Fast and Compact Hamming Distance Index

Simon Gog
Institute of Theoretical Informatics, Karlsruhe Institute of Technology
gog@kit.edu

Rossano Venturini
University of Pisa *and* Istella Srl
rossano.venturini@unipi.it

ABSTRACT

Searching for similar objects in a collection is a core task of many applications in databases, pattern recognition, and information retrieval. As there exist similarity-preserving hash functions like SimHash, indexing these objects reduces to the solution of the *Approximate Dictionary Queries* problem. In this problem we have to index a collection of fixed-sized keys to efficiently retrieve all the keys which are at a Hamming distance at most k from a query key.

In this paper we propose new solutions for the approximate dictionary queries problem. These solutions combine the use of succinct data structures with an efficient representation of the keys to significantly reduce the space usage of the state-of-the-art solutions without introducing any time penalty. Finally, by exploiting triangle inequality, we can also significantly speed up the query time of the existing solutions.

1. INTRODUCTION

Indexing large collections of objects for similarity search is a core task of many applications in databases, pattern recognition, and information retrieval (IR). Duplicate and near-duplicate detection in a collection of web pages or content-based retrieval in a collection of images are among the most prominent applications in IR (see e.g., [16, 22, 27] and references therein). In a general framework, each object is modelled with a vector of features and the similarity of two objects is measured by means of a similarity function, e.g., the Jaccard and the cosine similarity, on their vectors. The selection of the most appropriate set of features strictly depends on the application's domain and objectives. For example, the use of words/shingles as features for web pages is a established approach introduced by Broder et al. [5]. In image search, images are modelled as vectors of features which include color histograms, texture features, and edge orientation (see e.g., [27] and references therein). Retrieving the most similar objects by scanning and checking the whole collection is not feasible for two reasons: 1) computing the similarity function between any two objects is demanding because the vectors usually have a high-dimensionality; 2) we are dealing with collections of billions of objects and query streams of the same magnitude today. The combined use of sketching and indexing techniques is the well-established approach for addressing these two issues. Sketching reduces the dimensionality by producing succinct sketches, usually 64 bits, of the vectors, such that the similarity of objects can be estimated from their sketches. Several sketching algorithms have been proposed in the last decades; among them we mention Charikar's SimHash [7], Li and König's b-bit minwise hashing [19], and Mitzenmacher et al.'s Odd Sketches [24].

SIGIR '16, July 17 - 21, 2016, Pisa, Italy
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4069-4/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2911451.2911523

A property of most of these techniques is that if two objects are similar, the Hamming distance of their sketches is small. Thus, the problem of retrieving the most similar objects to an input one boils down to the problem of solving Hamming distance queries on a collection of fixed-length binary keys. This indexing problem is known as the *Approximate Dictionary Queries* problem and was originally formulated by Minsky and Papert [23] as follows. We are given a dictionary $D = \{K_1, K_2, \ldots, K_d\}$ of d binary keys of length m bits each, a k-query Q asks for identifying all the keys in D which are at Hamming distance at most k from Q. While there exist several theoretical solutions which provide guarantees on their space and time complexities [28, 4, 1, 2, 10, 6], the most efficient solutions in practice [22, 20, 26, 29] are based on the multi-index approach proposed by Greene et al. [14]. The main idea of the multi-index is that of splitting each key into blocks of a certain number bits. For each of these block, an index duplicates and organizes the keys in D to be able to efficiently search within the content of this block. A k-query Q is answered in two phases. The searching phase searches with all these indexes for the content of Q's blocks to identify all the keys that match these blocks, possibly up to 1 or more errors. Those keys form a set of candidates which is guaranteed to contain all the query results by properties of the above splitting. Finally, the checking phase scans all these candidates to filter out the ones with Hamming distance larger than k with Q.

Our contributions. We list here our main contributions.

1. We introduce the use of succinct data structures in the searching phase of any multi-index based solution together with a suitable representation of the keys, SIMD-based optimizations, and other tricks. This combination guarantees a space usage reduction that ranges between 18% and 40% depending on the value of k. This does not introduce any new space/time tradeoff

because our solution is always as fast as the best ones and always smaller. This solution is actually faster for smaller values of k ($k = \{2, 3, 4\}$) with a time improving factor that ranges between 2.1 and 3.

The description of these techniques is deferred to the experimental section (Subsection 4.2) because they are motivated and supported by experimental evidences.

2. The cost of the checking phase dominates the overall query time of any multi-index for sufficiently large values of k or dataset sizes. This is because the searching phase has only to identify large ranges of consecutive keys which may contain some query results while the checking phase has the more demanding task of explicitly checking each of these keys. In this paper we introduce the use of the triangle inequality to reduce size of this set of candidates. We propose two solutions based on this idea. The fastest one uses a two-level indexing approach. The second level stores clusters of keys which are close by the Hamming distance while the first level stores just one key to represent each cluster. At query time, all the candidates at the first level are explicitly checked but a fraction of the clusters at the second level can be safely skipped because, by triangle inequality, they cannot contain any query result. The use of this strategy reduces the set of candidates to check by a factor that ranges from 5.3 to 8.4 depending on the dataset and the value of k.

