Effective Incremental Clustering for Duplicate Detection in Large Databases

Francesco Folino[1], Giuseppe Manco[1],
Luigi Pontieri[1]

RT-ICAR-CS-05-05 **Ottobre 2005**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Effective Incremental Clustering for Duplicate Detection in Large Databases

Francesco Folino[1], Giuseppe Manco[1],
Luigi Pontieri[1]

Rapporto Tecnico N.:
RT-ICAR-CS-05-05

Data:
Ottobre 2005

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, Via P. Bucci 41C, 87036 Rende(CS)

I rapporti tecnici dell'ICAR-CNR sono pubblicati dall'Istituto di Calcolo e Reti ad Alte Prestazioni del Consiglio Nazionale delle Ricerche. Tali rapporti, approntati sotto l'esclusiva responsabilità scientifica degli autori, descrivono attività di ricerca del personale e dei collaboratori dell'ICAR, in alcuni casi in un formato preliminare prima della pubblicazione definitiva in altra sede.

Effective Incremental Clustering for Duplicate Detection in Large Databases

Francesco Folino, Giuseppe Manco, Luigi Pontieri

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

Abstract. We propose an incremental algorithm for discovering clusters of duplicate tuples in large databases. The core of the approach is the usage of a suitable indexing technique which, for any newly arrived tuple t, allows to efficiently retrieve a set of tuples in the database which are mostly similar to t, and which are likely to refer to the same real-world entity that t is associated with. The proposed indexing technique is based on a hashing approach, which tends to assign objects with high similarity to the same buckets. Empirical and analytical evaluation demonstrates the effectiveness of the proposed method in terms of performance, at the cost of limited accuracy loss.

1 Introduction

Recognizing similarities in large collections of data is a major issue in the context of information integration systems. An important challenge in such a setting is to discover and properly manage duplicate tuples, i.e., syntactically different tuples which are actually identical from a semantical viewpoint, for they referring to the same real-world entity. There are several application scenarios involving this important task. A typical example consists in the reconciliation of demographic data sources in a data warehousing setting. Names and addresses can be stored in rather different formats, thus raising the need for an effective reconciliation strategy which could be crucial for decision making. In such cases the problem is the analysis of a (typically large) volume of small strings, in order to reconstruct the semantic information on the basis of the few syntactic information available. Consider, e.g., a banking scenario, in which 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 using a possibly different format for storing the data, de-duplicating demographic tuples is crucial in order to correctly monitor customer behavior.

In such application scenarios, tuples, usually coming from legacy systems, are represented by (small) sequences of strings, and no typing information is available. Thus, by assuming that each possible string represents a dimension along which the information contained in a tuple can projected, tuples represent a small informative content in a high-dimensional space.

In the literature, the problem of tuple de-duplication has been dealt with mainly from an accuracy viewpoint, by taking care to the minimization of incorrect matchings: false positives (i.e., object recognized as similar which actually do not correspond to the same entity) and false negatives (i.e., objects corresponding to the same entity but which are not recognized as similar). However, efficiency and scalability issues do play a predominant role in many application contexts where large data volumes are involved, especially when the object-identification task is part of an interactive application, calling for short response times. Consider again the banking scenario: data collected on a daily basis typically consists of 500,000 instances, representing credit transactions performed by customers throughout the various agencies. In such a case the naïve solution of comparing all database instances in a pairwise manner, according to some given similarity measure, is clearly impractical. For example, for a set of 30,000,000 tuples (i.e., data collected during a 2 months-monitoring), this naïve method would require $O(10^{15})$ (a quadrillion) comparisons. This is clearly infeasible.

In general, the large volume of involved data imposes severe restrictions on the design of data structures and algorithms for data de-duplication, and disqualifies any approach requiring quadratic time in the database size, or producing many random disk accesses and continuous paging activities. Thus, in this paper our main objective is to devise a scalable method for duplicate detection that can be profitably applied to large databases. We approach the problem from a *clustering* perspective: given a set of tuples, our objective is to recognize subsets (clusters) of tuples such that intra-cluster similarity is high, and inter-cluster similarity is low. Three main features make the problem at hand significantly different from traditional approaches:

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

The solution we propose essentially relies on an efficient technique for discovering, in an incremental way, all clusters that contain duplicate tuples. The core of the approach is the usage of a suitable indexing technique which, for any newly arrived tuple t, allows to efficiently retrieve a set of tuples in the database which are likely mostly similar t, and hence are expected to refer to the same real-world entity associated with t. The proposed indexing technique is based on a hashing scheme, which tends to assign objects with high similarity to the same buckets.

