremental approach to the reconciliation of textual entities, that is capable to efficiently de-duplicate volumes of data.

In particular, the requirement to process large bodies of data generally imposes severe restrictions on the design of data structures and algorithms for de-duplication. Such restrictions are necessary to ultimately limit both the computational complexity of the de-duplication scheme (a required time, that is quadratic in the size of the underlying database, is prohibitively high) and its I/O operations on disk (several random accesses to secondary storage imply continuous paging activities). Thus, we focus on scaling techniques for duplicate detection to large databases. The problem is approached from a *clustering* perspective: given a set of tuples, the objective is to recognize subsets (clusters) of tuples such that intra-cluster similarity is high and inter-cluster similarity is low. Three major peculiarities make the de-duplication problem, that we intend to cope with, significantly different from the traditional one:

- tuples are represented as (small) sequences of tokens, where the set of possible tokens is large;
- the number of clusters is too high to allow the adoption of traditional clustering techniques;
- the streaming (constantly increasing) nature of the data calls for linear-time clustering algorithms.

Typically, tuple de-duplication has been addressed from an accuracy viewpoint, by attempting to minimize incorrect matchings: false-positives (i.e., records recognized as similar, that actually do not correspond to the same entity) and false-negatives (i.e., records corresponding to the same entity, that are not recognized as similar). In principle, the pursuit of such objectives would call for the development of accurate clustering methods, that exploit suitable distance metrics for catching syntactic proximity between tuples with multiple fields. Viewed in this respect, one possible approach would consist in the adoption of a hierarchical clustering method (this family of algorithms is widely known in the literature for producing top quality results [46]), equipped with a suitable record matching scheme, that leverages accurate field-wise similarity metrics, such as Edit Distance, Affine Gap Distance, Smith-Waterman Distance and Jaro Distance (see [47] for a survey) to match corresponding tuple tokens. However, the quadratic complexity of hierarchical clustering in the number of available tuples, combined with the high cost of field matching schemes (that becomes quadratic w.r.t. the length of tokens in the case of Edit Distance), would make the resulting approach impractical in the great majority of applicative domains, where even a moderate volume of data is available.

Efficiency and scalability issues do play a predominant role in many applicative contexts, where large data volumes are involved, especially when the object-identification task is part of an interactive application, calling for short response times. For instance, the typical volume of data collected on a daily basis in a banking context amounts on average to $500,000$ instances, representing credit transactions performed by customers throughout the various agencies. In such a case, the naive solution of comparing such instances in a pairwise manner, according to some given similarity measure, is infeasible. As an example, for a set of $30,000,000$ tuples (i.e., data collected in a 2 months-monitoring), the hypothetical hierarchical strategy, would require $O(10^{15})$ (a quadrillion) comparisons, which is clearly prohibitive.

In order to meet the efficiency and scalability requirements, we study efficient schemes for effectively approaching de-duplication in terms of clustering. The starting point is to sensibly lower the overall number of tuple comparisons along with the cost for evaluating similarity between individual tokens. A viable approach in this respect consists in partitioning the original data into subsets, such that each subset includes very similar tuples. The search for duplicates of a given tuple is thus narrowed to the records

within the subsets ascribable to the tuple. Canopies [42] may be taken into account to this purpose. Unfortunately, their construction does not satisfy the incrementality requirement. Rather, in this paper, we essentially rely on efficient techniques, that allow to discover all clusters of duplicate tuples in an incremental fashion. The intuition is to progressively construct some sort of canopies with a single scan of the data, by exploiting a suitable index/storage structure, that efficiently supports similarity queries. The de-duplication of any newly arrived tuple t is thus achieved by retrieving a set of neighboring tuples in the database, which are mostly similar to t and, hence, are expected to refer to the same real-world entity associated with t. The cluster membership of t is then established via a majority vote from the neighbors.

Conceptually, the M-Tree index structure [10] seems useful for indexing and organizing tuples, according to a certain similarity metric, so that nearest-neighbor queries would be answered with minimal processing time and I/O cost. However, the empirical analysis of the behavior of the M-Tree structure reveals that the existence of several tuples, with heterogeneous syntactic representations, results into multiple inner levels of the index, which makes the cost of proximity search almost linear in the number of the original records. Such a phenomenon makes the M-Tree index inadequate for de-duplication in the delineated applicative context.

A more convenient approach consists in adopting a hash-based index, capable of executing nearest neighbor searches in a time, that is independent of the number of database tuples. Here, a suitable hashing scheme assigns similar tuples to the same corresponding buckets. Hence, the search for duplicates is focused only on the records that fall within the same buckets associated to the query tuple.

The main contribution of this paper is thus a methodological approach to the detection of syntactically similar strings in large databases, that is of great practical relevance in a wide variety of applicative domains, other than the banking context that motivated our research effort, such as the manipulation of search logs, customer data and census applications.

In particular, we first introduce a general framework for formally characterizing the problem of discovering and merging duplicate objects, essentially in terms of a specific clustering problem. For the sake of generality, we assume any database instance to be represented as an itemset, thus going beyond the realm of structured data usually considered in some related approaches.

We then study two different hashing schemes. The first approach, initially proposed in [9], is tailored to a set-based distance function. Here, the management of each tuple is faced at a coarser granularity. Notwithstanding, such a naive approach is compared against the M-Tree index structure [10], and the superiority of the former is shown. Moreover, the comprehension of the limitations of the naive hashing scheme (essentially, the required tradeoff between accuracy and index updates as well as its incapability at properly dealing with nearly identical tokens) provides essential hints for extending and improving the original proposal in both effectiveness and efficiency.

The second approximated hashing scheme allows for a direct control on the degree of granularity needed to properly discover the actual neighbors (i.e. duplicates) of a tuple. More precisely, a refined key-generation technique is developed, which guarantees, for each tuple under consideration, a controlled level of approximation in the search for the nearest neighbors of the tuple itself. To this purpose, we resort to a family of *Locally-Sensitive* hashing functions [6, 7, 15], that are guaranteed to assign any two objects to the same buckets with a probability that is directly related to their similarity degree. In particular, we show how locally-sensitive hash functions can be fruitfully combined in a hierarchical fashion, thus enabling effective on-line matching, at the expense of negligible accuracy loss. As a preview, the key features of the devised methodology can be summarized as follows.

- An incremental clustering algorithm, that scales to de-duplicate volumes of data in terms of both effectiveness and efficiency. Incrementality has not been a major requirement in previous approaches from the literature.
- The development of a hierarchical hashing scheme for catching tuple similarity, that avoids resorting to costly similarity metrics.
- A mechanism for governing the behavior of the hash-based index, that is more easily tunable w.r.t.

to the ones in the current literature (such as, e.g., [6]).

- A thorough experimental evaluation carried out over both real and synthetic data, that reveals the better performance delivered by the naive hashing technique against the one resulting from the adoption of the M-Tree index and, also, an additional improvement in efficiency, without substantial loss in effectiveness, achieved by the second approximated method w.r.t. the first naive one.

It is worth remarking that the approach we propose in this paper is purely syntactic, as it deals with lexical variations occurring across the textual sequences at hand. As a matter of fact, our approach can be considered complementary to other methods, that also attempt to reconstruct and exploit additional information, such as the link-structure among the available sequences [2, 17]. Also, notice that the techniques proposed in this paper are effective in scenarios where large collections of data are available and, hence, can be exploited in a preliminary exploratory phase, where the number of candidate duplicates is significantly reduced.

The paper proceeds as follows. Section 2 provides a formal framework for the overall entity-resolution process and introduces a series of algorithms, that pursue tuple de-duplication via k-nearest neighbor clustering. A hash-based indexing scheme is discussed in Section 3 as a basic nearest-neighbor procedure, that extracts tuple subsets as indexing keys and naively matches such keys to retrieval purposes. Section 4 develops a refined, hierarchical scheme for tuple-key generation, that overcomes the limitations of exact token-matching. Section 5 presents the results of an intensive experimental evaluation. Section 6 overviews works from the literature, that are most closely related to our study. Finally, section 7 concludes the paper by drawing some conclusions and highlighting some major directions, that are worth further research.

2 Problem Statement and Overview of the Approach

We start by providing some basic notation and preliminary definitions. A *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. A textual *sequence* μ is a tuple $a_1, \ldots, a_m$, where each $a_i \in \mathcal{M}$. The set of all possible sequences is denoted by $\mathcal{M}^*$. 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.

Also, we hypothesize 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 basic items.

An input database is a set of tuples $\mathcal{DB} = \{\mu_1, \ldots, \mu_N\}$. Data de-duplication can be formally stated as the problem of partitioning a database $\mathcal{DB}$ into l groups $\mathcal{P} = \{C_1, \ldots, C_l\}$, where each group represents the same entity and each different entity is enumerated in a different group. Notably, data de-duplication is essentially a clustering problem, as it requires finding groups of sequences such that their intra-cluster similarity is high, and inter-cluster similarity is low. However, 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 l of clusters to be very high – typically, $O(N)$.

In addition, we intend to cope with the clustering problem in an incremental setting, where a new input database $\mathcal{DB}_\Delta$ must be integrated within a previously de-duplicated $\mathcal{DB}$. Practically speaking, the cost of clustering tuples in $\mathcal{DB}_\Delta$ must be (almost) independent of the size N of $\mathcal{DB}$. To this purpose, each tuple in $\mathcal{DB}_\Delta$ must be associated with an appropriate cluster in $\mathcal{P}$, to be detected through a suitable of *nearest-neighbor* classification scheme.

A key intuition is that the comparison of few "close" neighbors provides enough information, in order to establish an appropriate cluster membership. Therefore, the latter can be detected with a minimal number of comparisons by considering, for each new tuple, only some relevant neighbors efficiently extracted from the current database through a proper retrieval method. The algorithm in figure 2 summarizes our

solution to the data de-duplication problem. The clustering method is parametric w.r.t. the distance function used to compare any two tuples and is defined in an incremental way, so that to enable the integration of a new set of tuples into a previously computed partition. Indeed, the algorithm receives a database $\mathcal{DB}$ and an associated partition $\mathcal{P}$, besides the set of new tuples $\mathcal{DB}_\Delta$; as a result, it produces a new partition $\mathcal{P}'$ of $\mathcal{DB} \cup \mathcal{DB}_\Delta$, obtained by adapting $\mathcal{P}$ with the tuples from $\mathcal{DB}_\Delta$.