3. In an extensive empirical evaluation we explore the properties of our proposals and compare to a state-of-the-art baseline on real world datasets of sizes up to one billion objects. Overall results show that our time improvement is by a factor up to 3.4 while the space usage reduction ranges between 18% and 40%.

2. RELATED WORK

The theoretical study of the approximate dictionary queries problem has been initiated by Yao and Yao [28], who present a solution in the bit probe model restricting their attention to queries with Hamming distance one (i.e., $k = 1$). Their solution indexes D by using $O(dm \log m)$ bits and solves a query with $O(m \log\log d)$ bit probes. Subsequently, Brodal and Venkatesh [4] improve the above time complexity by a $\log\log d$ factor: their solution uses $O(dm \log d)$ bits of space and $O(m)$ bit probes. In the RAM model, the time complexity is $O(m/w)$, where w is the size of a machine word. The solution by Belazzougui [1] obtains better time and space complexities. Indeed, this solution achieves $O(m + occ)$ query time by requiring optimal $O(dm)$ bits of space. Finally, a solution by Belazzougui and Venturini [2] achieves the same time complexity but works in compressed space. We mention that some of these solutions have been experimental compared in [8].

Solving queries for larger values of k is a much more expensive task. Naïve solutions have either $\Omega(\sum_{i=0}^{k} \binom{m}{i} + \mathsf{occ}) = \Omega(\left(\frac{m}{k}\right)^k + \mathsf{occ})$ query time by using linear space or $O(m + \mathsf{occ})$ query time by using $\Omega(d\binom{m}{k}) = \Omega(d\left(\frac{m}{k}\right)^k)$ words of space, where occ is the number of reported results. In the former case, we index the dictionary to answer membership queries and, given P, we query for the membership of any key obtained by flipping at most k bits in P. In the latter case, we index the superset of keys which contains every key which is at distance at most k from any key in D. These were essentially the best solutions before the work by Cole *et al.* [10]. Their data structure significantly improves these time-space bounds by requiring $O(dm + d\frac{(c_1 \log d)^k}{k!})$ words of space and has $O(m + \frac{(c_2 \log d)^k}{k!} \log\log n + occ)$ query time, where $c_1, c_2 > 1$ are constants. The best linear space solution has $O(m^{k-1} \log n \log\log n + occ)$ query time [6]. However, we are not aware of any attempt to implement these sophisticated solutions.

We point out that there exist solutions which solve variants of this problem which, for example, ask to solve queries under the more difficult Edit distance or admit keys of variable length (i.e., strings) and drawn from larger alphabets. We refer to a survey by Boytsov [3] for a more exhaustive treatment of this topic.

The most efficient solutions in practice [14, 22, 20, 26, 29] are based on the so-called *multi-index* approach introduced by Greene *e*t al. [14]. We defer a detailed description of these solutions to Section 3 because they are the starting point of ours.

3. HAMMING DISTANCE INDEX

In this section we first introduce the basic multi-index approach and its generalizations [14, 22, 20, 26, 29] and discuss its most important inefficiencies (Subsection 3.1).

As we will see, a query in a multi-index is solved in two phases: *searching* and *checking* phases. In the former phase, we are given substrings of the query and the goal is to find all the keys of the dictionary that match these substrings (possibly, up to one Hamming error). These keys are the candidates to be processed by the next phase which checks all of them to filter out the ones with Hamming distance larger than k with the query.

In this paper we improve the time efficiency of both phases and the overall space usage of any multi-index solution. However, in this section we only describe our improvements to the checking phase (Subsection 3.2). We defer the description of our other contributions to the experimental section (Subsection 4.2) because they are motivated, driven and supported by experimental evidences.

3.1 Multi-index approach

The multi-index approach has been introduced by Greene *e*t al. [14] and has been popularized by Manku *e*t al. [22] who use this index for the near-duplicate detection in a collection of Web pages showing that this approach scales with the size of the dictionary. In the following we will use multi-index to refer this approach.

The multi-index is based on a simple property: if two keys have a small Hamming distance, they must share at least a (sufficiently long) aligned substring. More precisely, consider the splitting of any two keys K and Q into $b = k + 1$ blocks of, possibly, variable length. We use $K[i]$ to denote the ith block of K in this splitting. The property is stated as follows.