We exploit a refined key-generation technique which, for each tuple under consideration, guarantees a controlled level of approximation in the search for the nearest neighbors of such a tuple. To this purpose, we resort to a family of *Locality-Sensitive* hashing functions [1–3], which are guaranteed to assign any two objects to the same buckets with a probability which is directly related to

their degree of mutual similarity. Notably, with the help of an empirical evaluation carried out on synthesized and real data, we can asses that the hashing-based method allows to achieve effective on-line matching, at the expense of limited accuracy loss.

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

2 Problem statement and overview of the approach

In the following we introduce some basic notation and preliminary definitions. An item domain $\mathcal{M} = \{a_1, a_2, \ldots, a_m\}$ is a collection of items. We assume m to be very large: typically, $\mathcal{M}$ represents the set of all possible strings available from a given alphabet. Moreover, we assume that $\mathcal{M}$ is equipped with a distance function $dist_{\mathcal{M}}(\cdot, \cdot) : \mathcal{M} \times \mathcal{M} \mapsto [0, 1]$, expressing the degree of dissimilarity between two generic items a_i and a_j.

A tuple μ is a subset of $\mathcal{M}$. An example tuple is

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

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

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

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

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

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

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

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

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

In more detail, for each tuple μ_i in $\mathcal{DB}$ to be clustered, the neighbors of μ_i are retrieved by means of procedure kNearestNeighbor, which performs a search for the k most prominent neighbors with a bounded range δ and using μ_i as query object. The cluster membership for μ_i is determined by calling the MostLikelyClass procedure, which estimate the *most likely* 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. 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.

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 in its input set, we estimated again, by means of MostLikelyClass, the cluster that μ_j best fit to; if it does not coincide with the cluster actually containing μ_j, the membership of μ_j is updated accordingly, and procedure Propagate is recursively applied to the neighbors of μ_j. In principle, this task might be iterated over each reassigned tuple, and could then be of linear complexity w.r.t. the size of $\mathcal{DB}$. However,

```
GENERATE-CLUSTERS(𝒫,𝒟ℬ_Δ,k,δ)
Output: A partition 𝒫' of 𝒟ℬ ∪ 𝒟ℬ_Δ;
 1: 𝒫' ← 𝒫; 𝒟ℬ' ← 𝒟ℬ;
 2: Let 𝒫' = {𝒞_1,...,𝒞_m} and 𝒟ℬ_Δ = {μ_1,...,μ_n};
 3: for i = 1...n do
 4:    neighbors ← KNEARESTNEIGHBOR(𝒟ℬ', μ_i, k, δ);
 5:    𝒞_j ← MOSTLIKELYCLASS(neighbors, 𝒫');
 6:    𝒟ℬ' ← 𝒟ℬ' ∪ {μ_i};
 7:    if 𝒞_j = ∅ then
 8:       create a new cluster 𝒞_{m+1} = {μ_i};
 9:       𝒫' ← 𝒫' ∪ {𝒞_{m+1}};
10:    else
11:       𝒞_j ← 𝒞_j ∪ {μ_i};
12:       PROPAGATE(neighbors, 𝒫');
13:    end if
14: end for
-----------------------------------------------------------
PROPAGATE(S,𝒫)
P1: for all τ ∈ S do
P2:    neighbors ← KNEARESTNEIGHBOR(𝒟ℬ, τ, k);
P3:    𝒞 ← MOSTLIKELYCLASS(neighbors, 𝒫);
P4:    if τ ∉ 𝒞 then
P5:       𝒞 ← 𝒞 ∪ {τ};
P6:       PROPAGATE(neighbors, 𝒫);
P7:    end if
P8: end for
```

Fig. 1. Clustering algorithm

in typical Entity Resolution settings, where clusters are quite distant from each other, the propagation affects only a reduced number of tuples, and ends in a low number of iterations.