Figure 1: Clustering algorithm

More specifically, for each tuple μ_i in $\mathcal{DB}_\Delta$ to be clustered, the neighbors of μ_i are retrieved by means of the procedure kNearestNeighbor, which searches for the K most prominent neighbors within a bounded range δ from the query tuple μ_i. The cluster membership for μ_i is determined by calling the MostLikelyClass procedure, which estimates the *likeliest* cluster among the ones associated with the neighbors of μ_i. Such an estimation is carried out via a *voting* strategy, where each neighbor μ_j of μ_i votes for the cluster it belongs to, by adding a contribution $\frac{1}{dist(\mu_i,\mu_j)}$ to the score of its cluster. Figure 2 details the process.

Figure 2: Establishing cluster membership

The score of each cluster is normalized by dividing it by the number of tuples that voted for the cluster; tuple μ_i is then assigned to the cluster receiving the highest normalized score, provided that this is greater than a given threshold – in our usual setting we use 0.5 for the threshold. Otherwise, μ_i is estimated not to belong to any of the existing clusters with a sufficient degree of certainty and, hence, it is assigned to a newly generated cluster.

Interestingly, votes from neighbors may alternatively be used for estimating either a probabilistic or a fuzzy degree of cluster membership for μ_i, so that to allow μ_i to belong to more than one cluster. However, in the envisaged applicative scenario, pairwise dissimilarity among data tuples is quite high and duplicates of an individual domain entity are not likely to appear also as potential duplicates of some other entity. For this reason, we focus on exclusive clustering in the rest of the paper.

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. In particular, for each tuple μ_j within the set of neighbors in input to Propagate, the membership of μ_j is estimated again by MostLikelyClass and, if it does not agree with the cluster actually containing μ_j, the membership of μ_j is updated and Propagate is recursively applied to the neighbors of μ_j.

Notice that, in figure 2, the execution of the Propagate procedure is naturally triggered at line **??** of the Generate-Clusters scheme whenever the cluster membership of the current tuple μ_i can be estimated with a sufficient degree of certainty. In all other cases, the Generate-Clusters scheme simply creates a new cluster $\mathcal{C}_{m+1}$ and assigns μ_i to $\mathcal{C}_{m+1}$ without invoking Propagate. This avoids relocating, in the absence of sufficient certainty, those tuples whose cluster membership was confidently established. Nevertheless, this behavior simply postpones the invocation of the Propagate procedure, in an attempt at accumulating sufficient certainty for the new cluster $\mathcal{C}_{m+1}$, i.e. a normalized score for $\mathcal{C}_{m+1}$ over the pre-specified threshold γ. Indeed, the incremental assignment of further tuples to $\mathcal{C}_{m+1}$ collects potential voting neighbors. As the size of cluster $\mathcal{C}_{m+1}$ increases, its elements tend to confidently vote in favor of $\mathcal{C}_{m+1}$ itself. At a certain point in time, any assignment to $\mathcal{C}_{m+1}$ determines again the invocation of the Propagate procedure.

Also, since the Propagate call is caused by an insertion of a new object not yet considered in the partition, the assignment of this object to a cluster has the effect of enlarging the cluster borders. Hence the possibility of attracting further objects into this cluster. Thus, a chain effect may cause further relocations, which eventually end up with neighbor clusters merged. In principle, this task might cause all clusters to merge, thus involving the whole dataset $\mathcal{DB}'$. However, in typical de-duplication settings,

where clusters are quite distant from each other, the propagation affects only a reduced number of tuples and converges within few iterations.

Notably, the complexity of Algorithm 2, given the size N of $\mathcal{DB}$ and M of $\mathcal{DB}_\Delta$, depends on the three major tasks: the search for neighbors (line **??**, having cost $f(N)$), the voting procedure (line **??**, with a cost proportional to K), and the propagation of cluster labels (line **??**, having a cost proportional to n, based on the discussion above). Being performed for each tuple in $\mathcal{DB}_\Delta$, the overall complexity is $O(M(f(N) + K))$. Since K is $O(1)$, it follows that the main contribution to the complexity of the clustering procedure is due to the cost $f(N)$ of the kNearestNeighbor procedure.

Therefore, the main efforts towards computational savings are to be geared along the design of an efficient method for neighbor search. In pursuing such a goal, we minimize the number of database accesses and avoid the computation of all pair-wise distances.

3 Optimizing the Search for Neighbors

The retrieval of neighbors in the clustering algorithm of figure 2 can be performed by using an indexing scheme, that supports the execution of similarity queries and can be incrementally updated with new tuples.

A substantial improvement to the performance of the Generate-Clusters algorithm can be achieved by exploiting a hash-based indexing scheme, which could guarantee, on average, the execution of neighbor searches in a time that does not depend on the number of database tuples.

A basic idea is to map any tuple to a proper set of features, so that the similarity between two tuples can be evaluated by simply looking at their respective features. Under this perspective, the role of the hashing method is to maintain the association between tuples and the corresponding features, so that the neighbors of a tuple μ can be efficiently computed, by simply retrieving the tuples that appear in the same buckets as μ.

To this purpose, a hash-based index structure, simply called *Hash Index*, is 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 through an optimized usage of disk pages: each bucket gathers tuples that are estimated to be similar to each other, for they sharing 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 3 illustrates how such an index can be exploited for performing nearest-neighbor searches and, then, supporting the whole clustering approach previously described. The algorithm works according to two main parameters: the number K of desired neighbors and the maximum allowed distance δ from the query tuple μ. It is worth noticing that both the indexing of a tuple and the retrieval of its neighbors are based on generating relevant features for the tuple itself.

Figure 3: The kNearestNeighbor procedure.

The algorithm uses two auxiliary structures, namely the set S of features to be generated as well as the set $\mathcal{N}$ of neighbor tuples to return as an answer. For convenience, tuples in $\mathcal{N}$ are sorted according to their distance from the query tuple μ.

Lines **??**-**??** specify how the set $\mathcal{N}$ is filled. First, a feature x is extracted (line **??**) and the *FI.Search* method is exploited to retrieve the logical address of the bucket associated with x. For each of these buckets lines **??**-**??** iteratively extract the indexed tuples (using *ES.Read*) and try to insert them into $\mathcal{N}$. Specifically, a tuple ν can be inserted within $\mathcal{N}$ in two cases: *(i)* either the size of $\mathcal{N}$ does not exceed its capacity, or *(ii)* $\mathcal{N}$ capacity is K, but it contains an element whose distance from μ is higher than the distance between ν and μ – actually, $\mathcal{N}.MaxDist()$ here denotes the maximum distance between μ and

any tuple in $\mathcal{N}$. If needed, the element least similar to μ is removed from $\mathcal{N}$, in order to make room for ν. As a side effect, the algorithm updates FI and ES, in order to correctly refer to the novel tuple μ.

3.1 Naïve Hashing based on Exact Matching

A fundamental aspect in our approach concerns the choice of features for indexing tuples, which heavily impacts the effectiveness and efficiency of the overall methodology, and should be carefully tailored to the criterion adopted for comparing tuples. An interesting starting point is proposed in [9], that defines an indexing scheme for the retrieval of similar tuples, relying on 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, $dist_{\mathcal{M}}$ is assumed to correspond to the Dirichlet function. Hence, the dissimilarity between two itemsets is measured by evaluating their degree of overlap.

In this case, a possible strategy for 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 for a given tuple is exponential in the cardinality of the tuple itself, the method was tuned to produce a minimal set of "significant" subkeys, which yet allow to retrieve all the tuples, whose distance from μ is lower than a specified threshold δ. In particular, a subkey s of μ is said *δ-significant* if $\lfloor |\mu| \times (1 - \delta) \rfloor \leq |s| \leq |\mu|$. Notice that, if we restrict ourselves to 1-subkeys, then the indexing scheme behaves exactly like an inverted-list index. However, inverted indexes do not guarantee that a minimal set of candidate tuples is retrieved.

Notably, any tuple ν such that $dist_J(\mu, \nu) \leq \delta$ must contain at least one of the δ-significant subkeys of μ [9]. Indeed, $dist_J(\mu, \nu) \leq \delta$ implies that

$$|\mu \cap \nu| \geq |\mu \cup \nu| \times (1 - \delta) \geq |\mu| \times (1 - \delta)$$

from which it follows that $|\mu \cap \nu|$ is a δ-significant subkey of μ.

Therefore, searching for tuples that include at least one of the δ-significant subkeys derived from a tuple μ, constitutes a strategy for retrieving the neighbors of μ without scanning the whole database. Such a 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 ν is retrieved for comparison with μ is low.

Despite its simplicity, the indexing strategy sketched above was proven to work quite well in practical cases [9]. 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 does not recognize nearly identical tokens, i.e. highly resemblant tokens, that exhibit few syntactic variations. As a consequence, any two tuples made up of nearly identical tokens would not be treated as duplicates. As an example, the tuples

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

   would not be recognized as duplicates, even though they clearly refer to the same entity, due mainly to semantically irrelevant syntactic differences between pairs of corresponding tokens, that cannot be properly treated via exact matching. Notice that, lowering the degree δ of dissimilarity, partially alleviates such an effect, though considerably worsening the performances of the index.

4 Hierarchical Approximated Hashing based on q-Grams

We here define a hash-based index that overcomes the aforementioned limitations. In particular, we aim at developing a key-generation scheme, that allows a constant number of disk writes and reads, being simultaneously capable of keeping a fixed (low) rate of false negatives. Notice that these are contrasting objectives in the approach described in section 3.1, since in order to keep I/O operations low we need to generate few δ-significant subkeys, whereas low values of δ produce more 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* [6, 8], 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 [5].

A *locally sensitive* hash function H for a set S, equipped with a distance function D, is a function that bounds the probability of collisions to the distance between elements. Formally, given H, for each pair of elements $p, q \in S$ and each distance value ϵ, there exist values P_1^ϵ and P_2^ϵ such that

- if $D(p, q) \leq \epsilon$ then $\Pr[H(p) = H(q)] \geq P_1^\epsilon$, and
- 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 tuple 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 purpose, we can exploit the theory of minwise independent permutations [8]. 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 that x is associated with the minimum code 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 $H(X) = \min_{x \in X}(\pi(x))$. 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 [3, 5] 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.

The theory of minwise independent permutations can even help us in solving the problem of bounding the number of I/O operations. Indeed, the generation of δ-significant subkeys can be avoided by resorting to a further minwise hash function defined over the tuples. If two tuples μ and ν exhibit $dist_J(\mu, \nu) = \delta$, then a minwise encoding H specifically tailored to tuples guarantees $\Pr[H(\mu) = H(\nu)] = 1 - \delta$. Hence, H can contribute to build the set S of relevant features to exploit in the kNearestNeighbor procedure of figure 3, by identifying a feature of μ with $H(\mu)$. By exploiting a fixed number of different encoding functions, we populate the set S with a controlled number of features to be exploited within the index.

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

- at 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.