PROPERTY 1. *Let $H(K, Q)$ be the Hamming distance between keys K and Q. If $H(K, Q) \le k$ and $b = k + 1$, then there must exist at least one index i, $1 \le i \le b$, such that $K[i] = Q[i]$.*

This property can be exploited by splitting each key of the dictionary D into $b = k+1$ blocks and by creating a separate

	multi-index				multi-index*	
	$b = k+1$		$b = k+2$		$b = \lfloor k/2 \rfloor + 1$	
k	idxs.	avg. cand.	idxs.	avg. cand.	idxs.	avg. cand.
2	3	3	3	> 1	2	66
3	4	262 144	10	845	2	66
4	5	3 011 244	15	24 382	3	108 908
5	6	15 853 476	21	282 248	3	108 908
6	7	53 184 325	28	1 835 008	4	4 456 448
7	8	134 217 728	36	8 089 969	4	4 456 448
8	9	279 604 801	45	27 101 194	5	41 555 165

Table 1: Number of indexes and average number of candidates for different number of erros k and fixed number of keys $d = 2^{32}$ of length $m = 64$ bits each.

index for each block; this is the reason for the name of this solution. Index I_i, for $1 \leq i \leq b$, is responsible for indexing the content of the ith block of each key. Since each of the b indexes duplicates the content of D, the overall space usage of this approach is $\Theta(b \cdot m \cdot d)$ bits.

A k-query for Q is solved in two phases: *searching* and *checking*. In the searching phase Q is first split into b blocks and, then, each of them is searched in the corresponding index to identify the subset of keys having the same block configuration. The union of these b subsets forms a set of candidates which is guaranteed by Property 1 to contain all the keys with Hamming distance at most k from Q. In the checking phase, we scan the set of candidates to identify the keys to be reported. The check of a candidate key K can be done in constant time whenever the key fits in a constant number of memory words. Indeed, the Hamming distance between K and Q equals to $H(K,Q) = \texttt{popcount}(K \oplus Q)$, where $\oplus$ denotes the bitwise XOR and `popcount` counts the number of bits set to one in a memory word; this instruction belongs to the instruction sets of any modern CPU.

Clearly, the overall efficiency of this approach is given by the efficiency of its two phases. As far as the searching phase is concerned, we point out that a possible solution consists in storing keys in the index I_i sorted by their ith block, so that we can use binary search to identify the range of keys which match the query block configuration. As we will see in the experiments, even if this is the most immediate approach, it is far from being the most efficient one for both time efficiency and space usage.

Storing consecutively the keys with the same ith block in the index I_i is convenient also for the cache efficiency of the second phase. Indeed this phase checks the candidates reported by the index I_i by scanning a consecutive range of keys, thus, obtaining the largest benefits from the CPU's prefetching. However, the multi-index cannot limit in any way the number of candidates to be checked so that a potentially very large fraction of the time may be wasted in filtering out false positive candidates. Observe that the number of candidates to be checked strictly depends on b which in turns is fixed by k. Indeed, we cannot have fewer than $d/2^{m/b} = d/2^{m/(k+1)}$ on average per index. Since this quantity grows rapidly with k, the query time becomes large even with relatively small values of k. For a concrete example, let us assume that D contains $d = 2^{32}$ keys of $m = 64$ bits each. Thus, each index gives $2^{16} = 65\,536$ candidates in average with $k = 3$ but this quantity becomes $2^{24} = 16\,777\,216$ with $k = 7$. The space usage is $O(4 \cdot 64 \cdot 2^{32})$ bits with $k = 3$ and $O(8 \cdot 64 \cdot 2^{32})$ bits with $k = 7$. In Table 1 we report the average number of candidates and number of indexes for different values of k.

The number of candidates can be reduced by generalizing the previous solution to let the number of blocks b be larger than $k+1$. In this generalization we build a index for every possible subset of $b-k$ block ids and each index is responsible for indexing the content of these $b-k$ blocks. Thus, we have $\binom{b}{b-k} = \binom{b}{k}$ indexes and the space usage is $O(d \cdot \binom{b}{k} \cdot m)$ bits. In the searching phase, we query each of these indexes to identify the subsets of keys having a certain configuration of each possible subset of $b-k$ blocks. As before, the checking phase processes these candidates and keeps only the query results. It is easy to see that, if the b blocks have the same length m/b, the average number of candidates identified by each index is $d/2^{(m/b)(b-k)}$. Thus, the number of candidates rapidly decreases as b increases. However, b cannot become too large because the number of indexes grows by a factor $\frac{b}{b-k}$ every time we increase b by one. In practice, the space usage of this solution on large datasets becomes prohibitive even with $b = k+2$ and, thus, this generalization may be useful only for very small datasets. We report in Table 1 the average number of candidates and number of indexes with this generalization with $b = k+2$.

A strategy used by recent papers [20, 26, 29] is able to improve both query time and space usage of multi-index by exploiting a well-known generalization of Property 1.

Property 2. *If $H(K,Q) \leq k$ and $b = \lfloor k/2 \rfloor + 1$, then there must exist at least one index i, $1 \leq i \leq b$, such that $H(K[i], Q[i]) \leq 1$.*

Essentially, this property says that, if we match blocks admitting up to 1 error, then we can reduce the number of blocks b almost by a half. This has the effect of increasing the length of the blocks and, thus, of reducing the number of candidates. We point out that Property 2 can be further generalized so that if we admit up to r errors in matching the blocks, the number of blocks can be reduced to $\lfloor k/(r+1) \rfloor + 1$.