Further details on the clustering scheme sketched above can be found in [4]. What is important to remark here is that the complexity of Algorithm 1, given the size N of $\mathcal{DB}$ and M of $\mathcal{DB}_\Delta$, depends on the three major tasks: the search for neighbors (line 4, having cost n), the voting procedure (line 5, with a cost proportional to k), and the propagation of cluster labels (line 12, having a cost proportional to n, based on the discussion above). As they are performed for each tuple in $\mathcal{DB}_\Delta$, the overall complexity is $O(M(n+k))$. Since k is $O(1)$, it follows that the main contribution to the complexity of the clustering procedure is due to the cost n of the KNEARESTNEIGHBOR procedure.

Therefore, the main efforts towards computational savings are to be addressed when designing an efficient method for neighbor searches. Our main goal is doing this task by minimizing the number of accesses to the database, and avoiding the computation of all pair-wise distances.

2.1 Optimizing the Search for Neighbors

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

We concentrate on hashing schemes here. 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.

Fig. 2 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.

The algorithm uses two auxiliary structures, namely the set S of features to be generated, and 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 3-16 specify how the set $\mathcal{N}$ is filled. First, a feature x is extracted (line 4), and the *FI.Search* method is exploited to retrieve the logical address of the bucket associated with x. For each of these buckets lines 9-13 iteratively extract the tuples it contain (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.

2.2 Naïve Hashing based on Exact Matching

A major point in the proposed approach is the choice of features for indexing tuples, which will strongly impact on the effectiveness and efficiency of the whole

```
kNearestNeighbors(𝒟ℬ,μ,k,δ)
 1: Let S = {s | s is a relevant feature of μ };
 2: 𝒩 ← ∅;
 3: while S ≠ ∅ do
 4:    x = S.Extract();
 5:    h ← FI.Search(x);
 6:    if (h = 0) then
 7:       h ← FI.Insert(x);
 8:    else
 9:       while ν = ES.Read(h) do
10:          if 𝒩.size < k or dist(μ,ν) < 𝒩.MaxDist() then
11:             𝒩.Insert(ν, dist(μ,ν));
12:          end if
13:       end while
14:    end if
15:    ES.Insert(τ, h);
16: end while
17: return 𝒩;
```

Fig. 2. The kNearestNeighbor procedure.

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

In this case, a possible 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. In particular, a subkey s of μ is said δ-significant if $\lfloor |\mu| \times (1 - \delta) \rfloor \leq |s| \leq |\mu|$.

Notably, any tuple ν such that $dist_J(\mu, \nu) \leq \delta$ must contain at least one of the δ-significant subkeys of μ [4]. Therefore, searching for tuples that exhibit at least one of the δ-significant subkeys derived from a tuple μ constitutes a strategy for retrieving all the neighbors of μ without scanning the whole database. 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 μ is retrieved for comparison with ν is low.

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

1. The cost of the approach critically depends on the number of δ-relevant subkeys: the larger is the set of subkeys, the higher is the number of writes needed to update the index.
2. More importantly, the proposed key-generation technique suffers from a coarse grain dissimilarity between itemsets, which does not take into account a more refined definition of $dist_{\mathcal{M}}$. Indeed, the proposed approach is subject to fail, in principle, in cases where likeliness among single tokens are to be recognized as well. As an example, the tuples

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

   are not recognized as similar in the proposed approach (even though they clearly refer to the same entity), due mainly to a dissimilarity between single tokens which is not kept by a simple matching between tokens. Notice that, lowering the degree δ of dissimilarity, partially alleviates such an effect, but at the cost of worsening the performances of the index considerably.

3 Hierarchical Approximate Hashing based on *q*-grams

Our objective in this section is to define an hash-based index which is capable of overcoming the above described drawbacks. In particular, we aim at defining a key-generation scheme which guarantees a constant number of disk writes and reads, yet being capable of keeping a fixed (low) rate of false negatives. Notice that these are contrasting objectives in the approach described in section 2.2, since in order to keep I/O operations low we need to generate few δ-significant subkeys, whereas low values of δ produce several 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* [3, 5], 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 [6].