- at 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 purged tuples sharing several codes are associated with a same key.

The key resulting from the final, second-level coding can be effectively adopted in the indexing structure described in section 3.

As an example, let us consider the tuples

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

By exploiting 1-grams, $dist_{\mathcal{M}}(\texttt{Jeff}, \texttt{Jef}) = 0$, $dist_{\mathcal{M}}(\texttt{Lynch}, \texttt{Lync}) = 0.2$, $dist_{\mathcal{M}}(\texttt{Maverick}, \texttt{Maverik}) = 0.125$ and $dist_{\mathcal{M}}(\texttt{Rd}, \texttt{Road}) = 0.5$. An appropriate minwise function would hence likely associate the same code to the first 3 terms and a distinct code to the remaining terms with higher syntactic difference. Hence, a first-level encoding would likely result in the transformations $rep(\mu_1) = \{h_1, h_2, h_3, h_4, h_5, h_6\}$ and $rep(\mu_2) = \{h_1, h_2, h_3, h_7, h_5, h_6\}$. Notice now that $dist(rep(\mu_1), rep(\mu_2)) = 0.285$: as a consequence, a second-level minwise hash function would likely associate the same code to both $rep(\mu_1)$ and $rep(\mu_2)$. This would allow to achieve an effective indexing strategy in support of the kNearestNeighbors procedure.

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 truly random, the resulting probability that a generic $x \in X$ is mapped in 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 [8], i.e., the functions $\pi(x) = (ac(x) + b) \bmod p$, where $a \neq 0$ and $c(x)$ is a unique numeric code associated with x (such as, e.g., the code obtained by the concatenation of the ascii characters it includes). Provided that a, b, $c(x)$ and p are sufficiently large, 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) as shown in figure **??**. Recall that a hash function on π is defined as $H_\pi(X) = \min_{x \in X}(\pi(X))$, 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 $H_1^l, \ldots, H_h^l$ (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 $\epsilon > \frac{1}{2}$ 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 $(1 - \epsilon)^h$.

Figure 4: The hierarchical key-generation procedure.

In general, allowing several trials generally flavors high probabilities. Consider the case where $\epsilon < \frac{1}{2}$. 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 $H_1^u, \ldots, H_k^u$, 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 $rep_i(\nu)$.

5 Experimental Results

In this section we study the behavior of the GENERATE-CLUSTERS algorithm proposed in figure 2. Experiments are aimed at evaluating whether the proposed indexing methods allow substantial efficiency in the clustering task and whether an appropriate number of clusters is generated. In particular, we study the behavior of the algorithm equipped with both types of hash-based indexing structures, introduced in section 3, to investigate its performances (i.e. its efficiency and effectiveness) in the two cases. Both the indexes are made of basic blocks of a fixed size (1k). Disk usage is evaluated, for each retrieval, by measuring the number of blocks involved in the operation (either read or write). The effects of caching are ignored: in principle, what we show in the graphs is a worst-case analysis. Experiments are conducted on both real and synthesized data. The algorithm was executed on an Intel Itanium processor with 4Gb of memory and 2Ghz of clock speed.

The section is structured as follows. First of all, table 1 summarizes all the different parameters mentioned throughout the section. Subsection 5.1 describes the features of the data exploited for evaluation purposes, as well as the quality measures adopted for assessing the effectiveness of the proposed approaches. In section 5.2 we measure the effectiveness and the efficiency of the *Naive hashing* approach. To this purpose, we equip the search for neighbors with both the Naive hashing scheme and a different, state-of-the-art, indexing scheme. The purpose, here, is to evaluate whether the Naive approach is competitive w.r.t. other methods in the literature. Finally, section 5.3 is devoted to the study of the *Hierarchical Approximated hashing* approach. The main purpose here is to study a suitable setting of the parameters h, k and q influencing the effectiveness of the approach.

Parameter	**Meaning**
q	size of contiguous substrings of a token
h	number of different lower-level encodings
k	number of different upper-level encodings
δ	maximum distance threshold
K	number of neighbors
N	size of the data
C	number of clusters

Table 1: Summary notation for all the parameters

The effectiveness of the *Hierarchical Approximated hashing* approach (see section 4) relies on a proper setting of parameters 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.

We aim at finding suitable values of these parameters, that fix a high correspondence between the retrieved and the expected neighbors of a tuple.

5.1 Experimental Setting

Efficiency was tested by applying the algorithm to the task of de-duplicating tuples representing demographic information in a banking scenario. The real data set used for this purpose consists of a collection of about $105,140$ tuples (not publicly available), corresponding to information about customers of an Italian bank. In particular, each record corresponds to registry information about a customer. Data was preliminary preprocessed by exploiting the techniques detailed in [30]. In addition, records corresponding to the same individual were known from background knowledge: in particular, each tuple exhibits an average of 8 relevant neighbors, and in general distances between two tuples referring to different individuals exhibit high values. Hence, data refers to almost $13,000$ individuals, and different individuals exhibit a high degree of separation.

Effectiveness, on the other side, was tested on synthetic data sets. Data generation was tuned according to the following parameters:

- the average size T of the itemsets associated with each attribute in the tuple;
- the number of clusters C;
- the number of tuples N;
- the percentage of perturbation p.

Each cluster was obtained by first generating a representative of the cluster and, then, producing the desired duplicates as perturbations of the representative itself. The individual cluster-representative is generated by randomly picking a certain number of tokens from one common set of tokens. In principle, this generation scheme allows for any supplied extent of overlap between different representatives. However, we did not generate overlapping representatives, since our intent was to investigate the behavior of the devised approaches in identifying groups of actual duplicates of neatly distinct entities. Within each cluster, a perturbation was obtained by either deleting, adding or modifying an original token within the corresponding cluster-representative. The number of perturbations was governed by a Gaussian distribution with p as mean value. The parameter p was eventually exploited to study the sensitivity of the proposed approaches to the level of noise affecting the data (due, for example, to misspelling errors).

Effectiveness was measured by evaluating the degree of overlap between the expected and the actual number of clusters. To this purpose, we adapted some standard quality measures from the literature. For a generic tuple μ in $\mathcal{DB}$, we are interested in evaluating the number TP_μ of *true positives* (i.e., the tuples which are retrieved from the index and that belong to the same cluster of μ) and compare the latter with the number of *false positives* FP_μ (i.e., tuples retrieved without being neighbors of μ) as well as *false negatives* FN_μ (i.e., neighbors of μ which are not retrieved). The indexes `TP-rate`, `FP-rate` and `FN-rate` represent moving-window averages, i.e., the average values of, respectively, TP_μ, FP_μ and FN_μ, relative to the current portion of the database $\mathcal{DB}_\Delta$ (with Δ ranging from $\emptyset$ to $\mathcal{DB}$). We also introduce some global indicators, i.e., $TP = \sum_{\mu \in \mathcal{DB}} TP_\mu$, $FP = \sum_{\mu \in \mathcal{DB}} FP_\mu$ and $FN = \sum_{\mu \in \mathcal{DB}} FN_\mu$, in addition to the average precision and recall per tuple, i.e. $\textit{precision} = \frac{1}{N} \sum_{\mu \in \mathcal{DB}} \frac{TP_\mu}{TP_\mu + FP_\mu}$ and $\textit{recall} = \frac{1}{N} \sum_{\mu \in \mathcal{DB}} \frac{TP_\mu}{TP_\mu + FN_\mu}$.

5.2 Evaluation of the Naïve Hashing Approach

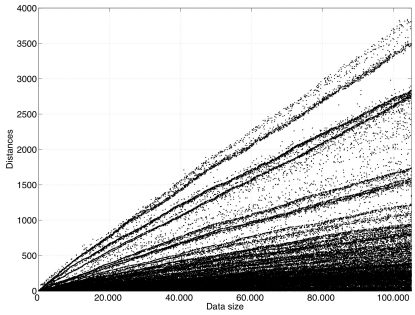
In the first part of our experimentation, we measure the effectiveness and the efficiency of the *Naive hashing* approach. To this purpose, we equip the search for neighbors with both the Naive hashing scheme and a different, state-of-the-art, indexing scheme. The purpose, here, is to evaluate whether the Naive approach is competitive w.r.t. other methods in the literature. The Naive index resides on secondary memory and, hence, it is necessary to evaluate its behavior on disk too. Thus, we consider the following three parameters in our experimentation:

- 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 while updating the index. Since the index has to be incrementally maintained, it is important to evaluate the cost of such a maintenance.

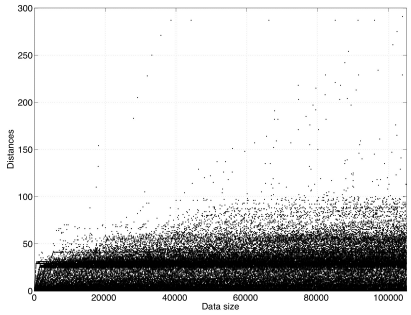
We chose to compare the *Naive hashing* scheme with the *M-Tree* [10] index structure, where tuple proximity is measured in terms of Jaccard distance.

Figures 5 and 6 compare the performance of the clustering algorithm, equipped with both the M-Tree and the Naive hash index structures, on real data. We adopted the M-Tree implementation available on the Web.[1] The tree was tuned by setting node size to 4K and adopting a random split policy. In both techniques we fixed $\delta = 0.2$ and $K = 10$.

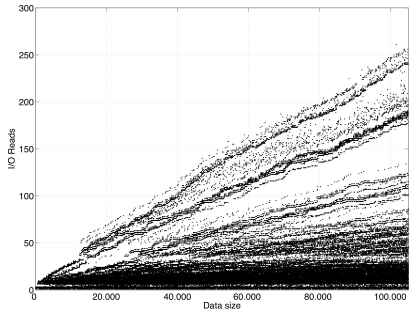
[1] Details can be found at `http://www-db.deis.unibo.it/Mtree/`.