The main idea for exploiting Property 2 is to change the searching phase of the multi-index approach. As before Q is split into b, now with $b = \lfloor k/2 \rfloor + 1$, blocks. Given the ith block in Q, the searching phase queries index I_i to identify all the keys whose ith block is at Hamming distance at most 1 with the block of Q. This can be done, for example, by using the previous binary search-based approach as follows. As before, we first search for Q's block; this gives the subset of keys which have the same block as Q. Then, if the length of the block is ℓ, we perform ℓ different searches. In each search we flip the ℓth bit of the block of Q and search for an exact match of this modified block. These ℓ searches give ℓ (possibly empty) subsets which contain all the keys whose ith block is at Hamming distance exactly 1 with the block of Q. Property 2 guarantees that the union of all the sets of candidates obtained by searching all the b indexes contains *all* the keys with Hamming distance at most k with Q. The idea is at the core of the solution in [20] and other similar approaches [26, 29]. We point out that an index using this strategy is called HEngine in [20]. However, we prefer to use multi-index* to refer to this solution to stress the fact that it is a multi-index in which the searching phase matches blocks up to 1 error.

Observe that the searching phase on multi-index* is more expensive than the one on multi-index. Indeed, we have to perform $m + b = m + \lfloor k/2 \rfloor + 1$ exact matches instead of $k + 1$ exact matches. Apart from this inefficiency, multi-index* has important benefits: 1) it has a better space usage because the number of indexes is reduced from $k + 1$ to $\lfloor k/2 \rfloor + 1$; 2) it reduces the average number of candidates to check which decreases from $d/2^{m/(k+1)}$ to $(m+b) \cdot d/(2^{m/b}) = (m + \lfloor k/2 \rfloor + 1) \cdot d/(2^{m/(\lfloor k/2 \rfloor + 1)})$. Table 1 reports these numbers for the case of a dictionary of size $d = 2^{32}$ and $m = 64$ bits.

3.2 Triangle inequality-based Multi-index

Experiments confirm that multi-index* improves significantly over multi-index on real datasets and scales with the number of errors k. Still, a large fraction of the checking phase is spent in filtering out false positive candidates. As we can see in Table 1, the average number of candidates is only 66 with $k = 3$ but this number increases rapidly to 4 456 448 with $k = 6$.

Our solution can be used indistinctly in combination with either multi-index or multi-index* and its goal is to significantly reduce the number of candidates to check. At a high level the idea consists in further subdividing the set of keys with the same block configuration into *clusters* and to exploit *triangle inequality* to avoid the check of keys of some of these clusters. The triangle inequality is stated as follows.

Property 3 (Triangle inequality). *For any three keys P, K, and Q, it holds $H(Q,P) \leq H(K,Q) + H(K,P)$.*

For each cluster, we select and store a pivot P and the maximum distance δ of any key K in the cluster with P. At query time we compute the distance $H(Q,P)$ so that we can lower bound $H(K,Q)$ for any K in the cluster. Indeed, we have $H(Q,P) - \delta \leq H(K,Q)$ by triangle inequality. If the lower bound is larger than k, we can safely discard all the keys in the cluster. If it is not the case the keys in the cluster have to be explicitly checked for the presence of any query result. This general idea can lead to several different solutions which differentiate themselves with respect to the strategies used to build, layout and index the clusters. In the following we present two of them. The first solution, called Clustered multi-index, has the merit of being conceptually simple and giving an interesting speed up for sufficiently small values of k or on smaller datasets. The second one, called BoundedClustered multi-index, is more sophisticated but is able to significantly reduce the number of candidates to check. We point out that the use of pivots and triangle inequality is at the basis of the so-called Metric Indexes (see [17] and references therein). However, our objectives and, thus, the indexing strategies differ from the ones of Metric indexes. Indeed, Metric indexes deal with highly complex and large objects (e.g., video clips) and, thus, their main goal is to minimize the number of distance computations and the number of I/O to answer a query. For this reason, most of the Metric indexes use a tree structure (e.g., B-tree like) to represent the dataset in which a different set of pivots is (possibly) selected at each node of the tree.

Clustered multi-index. For any existing block configuration in index I_i, we consider the subset S of all the keys with that block configuration. The idea in this strategy is to select any pivot P and to partition S into at most $m - \ell + 1$ clusters according to the Hamming distance of the keys with P, where ℓ is the length of the block. Thus, the cluster C_j contains all the keys in S which have Hamming distance exactly j with P. At query time, we only need to check those keys that belong to at most $2k+1$ of these clusters. Indeed, let Q be the query and K be any entry, if $H(Q,K) \leq k$, then $|H(Q,P) - H(K,P)| \leq k$ by triangle inequality. This means that we only need to check clusters whose keys have a Hamming distance in the range $[H(Q,P) - k, H(Q,P) + k]$ (namely, clusters C_j with $j \in [H(Q,P) - k, H(Q,P) + k]$). Observe that we could choose a different pivot for each subset S. However, for simplicity, our implementation fixes $P = 0$ for every subset, and, thus, $H(K,P)$ is just the Hamming weight of K, i.e., the number of bits set to 1 in K.