A *locally sensitive* hash function H for a set S equipped with a distance function D is a function which bounds the probability of collisions to the distance between elements. Formally, given h, for each pair $p, q \in S$ and value ϵ, there exists 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 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 [5]. A minwise independent permutation is a coding function π of a set X of generic items such that, for each $x \in X$, the probability of the code associated with x being the minimum is uniformly distributed, i.e.,

$$\Pr[\min(\pi(X)) = \pi(x)] = \frac{1}{|X|}$$

A minwise independent permutation π naturally defines a locally sensitive hash function H over an itemset X, defined as $H(X) = \min(\pi(x))$. Indeed, for each two itemsets X and Y, it can be easily verified that

$$\Pr[\min(\pi(X)) = \min(\pi(Y))] = \frac{|X \cap Y|}{|X \cup Y|}$$

This suggests that, by approximating $dist_{\mathcal{M}}(a_i, a_j)$ with the Jaccard similarity among some given features of a_i and a_j, we can adopt the above envisaged solution to the problem of false negatives. When $\mathcal{M}$ contains string tokens (as it usually happens in a typical entity resolution setting), the features of interest of a given token a can be represented by the q-grams of a. It has been shown [6, 7] 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 fig. 2, 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:

– In the first level, each element $a \in \mu$ is encoded by exploiting a minwise hash function H^l. This guarantees that two similar but different tokens a and b are with high probability associated with a same code. As a side effect, tuples μ and ν sharing "almost similar" tokens are purged into two representations where such tokens converge towards unique representations.
– In the second level, the set $rep(\mu)$ obtained from the first level is encoded by exploiting a further minwise hash function H^u. Again, this guarantees that 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 2.1.

As an example, let us consider the tuples

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

If 1-grams are adopted, then $dist_{\mathcal{M}}(\texttt{Jeff}, \texttt{Jef}) = 0$, whereas $dist_{\mathcal{M}}(\texttt{Lynch}, \texttt{Lync}) = 0.2$, $dist_{\mathcal{M}}(\texttt{Maverick}, \texttt{Maverik}) = 0.125$ 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 would encode separately the remaining terms. 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 [5], 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 choosing different values of a, b and p) as shown in fig. 3. Recall that a hash function on π is defined as $H_\pi(X) = \min(\pi(X))$, and that $\Pr[H_\pi(X) = H_\pi(Y)] = |X \cap Y|/|X \cup Y| = \epsilon$. Notice that the choice of a, b and p in π introduces a probabilistic bias in H_π, which can in principle leverage false negatives. Let us consider the events $A \equiv$"sets X and Y are associated with the same code", and $B = \neg A$. Then, $p_A = \epsilon$ and $p_B = 1 - \epsilon$. By exploiting h different encodings $H_1^l, \ldots, H_h^l$ (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 $\epsilon > 1/2$ represents the average similarity of items, we can exploit the h different encodings for computing h alternative representations $rep_1(\mu), \ldots, rep_h(\mu)$ of a tuple μ. Then, by exploiting all these representations in a disjunctive manner, we lower the probability of false negatives to $(1-\epsilon)^h$.

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

```
HASH(μ = {a_1, ..., a_n}, k, h, q)
 1: for each a_i ∈ μ do
 2:    compute the q-gram representation q_i of a_i;
 3:    compute h encodings H_1^l(q_i), ... H_h^l(q_i);
 4: end for
 5: for i = 1 to h do
 6:    rep_i(μ) ← {H_i^l(q_j) | a_j ∈ μ};
 7:    for j = 1 to k do
 8:       e_i^j ← H_j^u(rep_i(μ));
 9:    end for
10:    key_i ← e_i^1 ∧ e_i^2 ∧ ... ∧ e_i^k;
11: end for
12: return {key_1, ..., key_h};
```

Fig. 3. The key-generation procedure.

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 $rep_i(\nu)$.