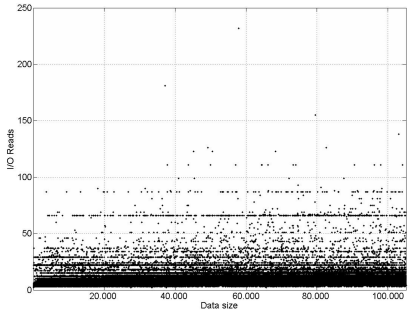
(a) M-Tree: No of distances computed per tuple



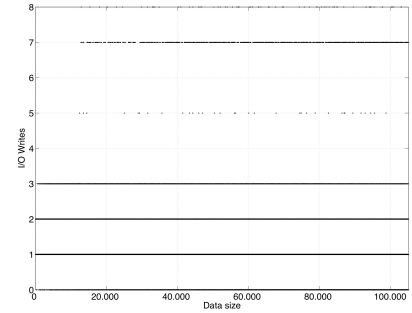
(b) Hash: No of distances computed per tuple



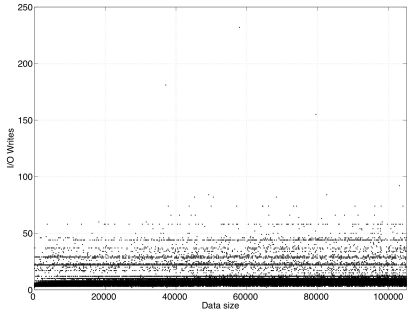
(c) M-Tree: No of I/O Reads



(d) Hash: No of I/O Reads



(e) M-Tree: No of I/O Writes



(f) Hash: No of I/O Writes

Figure 5: Performance comparison

The graphs represent the performances of the approaches w.r.t. the data size. In particular, the horizontal axis represents the portion of data examined thus far. The evaluation of the incremental behavior of the approach can be made by observing whether the increase of the measure under consideration is bounded. This clearly does not happen in figures 5(a) and 5(c), representing, respectively, the number of distances and I/O Reads of the M-Tree approach.

Table 2 summarizes the above graphs, by reporting the amount of comparisons, I/O Reads and I/O Writes for increasing amounts of data stored in the index. Notice that, on average, the amount of comparisons and I/O reads is low. Nevertheless, a search in the M-Tree exhibits a substantially linear behavior in the number of objects stored in the tree. This is also testified by figures 6(a) and 6(b). In these graphs, the performances of the approaches have been averaged on $5,000$ tuples.

Index size		0	10513	21027	31541	42055	52569	63083	73597	84111	94625	105139
Distances	M-Tree	0	15	57	121	201	298	425	620	1013	1956	8719
	Hash	0	0	0	1	5	11	24	28	32	41	291
I/O Reads	M-Tree	1	2	8	12	16	22	30	47	74	139	633
	Hash	2	5	5	6	6	8	8	10	11	15	232
I/O Writes	M-Tree	0	0	0	1	1	1	1	1	2	2	12
	Hash	2	4	5	5	5	6	6	7	8	10	232

Table 2: Performances of the approaches for increasing amounts of examined data

On the other side, the *Naive hashing* scheme exhibits a performance which is bounded by a constant factor, as shown in figures 5(b) and 5(d). In particular, 90% of the tuples retrieve their neighbors by exploiting less than 41 comparisons and 15 disk reads, as shown in Table 2. Also, graphs in figure 6 show a substantial difference in the performance of the approach based on hashing w.r.t. the one based on the M-Tree.

Notice that, at first glance, it seems that the two graphs in figures 6(a) and 6(b) show the same behavior. However, such graphs highlight a subtle difference in the selectivity of both indexes. Indeed, the average number of neighbors retrieved for each disk I/O read is around 10 for the M-Tree; by contrast the hash index detects a lower number of neighbors (1-2 objects per I/O read).

An opposite trend can be seen in the number of disk writes. This is mainly due to the different inner mechanisms, around which the two structures are built. The number of disk writes in the *Naive hashing* 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 proportional to the depth of the tree. Indeed, in order to update its structure, the M-Tree has to select the most appropriate position of the current tuple. After a suitable node has been identified (which does not necessarily correspond to the most suitable node), the tree inserts the tuple in the node and writes the node back to the disk. An overhead is possible only if the insertion causes a node overflow. In such a case, the node is split and the insertion is propagated upward in the tree.

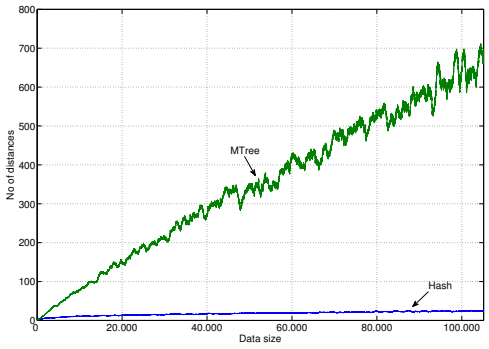
Effectiveness was evaluated by measuring the overlap between the expected number of clusters and the actual number of discovered clusters. Clearly, the latter depends from the K and δ parameters (section 3.1). Since such parameters directly influence an indexing scheme while performing neighbor searches, a major issue is whether the neighbors, retrieved from the index, suffice to perform a correct classification. To properly answer such a question, we exploit the FP-`rate`. The synthetic data sets, used to test the effectiveness of the *Naive hashing* approach, were generated according to the following parameters: $T = [5, 10, 20]$, $N = 100,000$ and $C = 20,000$. Since the *Naive hashing* approach is insensitive to the dissimilarity between single tokens, we set the perturbation percentage $p = 0$.

Figure 7 summarizes the values of the `FP-rate` with $K = 10$ and $\delta = 0.2$. Here, the `FP-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, that ultimately yield a large number of false positives.

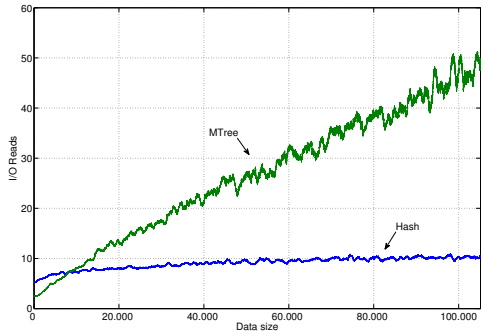
The discussion above shows that, despite its simplicity, the *Naive hashing* approach is more efficient than the M-Tree structure. To this point, we underline that the *Hierarchical Approximated hashing* technique overtakes some deficiencies of our previous approach. In particular, as we show next, it is capable of determining a further improvement in efficiency, without lowering effectiveness.

5.3 Evaluation of the Hierarchical Approximated Hashing Approach

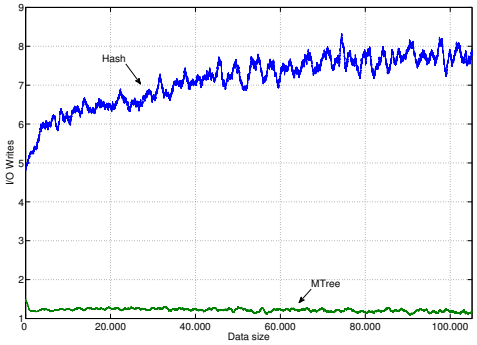
The effectiveness of the *Hierarchical Approximated hashing* approach (see section 4) relies on a proper setting of parameters h and k. Low values of h leverage false negatives, whereas high values leverage false positives. Analogously, low values of k leverage false positives, whereas high values should, in principle, leverage false negatives. We aim at finding suitable values of these parameters, that fix a high correspondence between the retrieved and the expected neighbors of a tuple. We do not compare the running times of *Hierarchical Approximated hashing* to *Naive Hashing*: clearly, the hierarchical approach



(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

Figure 6: Summary of evaluation.

allows to bound the number of accesses to disk (and consequently the number of prospective neighbors discovered) by h (that is, the number of keys associated to a record). Hence, there is a direct way of keeping the running times of the hierarchical approach low.

The values of quality indicators, such as *precision* and *recall*, influence the effectiveness of the clustering scheme of figure 2. In general, high values of precision allow for correct de-duplication: indeed, the retrieval of true positives directly influences the MostLikelyClass procedure, that 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.

The synthetic data set, used to the test the *Hierarchical Approximated hashing* approach, was produced by setting the foresaid generation parameters as follows: $T = 20$, $N = 1,000,000$ and $C = 50,000$ (i.e., an average of 20 tuples per cluster).

We decided to use a larger synthetic data set of $1,000,000$ of tuples to fully test and highlight the scalability of the *Hierarchical Approximated hashing* approach.

Figures 8 and 9 illustrate the results of some tests conducted on the synthesized data, in order to analyze the sensitivity of retrieval to the parameters q, h and k (relative to the indexing scheme) as well as p (relative to 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).

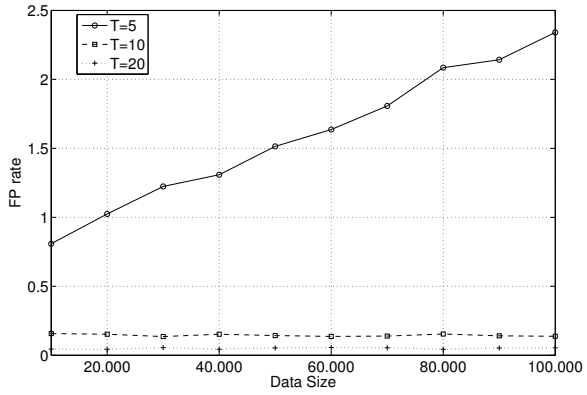
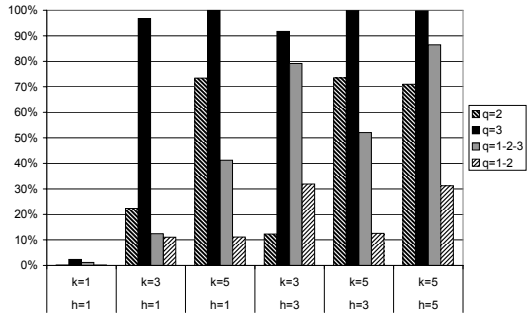
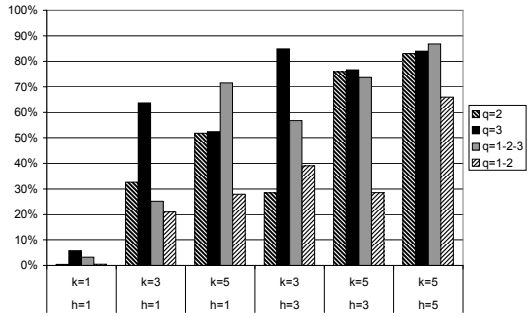


Figure 7: `FP-rate` on artificial data ($K = 10$ and $\delta = 0.2$)

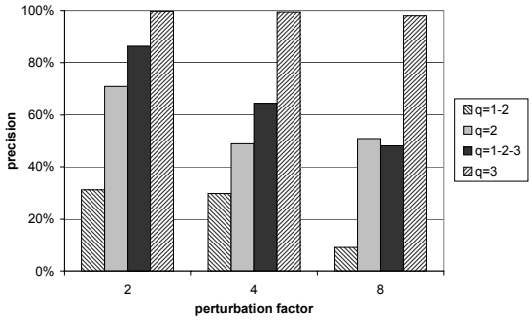


(a) precision vs. h, k and q

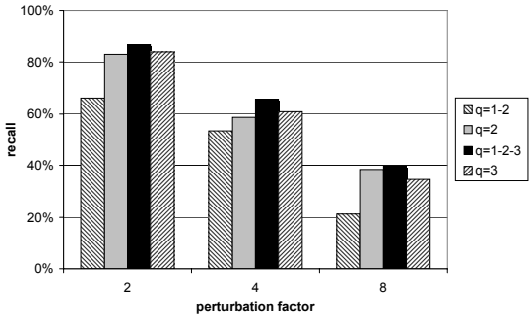


(b) recall vs. h, k and q

Figure 8: Precision and recall on synthetic data w.r.t. q-gram size (q) and nr. of hash functions (h,k)



(a) precision vs. perturbation and q



(b) recall vs. perturbation and q

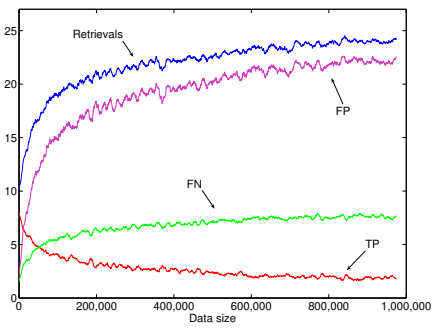
Figure 9: Precision and recall on synthetic data w.r.t. q-gram size (q) and perturbation

Figures 8 (a) and (b) show the results of precision and recall for different values of h and k, with $p = 2$. 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.