The layout of a index I_i is as follows. We first group the keys into subsets accordingly to the content of their ith block as done by any multi-index, then we partition the keys into clusters as explained above. For each block configuration, we write its clusters sorted by their distance with P and we store pointers to the beginning of each cluster. At query time, the group of keys with the appropriate block configuration is identified in the searching phase as in any other multi-index. In the checking phase we know after computing $H(Q,P)$ that it is sufficient to check clusters with Hamming weight in $[H(Q,P) - k, H(Q,P) + k]$. An interesting feature of the layout above is that these clusters are stored one after the other. This means that the checking phase has to perform just one scan that starts at the first cluster and ends at the last one without any random access in between. However, an important drawback of this strategy is that the sizes of the obtained clusters are far from being uniformly distributed. For example, we expect that a large fraction of the keys in S belong to the cluster $C_{\frac{m}{2}}$, since it contains all the keys in S with exactly $\frac{m}{2}$ bits set to 1.

BoundedClustered multi-index. In this solution we use a two level indexing approach and we allow the selection of several pivots for each block configuration. The first level of the index stores those pivots and some additional information, including pointers to the clusters obtained with those pivots which are stored in the second level. Consider again the subset S of all the keys with a certain block configuration in index I_i. The keys in S are subdivided into clusters by using several pivots. Both pivots and clusters are chosen according to the following greedy algorithm.

The greedy algorithm works in steps. At each step it chooses a pivot and selects a subset of keys from S which form the cluster. Those keys are removed from S and the greedy algorithm continues with another step as soon as S is not empty. Consider the ith of these steps. The pivot P_i of this step is chosen at random among the most distant keys from the pivot of the preceding step which are left in S. Once the pivot P_i is selected, we can consider the subset S_j, for any $j \in [0, 64 - \ell]$, which contains all the keys in S with Hamming distance *at most* j with P_i. Since ℓ is the length of the block and all the keys in S agree on this block, the union of these subsets covers S. Moreover, S_j is contained in S_{j+1} and $S_{64-\ell} = S$. The cluster C_i at the ith step equals the subset S_{E_i} where $0 \leq E_i \leq 64 - \ell$ is the smallest index such that $|S_{E_i}| \geq \tau$ or $E_i = 64 - \ell$ whenever such a subset does not exist. Note that the latter case may happen only at the last step of the greedy algorithm. Thus, the parameter τ fixes the minimum size of all the clusters but the last one.

The index I_i is a two level index. The second level stores, the clusters of each possible block configuration while the first level stores the pivots P_i, the errors E_i and a pointer the first key of corresponding cluster in the second level.

At query time, we process the pivots P_i in their selection order. If $H(P_i, Q) \geq E_i + k + 1$, we are sure that the Hamming distance of any key in C_i to Q is larger than k and, thus, we can safely skip the whole cluster. Indeed, consider a key K in C_i. By triangle inequality, we know that $H(K, Q) \geq H(P_i, Q) - H(P_i, K)$ and, by construction, that $H(P_i, K) \leq E_i$. This means that $H(K, Q) \geq E_i + k + 1 - E_i = k + 1$ and, thus, K cannot be a query result. Instead, if $H(P_i, Q) < E_i + k + 1$, the cluster C_i may contain query results and, thus, its keys have to be explicitly checked. However, if the distance with the pivot is small enough, we can *early exit* from the checking phase. Indeed, if $H(P_i, Q) \leq E_i - k$, we are guaranteed that C_i is the last cluster to be checked because none of the subsequent ones can contain any query result. Indeed, by construction, any key K in the subsequent clusters is such that $K(P_i, K) \geq E_i + 1$ otherwise K would belong to the cluster C_i. Thus, by triangle inequality we have $H(K, Q) \geq H(P_i, K) - H(P_i, Q)$ and, since $K(P_i, K) \geq E_i + 1$ and $H(P_i, Q) \leq E_i - k$, we have $H(K, Q) \geq E_i + 1 - (E_i - k) = k + 1$.

4. EXPERIMENTS

We evaluate our indexes on four different datasets of 64-bit keys from two common application areas: similarity search in web collections and image databases. The first two datasets were generated from the ClueWeb 2009 TREC Category A test collection which consists of 503.9 million English web pages crawled in 2009. The body text of each document in the collection was extracted using Apache Tika[1], and the words lowercased and stemmed using the Porter2 stemmer. We used two different hash functions on the resulting documents and removed duplicated hashes to obtain the following datasets.