4 Experimental Results

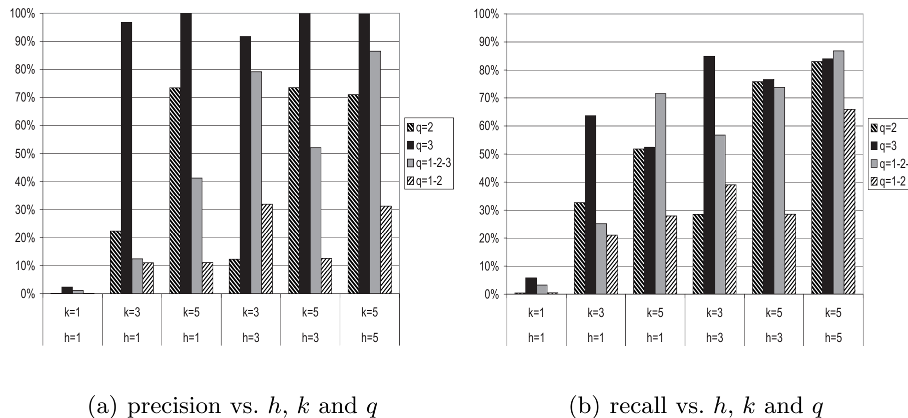
The above discussion shows that the effectiveness of the approach relies on proper values of h and k. Low values of h leverage false negatives, whereas high values leverage false positives. Analogously, low values of k leverage false positives, whereas high values should, in principle, leverage false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

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

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

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

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

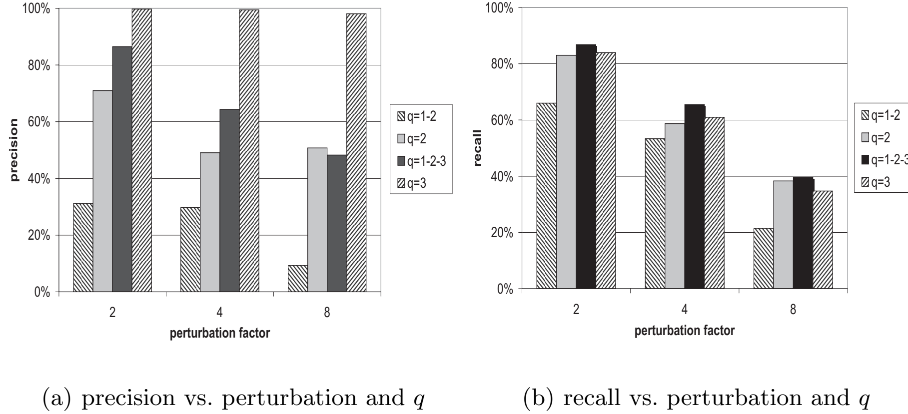


(a) precision vs. h, k and q (b) recall vs. h, k and q

Fig. 4. Results on synthetic data w.r.t. q-gram size (q) and nr. of hash functions (h,k)

Figures 4 and 5 illustrate results of some tests we conducted on this synthesized data, in order to analyze the sensitivity of the retrieval to the parameters q, h and k (relative to the indexing scheme), and p (relative to the noise in the data). In particular, the values of q ranged over 2, 3, 1-2 (both 1-grams and 2-grams) and 1-2-3 (q-grams with sizes 1, 2 and 3).

Figures 4(a) and 4(b) show the results of precision and recall for different values of h and k, and $p = 2$. We can notice that precision raises on increasing values of k, and decreases on increasing values of h. The latter statement does not hold when q-grams of size 1 are considered. In general, stabler results are guaranteed by using q-grams of size 3. As to the recall, we can notice that, when k is fixed, increasing values of h correspond to improvements as well. If h is fixed and k is increased, the recall decreases only when $q = 3$. Here, the best