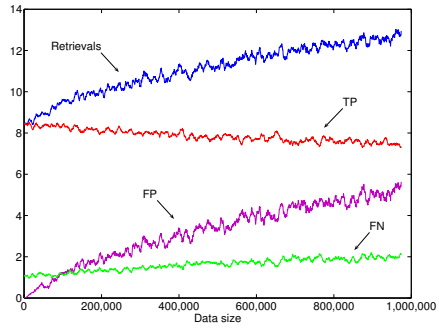
Figures 9(a) and (b) are useful to check the robustness of the index. As expected, the effectiveness of the *Hierarchical Approximated hashing* approach tends to degrade when higher values of the perturbation factor p are used to increase intra-cluster dissimilarity. However, the proposed retrieval strategy keeps on exhibiting values of precision and recall, that can still enable an effective clustering. More precisely, the impact of perturbation on precision is clearly emphasized when tuples are encoded by using also 1-grams, whereas using only either 2-grams 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 10 provides some details on the progress of the number of retrieved neighbors, TP, FP and FN, when an increasing number of tuples is inserted in the index.[2] Conceptually, we can expect a given

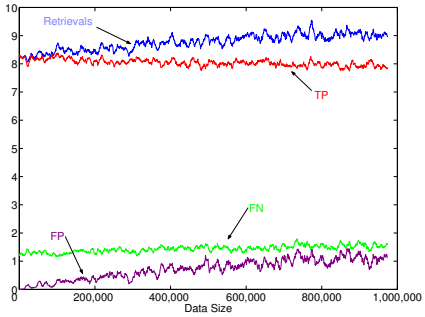
[2]The y-axis in the graphs represent the number of prospective neighbors.



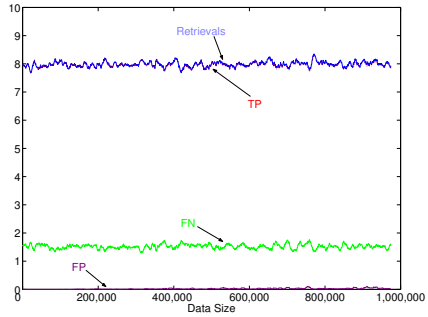
(a) $q = 2, h = 3, k = 3$



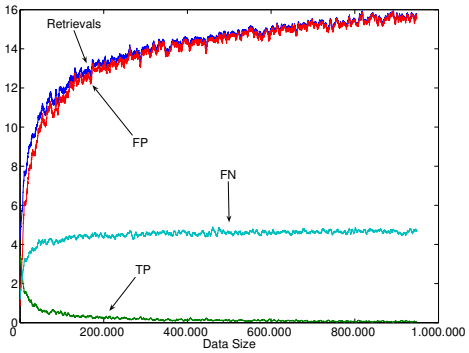
(b) $q = 2, h = 5, k = 5$



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



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



(e) $q = 3$, one level of hashing

Figure 10: Scalability w.r.t. the data size

number n of true neighbors. However, the label `Retrievals` refers to the number of actual neighbors detected by the technique. In practice, `Retrievals` $= TP + FP$ and $n = TP + FN$. In other words, the objects detected by the algorithm are either true neighbors or wrong "side effects", and FN represents the neighbors missed. The graphs are a compact representation of all these measures.

Due to space limitations, 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. As usual, the values are averaged on a window of $5,000$ tuples.

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

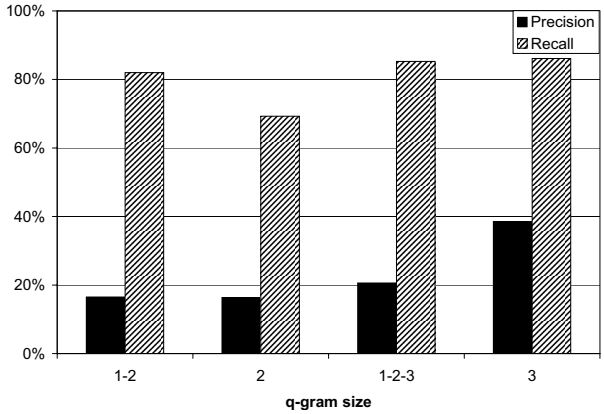
configurations of h, k and q, clearly demonstrates the scalability of this approach. In particular, we highlight that for $q = 3$ the number of retrievals is always very low and nearly independent of the number of tuples inserted (see figures 10 (c) and (d)). In general, the figures confirm the conceptual 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 analysis on effectiveness we previously conducted with the help of figure 8. In particular, notice the rapid decrease of FP and FN when both k and h turn from 3 to 5, in the case of $q = 2$ (figure 10(a) and (b)), that motivates the improvement in both precision and recall observed in these cases. Moreover, the high precision guaranteed by using our *Hierarchical Approximated hashing* approach with q-grams of size 3, is substantiated by figures 10 (c) and (d), where the number of retrieved tuples is very close to the number of TP; in particular, for $k = 5$ (figure 10(d)), the FP curve definitely flattens on the horizontal axis as well as the best behavior obtained by the *Naive hashing* approach in figure 7 for $T = 20$. Figure 10(e) shows the baseline performance of one level of approximated hashing, in which one key is generated for each tuple and token similarity is approximated by means of q-grams with size 3 . A comparison of the performances in figures 10(c) and 10(d) against the one in figure 10(e) is particularly helpful to highlight the actual improvement inherent in using two levels of hashing versus one level of hashing. Notice that, in the latter case, despite the high degree of precision enforced by setting q to 3, retrieved neighbors are essentially false positives and the number of false negatives sensibly increases. As discussed at the end of section 4, the presence of one hash function to provide a single-level encoding of the available data tuples prevents from suitably dealing with false negatives, which has been empirically shown to require multiple first-level encodings in a disjunctive manner. In addition, the absence of a second-level encoding does not allow to cope with false positives, which would conversely require the conjunctive exploitation of multiple second-level encodings.

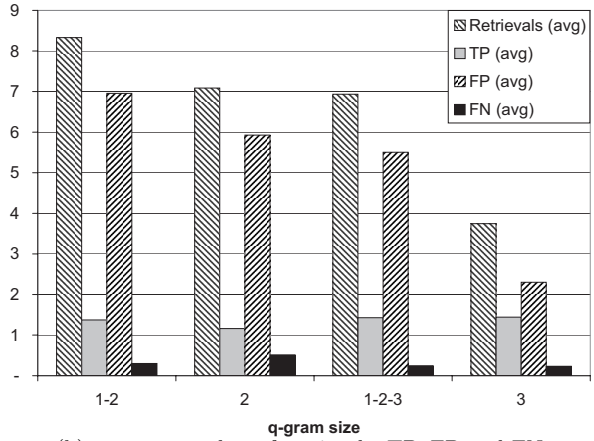
The considerations drawn from the above analysis are confirmed by experiments on real data. Figure 11 (a) shows the results obtained for precision and recall by using different values of q. Notably, precision is measured by exploiting the true identity associated with each record, as anticipated in section 5.1. In practice, this means that real data also contains class information. This ultimately allows us to compute meaningful TP, FP, and FN measures. As we can see, recall is quite high even if precision is low (which still enables an effective clustering). Figure 11 (b) summarizes the average number of retrievals and quality indices. Notice that the average number of retrievals is fairly low, thus guaranteeing a good scalability of this approach.

Figure 11(b) allows a direct comparison with the average number of retrievals obtained by the *Naive hashing* approach in figure 5(b) and, also, shows the superiority in efficiency of the *Hierarchical Approximated hashing* scheme. To this purpose notice that, in the worst case ($q = 1, 2$), the average number of retrievals obtained by the *Hierarchical Approximated hashing* approach is lower than 9, whereas in the case of *Naive hashing* technique, it reaches 50.

We next report on two further aspects related to the incremental de-duplication process based on *Hierarchical Approximated hashing*, i.e. the dependence of both the index size as well as the running time on the number of clusters in the underlying data. This allows to highlight the performance of incremental de-duplication, when the number of clusters is significantly less than $O(N)$. Interestingly, such an objective neatly contrasts with the underlying assumption of the previous tests, where the number of clusters is implicitly $O(N)$ (N represents the size of the underlying data), which is a particularly beneficial setting since the PROPAGATE procedure does not end up reorganizing the whole database. For the evaluation, we studied the behavior of incremental de-duplication obtained by fixing parameters q, h, k to some specific values, that were previously found capable of ensuring effectiveness, on three synthetic datasets of $1,000,000$ tuples including, respectively, $1K$, $10K$ and $100K$ clusters. Specifically, q was set to 3, while both h (i.e. the number of disjunctive hash functions at the lower level of the encoding scheme) and k (i.e. the number of conjunctive hash functions at the upper level of the encoding scheme) ranged into the set $\{3, 5\}$. The number of retrieved neighbors was fixed to 5. Table 3 summarizes the running times expressed in minutes for a certain combination of parameter values over any cluster number, while figure 12 indicates the respective index size. By looking at figure 12 and table 3, one can notice that both the index size and the running time increase when q, h, k are increased. In particular, higher values of h



(a) precision and recall



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

Figure 11: Results on real data using different q-gram sizes

and k result into a larger number of longer keys for each tuple, which requires more storage space and also involves an increasing amount of computation since the kNearestNeighbor procedure must search for such keys within both the *Feature Index* and the *External Store*. Furthermore, the foresaid illustrations reveal that, in general, as the number of clusters in the data increases, the overall running time decreases. The rationale is twofold. Foremost, with a decreasing number of clusters, the kNearestNeighbors procedure must scan, for each newly arrived tuple, a far larger number of candidate neighbors to distill 5 actual neighbors that are nearest to the tuple. Moreover, with a smaller number of clusters, the de-duplication of a new tuple is essentially a further vote in favor of one among fewer classes and, hence, it is likelier to alter the cluster memberships of its neighbors. Sporadic propagation tends to degenerate into massive relocation as the number of clusters is decreased to much less than $O(N)$. Indeed, in such cases the Propagate procedure is invoked at each reassignment of a tuple to a cluster, thus becoming a workload for the incremental clustering process of linear complexity w.r.t. the size of the data at hand.