- CW09_SimHash consists of 450 806 115 unique keys (i.e., $\approx$ 3.7 GBytes) obtained by using SimHash [7], a Locality Preserving Hashing (LSH);
- CW09_OddSketch consists of 471 510 174 unique keys (i.e., $\approx$ 3.7 GBytes) obtained by using OddSketches as described in [24].

We obtained the second two datasets from the INRIA BIGANN dataset which consists of 1 billion of 128-dimensional SIFT descriptors [21] extracted from approximately 1 million images [18][2]. This dataset has been released in 2011 to evaluate high dimensional indexing algorithms on a realistic scale. The following two datasets are obtained in [26] by applying two different similarity-preserving hash functions.

- BA_Sift_Lsh consists of 949 923 339 unique keys (i.e., $\approx$ 7.5 GBytes) obtained by using SimHash [7]. We use the suffix Lsh to refer this dataset to be consistent with the naming in [26].
- BA_Sift_Mlh consists of 966 592 208 unique keys (i.e., $\approx$ 7.5 GBytes) obtained by using Minimal Loss Hashing (MLH) [25].

[1]http://tika.apache.org/

[2]http://corpus-texmex.irisa.fr/

For each dataset we generated a query set consisting of 200 000 keys which were selected as follows. The first 100 000 keys were randomly selected and removed from the original dataset while the second 100 000 keys are a subset of the remaining dataset. We performed experiments also with these two subsets of queries separately without observing any significant difference on the final results.

Testing details. All the algorithms were implemented. in C++14 and compiled with GCC 5.3 with the highest optimization settings. Our source code – which is a flexible C++ template library – and the benchmark suite, which facilitates the automatic execution of all following experiments are avaiable at https://github.com/xxsds/multi_index. The tests reported in the paper were performed on a machine with 6 Intel Xeon E5-2630 Haswell cores (12 threads) clocked at 2.80Ghz, with 64GiB RAM, running Linux 3.13.0. In addition we have run the benchmarks suite on a second machine equipped with 32 Intel Xeon E5-4640 cores (64 threads) clocked at 2.40Ghz, with 512GiB RAM, running Linux 3.13.0 to confirm the results. Note that all experiments were run in-memory.

4.1 Comparing multi-index and multi-index*

We first provide experiments similar to the ones presented in [20] to compare the time and space efficiency of multi-index and multi-index*. Even if the implementations of both index types are comparable with the one tested in [20], we obtain slightly different results. The different outcome is motivated by the fact that we run the experiments on much larger datasets: the largest dataset used in [20] contained 0.5 million keys while we scale the experimental set by three orders of magnitudes. However, our experiments further confirm the thesis in [20]: multi-index* is more convenient than multi-index.

In Figure 1 we report the average query time and space usage of multi-index and multi-index* for k ranging from 2 to 5. Due to space limitation, we report the results only for CW09_SimHash and BA_Sift_Lsh but we observed a similar behaviour also on the other datasets. Both multi-index and multi-index* use binary search in the searching phase.

As expected from our discussion in Subsection 3.1 (see Table 1), multi-index* is much more space efficient than multi-index: it improves space usage by either a factor 1.5 (for even k) or 2 (for odd k). Indeed, multi-index* uses $b = \lfloor k/2 \rfloor + 1$ indexes instead of $b = k + 1$. As far as query time is concerned, we observe that multi-index* is faster (up to a order of magnitude) than multi-index for all values of k but $k = 2$. Observe that the gap rapidly increases with the increasing of k. This is because, even if the searching phase is more demanding in multi-index*, the much smaller number of candidates to check suffices to let multi-index* outperform multi-index also regarding query time.

For $k = 2$, multi-index is faster than multi-index*. Since in this case the number of candidates is small, the more expensive searching phase in multi-index* determines the outcome. However, in the next subsection we will significantly improve the efficiency of the searching phase. This would suffice to significantly reduce, or even avoid, this effect.

The space and time inefficiency of multi-index leads us to focus our attention only to multi-index* in the remaining of the paper. Nevertheless, all the improvements we will introduce for multi-index* are also applicable to multi-index.