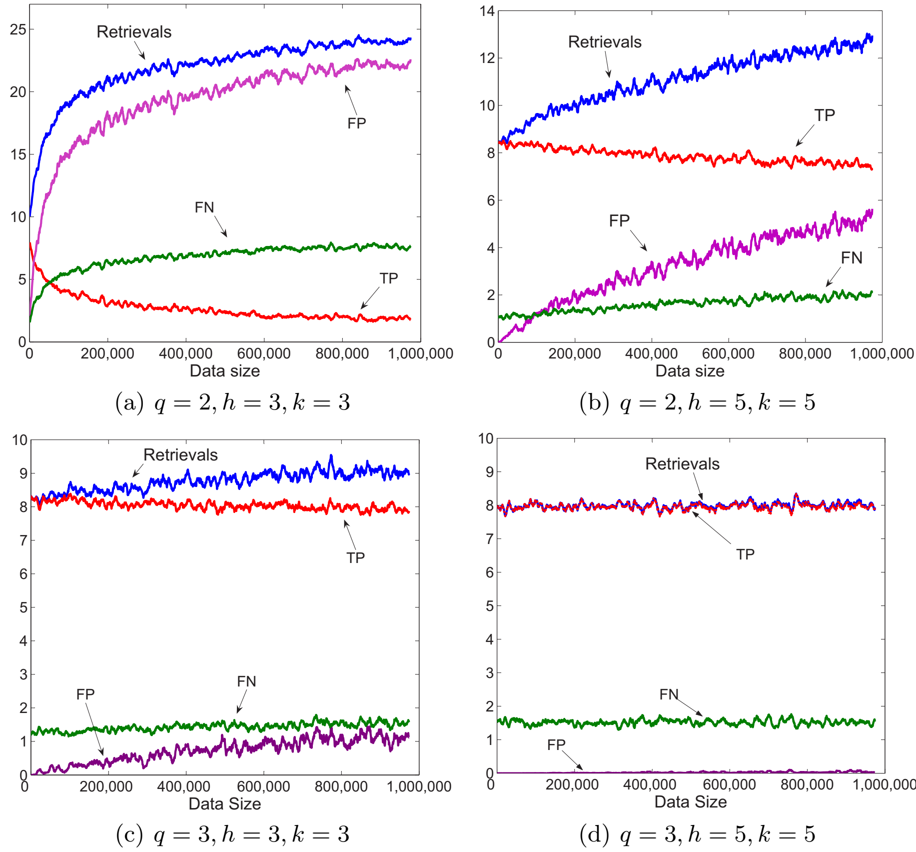
(a) precision vs. perturbation and q (b) recall vs. perturbation and q

Fig. 5. Results on synthetic data w.r.t. q-gram size (q) and perturbation

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 5(a) and 5(b) are useful to check the robustness of the index. As expected, the effectiveness of the approach tends to degrade when higher values of the perturbation factor p are used to increase intra-cluster dissimilarity. However, the proposed retrieval strategy keeps on exhibiting values of precision and recall that can still enable an effective clustering. In more detail, the impact of perturbation on precision is clearly emphasized when tuples are encoded by using also 1-grams, whereas using only either 2-grams 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.

Fig. 6 provides some details on the progress of the number of retrieved neighbors, TP, FP and FN, when an increasing number of tuples, up to 1,000,000, is inserted in the index. For space reasons, we only examine some selected combinations of h and k, and q that were deemed as quite effective in previous analysis. Anyway, we pinpoint that some general results of the analysis illustrated here also apply to other cases. The values are averaged on a window of 5,000 tuples. In general, it is interesting to observe that the number of retrievals for each tuple is always bounded, although for increasing values of the data size the index grows. This general behavior, which we verified for all configurations of h, k and q, clearly demonstrates the scalability of the approach. In particular, we observe that for $q = 3$ the number of retrievals is always very low and nearly independent of the number of tuples inserted (see figures 6(c) and 6(d)). More in general, the figures confirm the conceptual analysis that the number of I/O operations directly depends on the parameter h, the latter determining the number of searches and updates against the index.



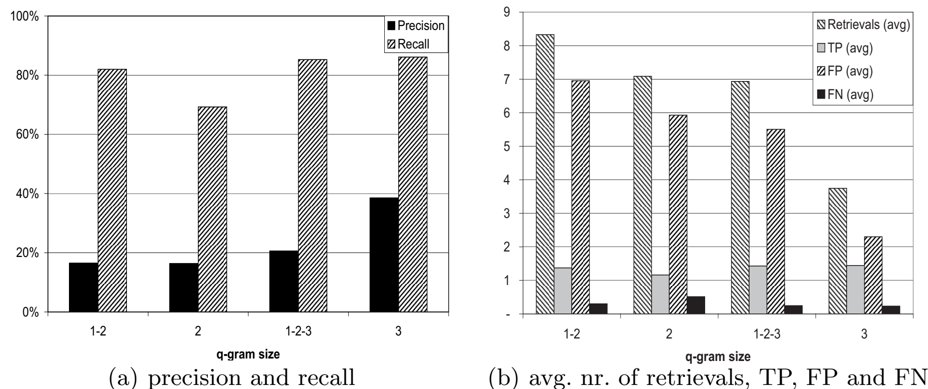
(a) $q = 2, h = 3, k = 3$ (b) $q = 2, h = 5, k = 5$

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

Fig. 6. Scalability w.r.t. the data size

All these figures also agree with the main outcomes of the analysis on effectiveness we previously conducted with the help of fig. 4. In particular, notice the quick decrease of FP and FN when both k and h turn from 3 to 5, in the case of $q = 2$ (see figures 6(a) and 6(b)), that motivates the improvement in both precision and recall observed in these cases. Moreover, the high precision guaranteed by using our approach with q-grams of size 3, is substantiated by figures 6(c) and 6(d), where the number of retrieved tuples is very close to the number of TP; in particular, for $k = 5$ (see fig. 6(d)) the FP curve definitely flatten on the horizontal axis.