The index size in figure 12 is also at the basis of one further consideration. More specifically, for a certain number of clusters, the two settings $q = 3, h = 3, k = 3$ and $q = 3, h = 3, k = 5$ lead to an index size that in some cases might in principle fit into main memory, being 4Gb the memory equipment exploited within our experimental setting. Nonetheless, the combination $q = 3, h = 5, k = 5$ produces a sensible growth of the index structure, whose size is far larger than primary memory for any number of clusters in the data. Therefore, in the preceding experiments, the index structure was resident on

secondary memory. Obviously, this inflates the running times reported in table 3, that are adversely affected by the workloads due to the I/O operations on secondary memory.

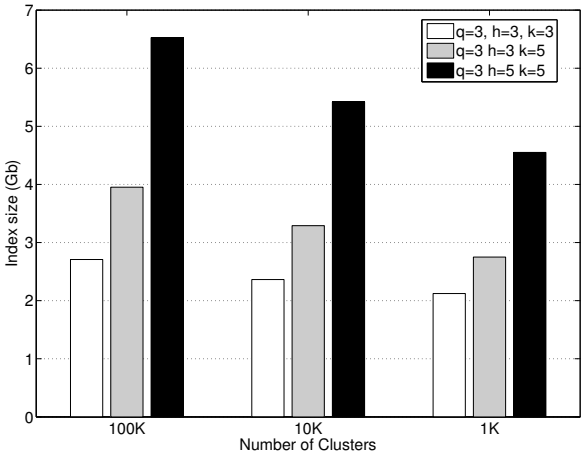


Figure 12: Size of the disk-resident index-structure versus number of clusters

Number of clusters	**Parameter configuration**		
	$q = 3, h = 3, k = 3$	$q = 3, h = 3, k = 5$	$q = 3, h = 5, k = 5$
$1k$	$2,096$	$2,562$	$3,919$
$10k$	$1,062$	$1,155$	$1,765$
$100k$	985	$1,062$	$1,714$

Table 3: Running times in minutes for de-duplication with disk-resident index versus number of clusters

To conclude, figure 13 shows the results of a series of experiments with which we investigated the actual running time of incremental de-duplication when the entire index structure is loaded into main memory. Again, we stressed incremental de-duplication by reducing the number of clusters to considerably less than $O(N)$.

Each curve in figure 13 highlights the performance of incremental clustering for a meaningful combination of parameters h, k. The analysis was conducted on four synthetic datasets of $1,000,000$ tuples including, respectively, 100, $1K$, $10K$ and $100K$ clusters. In particular, we underline that the three datasets with $1K$, $10K$ and $100K$ clusters are exactly the same as those employed in the set of experiments behind figure 12 and table 3. Parameter q was set to 3, while the number of retrieved neighbors was fixed to 5. Tests were performed on an Intel Itanium processor with 16Gb of memory. All other software/hardware equipments are identical to the ones in the original experimental setting. As expected, the computational cost of incremental clustering is still influenced by both the number of hash functions employed at each level and the overall number of clusters in the data. Nevertheless, for each number of clusters, the overall running time is at least one order of magnitude less than the corresponding one in table 3.

6 Related Work

In the following, we shortly review some prominent proposals for the detection and management of duplicated data. As a matter of facts, this problem has given rise to a large body of works in several research communities, where it is referred to with as many umbrella names, such as, e.g., Merge/Purge [27], Record Linkage [25, 40], De-duplication [39], Entity-Name Matching [23], Object Identification [38].

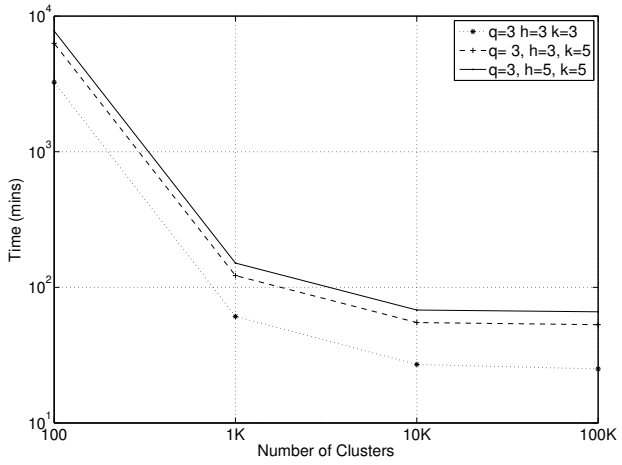


Figure 13: Running times in minutes for de-duplication with memory-resident index versus cluster number

In most of these approaches, a major issue is represented by the definition of a method for comparing objects, especially when information on object identity is carried by textual fields (indeed, these are subject to various kinds of heterogeneity and mismatches across different information sources). To this purpose, in addition to classical string (dis)similarity functions [26], several methods [18, 21, 22, 29, 39] were defined, which allow to effectively compare textual information in the context of duplicated data.

Many approaches to the de-duplication problem essentially attempt to match or cluster duplicated records [23, 24, 29], based on suitable similarity functions. Unfortunately, effectiveness is a major point of emphasis for the majority of these approaches, that pay minor attention to scalability and, hence, reveal inadequate under stronger efficiency requirements.

In general, de-duplication approaches can be divided into supervised and unsupervised.

Supervised approaches learn rules characterizing pairs of duplicates from training data, consisting of known duplicates [18, 23]. Such methods assume that training data contain the wide variety of possible errors in practice. However, such a comprehensive collection of training data very rarely exists in practice. Partially, this issue was addressed by approaches based on active learning [39, 41]. These require an interactive manual guidance. In many scenarios, it is not plain to obtain satisfying training data or interactive user guidance.

It is worth noticing that resorting to unsupervised methods, as consolidated clustering algorithms [11–14, 27], could not guarantee an adequate level of scalability either. Indeed, even these algorithms would not work adequately in a situation where far too many clusters are expected to be found, as it does happen in a typical de-duplication scenario, where the number of clusters can be of the same order as the size of the database. To the best of our knowledge, the only suitable approaches appear to be the ones in [20, 42].

Precisely [42] avoids costly pairwise comparisons by grouping objects in "canopies", i.e., subsets containing objects suspected to be similar according to some cheap (i.e. computationally inexpensive) similarity function and, then, computing actual pairwise similarities only within the discovered canopies. Since in a typical duplicate detection scenario there are several canopies, and an object is shared in a very few number of canopies, the main issue in the approach is the creation of canopies. The authors proposed an effective solution to this issue: nevertheless their approach is not incremental. In a sense, our approach also builds canopies (which are collected within the same buckets in the index). The main difference is that the properties of minwise hashing functions allow to approximately detect such canopies incrementally.

In [20], an efficient two-phase approach is proposed: first determine the nearest neighbors of every tuple in the database and, then, partition the original collection into groups of duplicates. The efficiency

of the algorithm strictly relies on the nearest neighbors computation phase, where the availability of any disk-based index (i.e., inverted index associated with edit or fuzzy similarity functions) is assumed. Efficiency comes from the lookup order in which the input tuples are scanned, in order to retrieve nearest neighbors. The order corresponds to a breadth first traversal of a tree, where the children of any node are its nearest neighbors. The benefit consists in accessing, for consecutive tuples, the same portion of the index, thus improving the buffer hit ratio. Our *Hierarchical Approximated hashing* approach allows a more direct control on the quality of the neighbors retrieved, also simulating (by means of q-grams) the edit distance behavior. This could lead to better values of precision and recall measures. Moreover, with respect to [20], our approach is completely incremental.

There is a plenty of approaches for distance-based search in metric spaces (see, e.g., [28, 43] for a survey). Again, these approaches suffer from the high dimensionality of the space, where search is performed, as described in [44]: indeed, high dimensionality causes too sparse regions to analyze and, thus, invalidates the proposed index methods. Recently, some approaches have been proposed [1, 4, 5], that exploit efficient indexing schemes based on the extraction of relevant features from the tuples at hand. Such approaches could be adapted to deal with the problem of de-duplication, even though they are not specifically designed to approach the problem from an incremental clustering perspective, as we instead discussed here.

Approaches to de-duplication based on locality-sensitive hashing have been proposed in the literature [6, 7,15] as well. Precisely, locality-sensitive hashing was originally developed in [15], as an efficient technique for accurately approximating the nearest-neighbor search problem. Here, the basic idea consists in hashing data points by means of suitable locality-sensitive hash functions, that bound the probability of collisions to the distance between the points: i.e. similar data points are likelier to be assigned to a same bucket than dissimilar ones. The approach in [6] refines this basic strategy in several respects, among which new theoretical guarantees on the worst-case time required for performing a nearest neighbor search and the generalization to the case of external memory. In particular, a high-dimensional space $\mathcal{P}$ of data points is randomly partitioned into hash buckets. The bucket storing each point $p \in \mathcal{P}$ is identified by a corresponding k-bit signature, that is suitably built from p. Since the overall number of buckets identified in this manner can be very large, a second level of standard hashing is exploited to arrange such bitstrings into a single hash table $\mathcal{T}$, whose buckets are directly mapped to disk blocks. In general, it is possible to loose proximity relationships if a point and its nearest neighbor are hashed to distinct buckets. Therefore, in order to lower the probability of such an event, the technique stores a same point into l hash tables $\mathcal{T}_1, \ldots, \mathcal{T}_l$, respectively indexed by as many independently-constructed bitstrings. At retrieval time, the query point q is hashed to a bucket within the individual hash tables $\mathcal{T}_1, \ldots, \mathcal{T}_l$. Objects previously hashed within the same buckets are collected as candidate neighbors. An exhaustive search is then carried out across these candidates, in order to find the neighbors closest to q. These are guaranteed to be at a distance from q within a small error factor of the corresponding optimal neighbors. The technique requires the identification of an optimal, data-specific tradeoff between two contrasting aspects of the index [45], namely its accuracy and required storage space. By increasing l, accuracy is guaranteed for the great majority of queries, through a correspondingly larger number of hash tables. However, this makes the storage requirement inversely proportional to the error factor. Also, it raises the number of potential neighbors and, hence, the overall response time. In such a case, one may act on the signature size k, since a high value of such a parameter would sensibly lower the number of collisions and, hence, mitigate the increase in response time. Unfortunately, large values of k augment the miss rate. By the converse, small values of parameter l cannot guarantee accuracy for all queries.