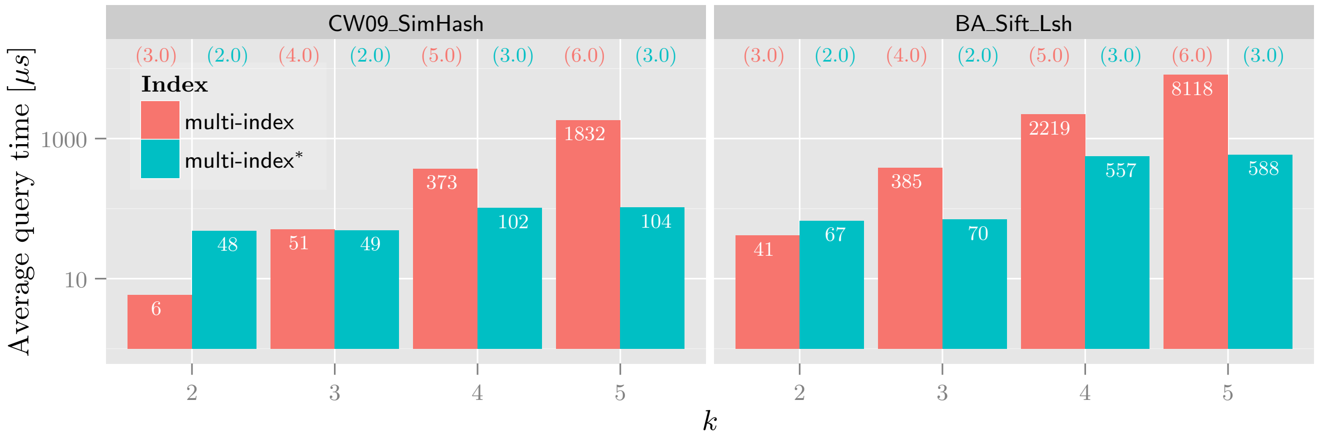

Figure 1: Average query time for multi-index and multi-index* on all data sets. The numbers in brackets denote the space usage as the increasing factor with respect to the size of the original dataset.

4.2 Improving any multi-index-based solution

In this subsection we provide an extensive experimental evaluation to measure the efficiency of both phases of the query resolution in any multi-index based solution. We first measure the efficiency of binary search in the searching phase and introduce the use of *succinct* data structures as a much better alternative for this task. This improves the efficiency of this phase up to one order of magnitude. As far as checking phase is concerned, we will first use a benchmark to estimate the time required to scan and compute Hamming distance of runs of variable-length keys. This drives the design of a novel layout for any multi-index based solution which improves the overall space usage by a factor ranging from 18% to 40% depending on the value of k, still preserving or even improving the time efficiency.

On the efficiency of the searching phase. A index I_i in the multi-index stores the keys of the dataset sorted by the content of their ith block. This way, given a certain block configuration B, the keys that have the block configuration B span a consecutive range on the sorted set of keys. The goal of the searching phase is to identify this range.

Even if the use of binary search is the simplest solution for this task, the next experiments show that it is far from being the most efficient one. A more efficient solution, that we will refer as Succinct, uses a binary vector V which is obtained as follows. We scan the sorted dataset and, for any possible block configuration, we write in unary the number of keys which have that block configuration. The binary vector V has $u = 2^\ell + n$ bits, where ℓ is the length of the block. Indeed, we have a bit set 1 for each possible block configuration (i.e., 2^ℓ bits set to 1 overall) and a bit set to 0 for each key in the dataset (i.e., n bits set to one overall). We build a data structure to answer $\mathsf{Select}_1(x)$ queries on V. A query $\mathsf{Select}_1(x)$, for any $1 \le x \le n$, returns the position of the xth bit set to 1 in V. There exists a data structure that answers any Select_1 query in $O(1)$ time by using $o(u)$ bits of additional space a part from V [9].

Note that two Select_1 queries suffice for identifying the range of keys which have a certain block configuration B. Indeed, $l = \mathsf{Select}_1(B) - B$ and $r = \mathsf{Select}_1(B+1) - B$ are the positions of the first and the last key, respectively, which have the block configuration B. Note that l equals r iff no such key exists. The space usage of this data structure is $u + o(u) = 2^\ell + n + o(2^\ell + n) < 2(2^\ell + n)$ bits and, thus, the space cost per entry of this data structure is relevant only if n is small compared to 2^ℓ. For all our datasets, n is even larger than 2^ℓ for any $k > 3$ since $\ell \le 16$, and, thus, the space usage is less than 2 bits per key. Instead, for $k \le 3$, we have $\ell = 32$ and the space usage of the data structure is less than $2 \cdot 2^{32}/n + 2$ bits per key. This data structure is not the best possible choice for any range of values of n and 2^ℓ. Indeed, if 2^ℓ is much smaller than n (i.e., $2^\ell = O(\frac{n}{\log n})$), it is better to directly tabulate the answers of all the possibile block configuration since it requires $O(2^\ell \log n)$ bits. Instead, if n is small compared to 2^ℓ, it is better to use a solution based on the Elias-Fano representation [11, 12], whose description is omitted due to space limitation. This solution uses $n \log \frac{2^\ell + n}{n} + 2n + o(n)$ bits of space and has a slightly larger query time. There exist several efficient implementation of these representations [15, 13] which provide essentially the same performance for these queries. We opted for the SDSL [13] library.