The above considerations are confirmed by experiments on real data. Fig. 7(a) shows the results obtained for precision and recall by using different values of q, whereas figure 7(b) summarizes the average number of retrievals and quality indices. As we can see, recall is quite high even if precision is low (thus allowing



(a) precision and recall (b) avg. nr. of retrievals, TP, FP and FN

Fig. 7. Results on real data using different q-gram sizes

for a still effective clustering). Notice that the average number of retrievals is low, thus guaranteeing a good scalability of the approach.

5 Related Work

In the following, we shortly review some relevant proposals for the detection and management of duplicated data. As a matter of fact, this problem has given rise to a large body of work in several research communities, and under a variety of names (such as, e.g., Merge/Purge [8], Record Linkage [9, 10], De-duplication [11], Entity-Name Matching [12], Object Identification [13]). In most of these approaches, a central issue is the definition of a method for comparing objects, especially when information on object identity is carried by textual fields (indeed, the latter are subject to various kinds of heterogeneity and mismatches across different information sources). To this purpose, in addition to classical string (dis)similarity functions [14], several methods [15–18, 11] 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 [19, 12, 15], based on suitable similarity functions. Unfortunately, most of these approaches mainly focus on effectiveness issues, while paying minor attention to scalability, and end up being inadequate under stronger efficiency requirements. It is worth noticing that resorting to consolidated clustering algorithms [8, 20–23], 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 approach appear to be the one proposed in [24]. Here, the authors avoid

costly pairwise comparisons by grouping objects in "canopies", i.e., subsets containing objects suspected to be similar according to a given similarity function, and then computing pairwise similarities only within 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 the approach they propose does not cope with incrementality issues. In a sense, our approach also builds canopies (which are collected within the same buckets in the index), the main difference being that the properties of minwise hashing functions allow to approximately detect such canopies incrementally.

There is a plenty of approaches for distance-based search in metric spaces (see, e.g., [25, 26] for a survey). Again, these approaches suffer from the high dimensionality of the space where search is performed, as described in [27]: indeed, high dimensionality causes too sparse regions to analyse, and thus invalidate the proposed index methods. Recently, some approaches have been proposed [6, 28, 29] which exploit efficient indexing schemes based on the extraction of relevant features from the tuples under consideration. 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 did here.

As previously described, in a previous work [4] we proposed some basic ideas to cope with the duplicate detection problem from an incremental perspective. Specifically, we proposed an indexing scheme tailored to a set-based dissimilarity function, where each tuple was regarded as sets of tokens, and a number of relevant subsets were exploited for indexing it. This paper extends and improves that proposal in both effectiveness and efficiency. First, 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 tune the approach to be less sensitive to little differences between matching tokens.

6 Conclusions and Future Works

In this paper, we addressed the problem of recognizing duplicate information, specifically focusing on scalability and incrementality issues. The core of the proposed approach is an incremental clustering algorithm, which aims at discovering clusters of duplicate tuples. To this purpose, we studied a refined key-generation technique, which allows a controlled level of approximation in the search for the nearest neighbors of a tuple. An empirical analysis, both on synthesized and on real data, showed the validity of the approach.

The approach is quite effective when the available information is based solely on strings, and Jaccard similarity is adopted to compare the features two tuples exhibit. We remark here that the described technique does not consider the database schema, whose adoption (and the consequent separation of the available string into fields) would likely allow to obtain a more refined de-duplication

strategy: as an example, two tuples with the same "name" are more likely to be duplicates than two tuples with the same "city". Nevertheless, the approach we presented here can be easily and effectively adapted to such a situation as well: the overall dissimilarity among tuples can be expressed as a combination of the dissimilarities relative to single fields, and consequently the retrieval of similar tuples can be accomplished by combining the keys relative to different fields and exploiting them within the index.