In our approach, the tradeoff between accuracy and storage space is much less challenging, since there exists a single hash-based index. Here, guaranteeing accuracy for all queries, i.e. lowering both the false positive and false negative rates, can be simply achieved by hashing a same tuple into as many buckets of the index as the number of lower-level encodings of the tuple itself (for each such a representation, the concatenation of its upper-level encodings yields the hash key associated to the tuple). In practice, a very limited number of encodings suffices to enforce high values of precision and recall for similarity search, at the cost of a compact storage space. Yet, from massive empirical evidence, the number of retrievals for

each tuple is always bounded, across all configurations of lower- and upper-level encodings. This general behavior determines the efficiency and scalability of the overall de-duplication process.

An approach for identifying near duplicate Web pages is proposed in [7]. Here, each Web page is first tokenized and then represented as the set of its distinct, contiguous n-grams (referred to as shingles). The most frequent shingles are removed from the set to both improve performance and avoid potential causes of false resemblance. After preprocessing, near duplicates are identified via a clustering strategy that consists of the following four steps. A sketch is computed for each Web page, by applying a suitable min-wise independent permutation to its shingle representation. Sketches are then expanded to generate a list of all the individual shingles and the Web pages they appear in. Subsequently, this list is exploited to generate a new list of all the pairs of Web pages with common shingles, along with the number of shared shingles. Clustering is eventually achieved by examining the triplet elements of the latter list. If a certain pair of Web pages exceeds a pre-specified threshold for resemblance (estimated by the ratio of the number of shingles they have in common to the total number of shingles between them), the two Web pages are connected by a link in a union-find algorithm, that outputs final clusters in terms of connected components.

The algorithm requires a considerable amount of time and space on disk, especially due to the third phase, which makes it unscalable. Optimizations based on the notion of super-shingle addressed such an aspect, although these do not properly work with short Web pages, which corresponds to the case of small sequences of tokens, addressed in this paper.

Yet, the de-duplication process strictly requires that the resemblance threshold is very high to effectively prune several candidate pairs of similar Web pages. Lower values of the threshold, corresponding to a typical setting for similarity search, cause several negative effects. False positive candidates are not appropriately filtered, which lowers precision. Very low values of the similarity threshold may also diminish false negatives, with a consequently moderate increase in recall. However, in such cases, the impact on effectiveness of a small gain in recall would be vanished by the corresponding (much larger) loss in precision. As a further remark, the overall clustering strategy is not incremental.

Similarity-search plays a major goal also in the context of our procedure for the retrieval of neighbors of a given tuple. In principle, this could be achieved by incorporating in our clustering algorithm an indexing scheme that supports the execution of similarity queries and can be incrementally updated with new tuples. A possible solution is the adoption of the *M-Tree index* [10], a well-known, state-of-the-art index/storage structure, which looks like a n-ary tree. The M-Tree structure 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). Similarity search can be accomplished by traversing the tree and ignoring those subtrees, reputed uninteresting for the search purpose. The index exhibits the following major features. Firstly, it is a paged, balanced, and dynamic secondary-memory structure able to index data sets from generic metric spaces. Secondly, similarity range and nearest-neighbor queries can be performed and results can be ranked with respect to a given query object. Thirdly, query execution is optimized to reduce both the number of pages read and the number of distance computations. These peculiarities, combined with its generality, make the M-Tree particularly worthwhile to consider in our setting. However, the benefits of this indexing structure is likely to degrade in a typical entity resolution scenario. Notably, in such a setting, 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. This causes a general degradation in the performance of the tree structure.

The exploitation of a hash-based indexing scheme substantially improves the performance of the Generate-Clusters algorithm. Indeed, on average, it guarantees the execution of neighbor searches in a time that does not depend on the number of database tuples.

The problem of finding tuples that are similar to a certain query record has been intensively studied within the database and information retrieval communities.

Some works from the database community, such as [33] and [34], focused on solving the problem exactly, by defining set-similarity joins, i.e. suitable operators for database management systems. The

techniques behind such operator are referred to as signature-based algorithms. Here, for a collection of input sets, the idea is to yield signatures with the desirable property that, if the similarity of two tuples exceeds a certain threshold, then such tuples share a common signature. By exploiting this property, signature-based schemes find all pairs of sets with common features and, eventually, filter all those sets, whose pairwise similarity actually trespasses the foresaid threshold. An important drawback of the operators in [34] is that their scaling is quadratic w.r.r. input size, which makes it impractical in several applicative domains. [33] enhances the basic scheme of signature-based algorithms in two respects, namely the adoption of a different scheme for computing set signatures and the incorporation of a theoretical guarantee, according to which two highly dissimilar sets are not considered as duplicates with a high probability. Clearly, this considerably lowers the overall number of false-positive candidate pairs and increases the efficiency of the resulting operators, that scale almost linearly in the size of the input set. However, linear scalability requires a non trivial parameter tuning, since no single parameter setting is appropriate for all computations. In practice, for a fixed setting, the operators still scale quadratically and some properties of the input data must be analyzed so that to establish an optimal tuning, that ensures linear scalability.

However, exact search is not always desirable, since even a linear dependence on the database size tends to be critical with volumes of data, thus representing a major limitation for scalability. Rather, in such cases, an approximate answer should be preferred [6]. Due to the peculiarities of the underlying applicative setting, we approach the problem of finding pairwise similar tuples approximately, i.e. from a proximity search perspective. Here, there exist several tuples, that are predominantly dissimilar from one another and, hence, the number K of tuple clusters is expected to be of the same order as the overall number N of tuples. In this context, finding a number of tuples, that are mostly similar to a certain query tuple provides enough information for establishing an appropriate cluster membership for the latter. To this purpose, locality-sensitive hashing is used to build an approximated answer from all those tuples, whose features are similar to the ones within the query tuple. In our de-duplication scenario, relevant neighbors are far closer to the query tuple than the irrelevant ones and, hence, the retrieved neighbors collectively represent an accurate approximation of the exact answer. The exploitation of locality-sensitive hashing for finding similar sets has been proven to deliver performances, that are competitive with the ones provided by the operators in [33]. Interestingly, the basic filtering-effectiveness of our proximity search strategy is guaranteed as in [33]: the probability that two tuples are associated with a similar encoding is proportional to their degree of overlap. Moreover, by suitably combining several min-wise independent permutations, we develop a mechanism with which to somehow govern the false positive and false negative rates. Notice that the effectiveness of [33] in proximity search has not yet been investigated.

Recently, an approach inspired to information retrieval methods [32] proposed to scale exact join-set methods to large volumes of real-valued vector data. This work refines the basic intuition in [34] of dynamically building an inverted list index of the input sets with some major indexing and optimization strategies, mainly concerning how the index is manipulated to evaluate the (cosine) similarity between the indexed records and the query one. From a methodological point of view, our approach exhibits analogies with the ones in [32, 34]. Indeed, a hash-based index is employed to maintain associations between a certain tuple feature and the subset of all available tuples that share that same feature. Also, the index can be incrementally updated as soon as further tuples need be de-duplicated.

What actually differentiates the proposed search strategy from [32, 34] is that these methods essentially pre-compute nearest neighbors, so that retrieving them becomes a dictionary lookup. The devised hash-based indexing strategy does not support neighbor pre-computation. Notwithstanding, it enables neighbor search in a time, that is not dependent on the number of database tuples. In principle, this basic performance may be improved by incrementally building and updating a hash-based dictionary, that stores the identified neighbors of the processed tuples.

Finally, notice that this work embraces the proposal in [9] and improves it in both effectiveness and efficiency. As a matter of facts, by introducing the *Hierarchical Approximated hashing* approach, we gain a direct control over the number of features used for indexing any tuple, which is a major parameter, that critically impacts on the overall cost of the approach. Moreover, we may tune the approach to be

less sensitive to little differences between matching tokens.

7 Conclusions and Future Works

We discussed an incremental technique for de-duplicating sequences within the process for pursuing Entity Resolution in text data.

The de-duplication technique foresees the identification of duplicate information and it is explicitly designed for dealing with the scalability and incrementality issues, that typically arise in this setting. It relies on an incremental clustering algorithm, that aims at discovering clusters of duplicate tuples. To this purpose, we studied two key-generation schemes, that allow a controlled level of approximation in the search for the nearest neighbors of a tuple. These are employed in a coarse-grained approach in subsection 3.1, that fails in those cases where likeliness among single tokens is to be recognized as well, and in a more refined hash-based indexing scheme, in section 4. An empirical analysis conducted both on synthesized and on real data, showed the validity of our coarse-grained approach w.r.t. M-Tree, an alternative, state-of-the-art index scheme. Furthermore, the empirical evaluation also revealed a higher efficiency, with no effectiveness loss, of the field-wise hash-based indexing scheme w.r.t. the coarse-grained approach.

Three challenging issues represent major directions for future research.

A first line of research concerns the incorporation of the underlying database schema into the de-duplication process. Our approach treats the individual database tuple as a whole textual field and, hence, ignores key information provided by the underlying schema of the tuples at hand. Indeed, it considers as nearly duplicates those pairs of data tuples, whose first-level encodings are mostly overlapping. In general, this could originate false positives. The point is that a very large extent of overlap between the first-level encodings of two tuples μ_1 and μ_2 tends to nullify the effects of those rare (though relevant) mismatches, that would instead be expected to prevent μ_1 and μ_2 from being considered as nearly duplicates.

Viewed in this respect, one can exploit the available database schema to refine the basic key-generation scheme. The idea is as follows. Let $\mathcal{S} = \{f_1, \ldots, f_p\}$ denote (suitable groupings of) the schema attributes of the tuples at hand. For each original tuple μ, the first-level encoding yields $rep(\mu) = \{v_1^\mu, \ldots, v_p^\mu\}$, i.e. a properly fragmented encoding, where v_i^μ is the token-by-token purged representation of the attribute value f_i. Subsequently, the second-level encoding separately encodes the distinct field values $v_1^\mu, \ldots, v_p^\mu$. The retrieval of similar tuples is hence accomplished by combining the keys relative to the different fields and exploiting the resulting aggregate key within the hash-based index. This corresponds to consider two tuples as nearly duplicates if their corresponding attributes exhibit strongly matching first-level encodings.