Clearly, when strings are too small or too different to contain enough informative content, the de-duplication task cannot be properly accomplished by the proposed clustering algorithm. To this purpose, we plan to study the extension of the proposed approach to different scenarios, where more informative similarity functions can be exploited. An example is the adoption of link-based similarity: recently, some techniques were proposed [30] which have been proved effective but still suffers from the incrementality issues which are the focus of this paper.

References

1. Indyk, P., Motwani, R.: Approximate nearest neighbor - towards removing the curse of dimensionality. In: Proc. 30th Symposium on Theory of Computing. (1998) 604–613
2. Broder, A., Glassman, S., Manasse, M., Zweig, G.: Syntactic clustering on the web. In: Proc. 6th Int.WWW Conf. (1997) 1157–1166
3. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Procs. 25st Int. Conf. on Very Large Databases (VLDB'99). (1999) 518–529
4. Cesario, E., Folino, F., Manco, G., Pontieri, L.: An incremental clustering scheme for duplicate detection in large databases. In: Procs. Int. Databases and Applications Symp. (IDEAS'05). (2005) To appear.
5. Broder, A., Charikar, M., Frieze, A., Mitzenmacher, M.: Minwise independent permutations. In: Procs. ACM Symp. on Theory of Computing (STOC'98). (1998) 327–336
6. Gravano, L., et al.: Approximate string joins in a database (almost) for free. In: Procs. 27st Int. Conf. on Very Large Databases (VLDB'01). (2001) 518–529
7. Ukkonen, E.: Approximate string matching using q-grams and maximal matches. Theoretical Computer Science **92** (1992) 191–211
8. Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (1995) 127–138
9. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. Journal of the American Statistical Association **64** (1969) 1183–1210
10. Winkler, W.E.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In: Proc. Section on Survey Research Methods, American Statistical Association. (1990) 354–359
11. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2002) 269–278
12. Cohen, W.W., Richman, J.: Learning to match and cluster large high-dimensional data sets for data integration. In: Proc. 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2002) 475–480

13. Neiling, M., Jurk, S.: The object identification framework. In: Proc. KDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation. (2003) 37–39
14. Gunsfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
15. Monge, A.E., Elkan, C.P.: The field matching problem: Algorithms and applications. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. (1996) 267–270
16. Monge, A.E., Elkan, C.P.: An efficient domain-independent algorithm for detecting approximately duplicate database records. In: Proc. SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery. (1997) 23–29
17. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proc. IJCAI Workshop on Information Integration on the Web. (2003) 73–78
18. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2003) 39–48
19. Cohen, W., Richman, J.: Learning to match and cluster entity names. In: Proc. ACM SIGIR Workshop on Mathematical/Formal Methods in Information Retrieval. (2001)
20. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. (1996) 226–231
21. Guha, S., Rastogi, R., Shim, K.: ROCK: A robust clustering algorithm for categorical attributes. Information Systems **25** (2001) 345–366
22. Guha, S., Rastogi, R., Shim, K.: CURE: An efficient clustering algorithm for large databases. In: Proc. ACM SIGMOD Int. Conf. on Management of Data. (1998) 73–84
23. Ganti, V., et al.: Clustering large datasets in arbitrary metric spaces. In: Proc. Int. Conf. on Data Engineering (ICDE'99). (1999) 502–511
24. McCallum, A.K., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets with application to reference matching. In: Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. (2000) 169–178
25. Chavez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Comput. Surv. **33** (2001) 273–321
26. Hjatason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions on Database Systems **28** (2003) 517–580
27. Weber, R., Schek, H., Blott, S.: A quantitative analsysis and performance study for similarity search in high-dimensional spaces. In: Proc. 24th Int. Conf. on Very Large Databases. (1998) 194–205
28. Ananthakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In: Procs. 28th Int. Conf. on Very Large Databases (VLDB'02). (2002) 586–597
29. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: Procs. ACM Conf. on Management of Data (SIGMOD'03). (2003) 313–324
30. Kalashnikov, D., Mehrotra, S., Chen, Z.: Exploiting relationships for domain independent data cleaning. In: Procs. SIAM'05 Conf. on Data Mining. (2005) 262–273