The resulting key-generation scheme implicitly considers the individual (groups of) schema attributes as equally relevant. Indeed, the second-level min-wise hashing simply bounds the probability of collisions, between corresponding fields of two tuples μ_1 and μ_2, to the Jaccard distance of their first-level encodings, i.e. $dist(\mu_1.f_i, \mu_2.f_i) = 1 - \frac{|v_i^{\mu_1} \cap v_i^{\mu_2}|}{|v_i^{\mu_1} \cup v_i^{\mu_2}|}$. The contribution of individual field resemblance to the overall tuple proximity is not further weighted by domain-specific attribute-relevance. This prevents from more effectively recognizing, as near-duplicates, those pairs of tuples with strong matchings over certain attributes, that are actually relevant for the specific application-purpose. In this regard, the naive indexing scheme in subsection 3.1 enables to somehow manipulate the relative relevance of tuple features, by increasing (resp. decreasing) the number of extracted subkeys.

The challenge of identifying a flexible scheme for explicitly weighting (domain-specific) attribute relevance motivates a second direction of further research. Our ongoing effort is at devising a field-wise tuple encoding scheme, that allows to measure the overall dissimilarity of two tuples as the weighted distance of their corresponding attribute values. This is useful in those applicative domains, where two tuples are considered as nearly duplicates if they exhibit similar values in correspondence of some particular fields, whatever their overlap degree over the remaining schema attributes. As an example, in a collection of personal demographic information, two tuples t_1 and t_2 generally refer to the same individual when both have similar values for the SSN, Name and Surname attributes. Possible mismatches over secondary

attributes, such as Street, City, State and Zipcode, may not significantly concur to prevent the recognition of t_1 and t_2 as nearly duplicates. Viewed in this perspective, the possibility of suitably ranking field relevance enables the use of domain-specific attribute-semantic for more effective de-duplication.

Finally, 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 also plan to study the exploitation of more informative similarity functions. An example is the adoption of link-based similarity: recently, some techniques [2] have been proved effective, though affected by the incrementality issues, that are instead a main motivation of our work.

References

[1] R. Ananthakrishna and S. Chaudhuri and V. Ganti. *Eliminating Fuzzy Duplicates in Data Warehouses.* Proc. of Int. Conf. on Very Large Databases, pp. 586–597, 2002.

[2] D.V. Kalashnikov and S. Mehrotra and Z. Chen. *Exploiting Relationships for Domain Independent Data Cleaning.* Proc. of SIAM Conf. on Data Mining, pp. 262–273, 2005.

[3] E. Ukkonen. *Approximate String matching using q-grams and Maximal Matches.* Theoretical Computer Science, 92(1), pp. 191–211, 1992.

[4] S. Chaudhuri and K. Ganjam and V. Ganti and R. Motwani. *Robust and Efficient Fuzzy Match for Online Data Cleaning.* Proc. of ACM SIGMOD Conf. on Management of Data, pp. 313–324, 2003.

[5] L. Gravano and P. G. Ipeirotis and H. V. Jagadish and Nick Koudas and S. Muthukrishnan and Divesh Srivastava. *Approximate String Joins in a Database (Almost) for Free.* Proc of Int. Conf. on Very Large Databases, pp. 491–500, 2001.

[6] A. Gionis and P. Indyk and R. Motwani. *Similarity Search in High Dimensions via Hashing.* Proc. of Int. Conf. on Very Large Databases, pp. 518–529, 1999.

[7] A. Broder and S. Glassman and M. Manasse and G. Zweig. *Syntactic Clustering on the Web.* Proc. of Int. Conf. on World Wide Web, pp. 1157–1166, 1997.

[8] A. Broder and M. Charikar and A.M. Frieze and M. Mitzenmacher. *Minwise Independent Permutations.* Proc. of ACM Symp. on Theory of Computing, pp. 327–336, 1998.

[9] E. Cesario and F. Folino and G. Manco and L. Pontieri. *An Incremental Clustering Scheme for Duplicate Detection in Large Databases.* Proc. Int. Databases and Applications Symp, pp. 89–95, 2005.

[10] P. Ciaccia and M. Patella and P. Zezula. *M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces.* Proc. of Int. Conf. on Very Large Databases, pp. 426–435, 1997.

[11] M. Ester and H. P. Kriegel and J. Sander and X. Xu. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise.* Proc. of Int. Conf. on Knowledge Discovery and Data Mining, pp. 226–231, 1996.

[12] V. Ganti and others. *Clustering Large Datasets in Arbitrary Metric Spaces.* Proc. of Int. Conf. on Data Engineering, pp. 502–511, 1999.

[13] S. Guha and R. Rastogi and K. Shim. *ROCK: A Robust Clustering Algorithm for Categorical Attributes.* Information Systems, 25(5), pp. 345–366, 2001.

[14] S. Guha and R. Rastogi and K. Shim. *CURE: An Efficient Clustering Algorithm for Large Databases.* Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp. 73–84, 1998.

[15] P. Indyk and R. Motwani. *Approximate Nearest Neighbor - Towards Removing the Curse of Dimensionality.* Proc. of Symposium on Theory of Computing, pp. 604–613, 1998.

[16] E. Agichtein and V. Ganti. *Mining Reference Tables For Automatic Text Segmentation.* Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 20–29, 2004.

[17] I. Bhattacharya and L. Getoor. *Iterative Record Linkage For Cleaning And Integration.* Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 11–18, 2004.

[18] M. Bilenko and R. J. Mooney. *Adaptive Duplicate Detection Using Learnable String Similarity Measures.* Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 39–48, 2003.

[19] M. Bilenko and R. J. Mooney. *On Evaluation and Training-Set Construction for Duplicate Detection.* Proc. of KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation, pp. 7–12, 2003.

[20] S. Chaudhuri and V. Ganti and R. Motwani. *Robust Identification of Fuzzy Duplicates.* Proc. of Int. Conf. on Data Engineering, pp. 865–876, 2005.

[21] A. E. Monge and C. P. Elkan. *An Efficient Domain-Independent Algorithm For Detecting Approximately Duplicate Database Records.* Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 23–29, 1997.

[22] W. W. Cohen and P. Ravikumar and S. E. Fienberg. *A Comparison of String Distance Metrics for Name-Matching Tasks.* Proc. of IJCAI Workshop on Information Integration on the Web, pp. 73–78, 2003.

[23] W. W. Cohen and J. Richman. *Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration.* Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 475–480, 2002.

[24] W. Cohen and J. Richman. *Learning to Match and Cluster Entity Names.* Proc. ACM SIGIR Workshop on Mathematical/Formal Methods in Information Retrieval, pp. 13–18, 2001.

[25] I. P. Fellegi and A. B. Sunter. *A Theory for Record Linkage.* Journal of the American Statistical Association, 64, pp. 1183–1210, 1969.

[26] D. Gunsfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[27] M. A. Hernández and S. J. Stolfo. *The Merge/Purge Problem for Large Databases.* Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 127–138, 1995.

[28] G. R. Hjatason and H. Samet. *Index-Driven Similarity Search in Metric Spaces.* ACM Transactions on Database Systems, 28(4), pp. 517–518, 2003.

[29] A. E. Monge and C. P. Elkan. *The Field Matching Problem: Algorithms and Applications.* Proc. of Int. Conf. on Knowledge Discovery and Data Mining, pp. 267–270, 1996.

[30] E. Cesario and F. Folino and A. Locane and G. Manco and R. Ortale. *Boosting Text Segmentation Via Progressive Classification.* Elsevier Journal on Knowledge and Information Systems, 15(3), pp. 285-320,2008.

[31] A. E. Monge and C. P. Elkan. *Automatic Segmentation of Text into Structured Records.* Proc. of ACM SIGMOD Conf. on Management of Data, 2001.

[32] R.J. Bayardo and Y. Ma and R. Srikant. *Scaling Up All Pairs Similarity Search.* Proc. of Int. Conf. on World Wide Web, pp. 131–140, 2007.

[33] A. Arasu and V. Ganti and R. Kaushik. *Efficient Exact Set-Similarity Joins.* Proc. of Int. Conf. on Very Large Databases, pp. 918–929, 2006.

[34] S. Sarawagi and A. Kirpal. *Efficient Exact Set-Similarity Joins*. Proc. of SIGMOD Int. Conf. on Management of Data, pp. 743–754, 2004.

[35] L. Gu and R. A. Baxter and D. Vickers and C. Rainsford. *Record Linkage: Current Practice and Future Directions*. Technical Report, number 03/83. CSIRO Mathematical and Information Sciences, 2003.

[36] M. Cochinwala and S. Dalal and A.K. Elmagarmid and V. S. Verykios. *Record Matching: Past, Present and Future*. 2005.

[37] W. E. Winkler. *The State of Record Linkage and Current Research Problems*. Technical Report. Statistical Research Division, U.S. Census Bureau, 1999.

[38] M. Neiling and S. Jurk. *The Object Identification Framework*. Proc. KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation, pp. 37–39, 2003.

[39] S. Sarawagi and A. Bhamidipaty. *Interactive Deduplication using Active Learning*. Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 269–278, 2002.

[40] W. E. Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. Proc. Section on Survey Research Methods, American Statistical Association, pp. 354–359, 1990.

[41] S. Tejada and C. A. Knoblock and S. Minton. *Learning Domain-Independent String Transformation Weights for High Accuracy Object Identification*. Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 350–359, 2002.

[42] A. K. McCallum and K. Nigam and L. Ungar. *Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching*. Proc. of ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, pp. 169–178, 2000.

[43] E. Chavez and G. Navarro and R. Baeza-Yates and J. Luis Marroquin. *Searching in Metric Spaces*. ACM Computing Survey, 33(3), pp. 273–321, 2001.

[44] R. Weber and H.J. Schek and S. Blott. *A Quantitative Analsysis and Performance Study for Similarity Search in High-Dimensional Spaces*. Proc. of Int. Conf. on Very Large Databases, pp. 194–205, 1998.

[45] M. Bawa and S. Tyson Condie and P. Ganesan *LSH Forest: Self-Tuning Indexes for Similarity Search*. Proc. of Int. Conf. on World Wide Web, pp. 651–660, 2005.

[46] A.K. Jain and M.N. Murty and P.J. Flynn *Data Clustering: A Review. ACM Computing Surveys*, 31(3): 264–323, 1999.

[47] P. G. Ipeirotis and V. S. Verykios and A. K. Elmagarmid *Duplicate Record Detection: A Review. IEEE Transactions on Knowledge and Data Engineering*, 18(1): 1–16, 2007.