Figure 2 reports the searching time with both binary search and Succinct and the checking time for multi-index* with $k = 3$ and $k = 4$ on CW09_SimHash and BA_Sift_Lsh. Succinct is always more efficient than binary search with a factor of improvement that ranges from 3 (CW09_SimHash with $k = 3$) to 14 (BA_Sift_Lsh with $k = 4$). Apparently, the Succinct solution is space inefficient compared to binary search since the latter does not need any auxiliar information apart from the sorted dataset. However, as we will show in the next paragraph, the information stored implicitly in the data structure can be exploited to significantly reduce the overall space usage of the index.

The comparison between searching and checking time confirm our expectation: the searching phase dominates the overall query time when $k \le 3$, while it is the checking phase to become dominant when $k > 3$. Indeed, on one hand, since multi-index* performs $64 + b$ searches the cost of the searching phase only slightly increases as k increases. On the other hand, the overall expected number of candidates to check is only a small constant with $k \le 3$ but it increases rapidly with the increasing of k (see again Ta-

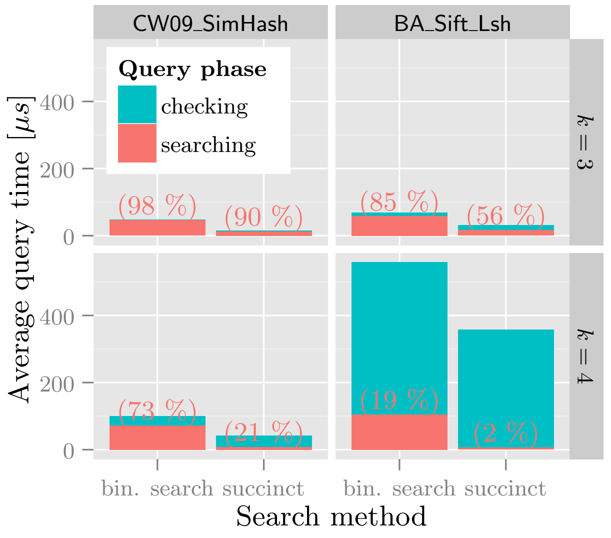

Figure 2: Searching time (resp. checking time) ratio of total time for multi-index* with $k = 3$ and $k = 4$ on CW09_SimHash and BA_Sift_Lsh

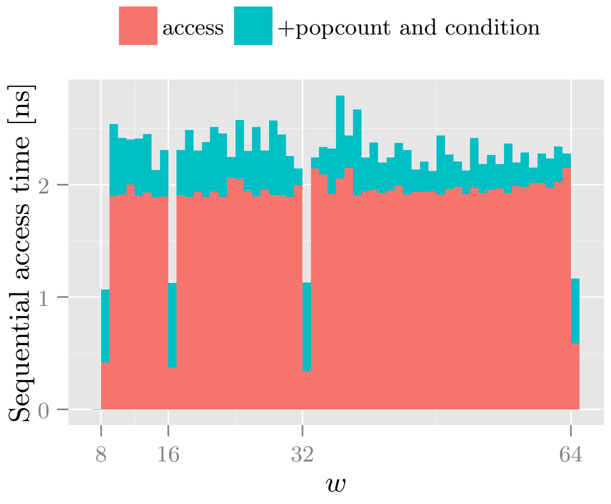

Figure 3: Average sequential processing time of a key of a vector of w-bit integers. The red part is only access, the blue part is the additional time to perform one popcount operation and branching into a condition.

ble 1). The searching phase with Succinct takes 90% on CW09_SimHash and 56% on BA_Sift_Lsh of the overall query time with $k = 3$. These percentages decrease to 21% and 2%, respectively, with $k = 4$ and further and more rapidly decrease as k increases so that the searching time becomes negligible compared to the checking time.

On the efficiency of the checking phase. Once we performed the searching phase on a index I_i, we have identified the range $R = \langle l, r \rangle$ of all the keys which have a certain block configuration B of length ℓ. The checking phase starts scanning the keys in this range and computes their Hamming distance with the query. There is a simple but crucial observation which allows us to save space in representing the keys. Indeed, we can easily note that the Succinct solution, differently from the binary search, does not need to access the ith block of the keys to identify the range R. Thus, we could store each key without its ith block by using $64 - \ell$ bits instead of 64 bits. However, since in general $64 - \ell$ is not a power of two, this would ask for storing a vector of unaligned entries which would significantly reduce the efficiency of the subsequent scanning. To measure the effect of these unaligned accesses we run a benchmark which scans a chunk of 2^{14} consecutive keys in a vector with 2^{30} keys of lengths from 8 to 64 bits each. For each processed key we also perform one `popcount` operation and a branch into a condition, thus, simulating the whole checking phase. Results (see Figure 3) show that on average accessing aligned entries is a factor 4.2 more efficient than unaligned entries. This factor reduces to 3.0 when considering the cost of the whole process because the cost of the subsequent operations does not depend on the block length.

These results discourage the use of the previous approach and motivate the design of a different representation, which, instead of using only a vector of unaligned entries, uses two different vectors: one with aligned entries and one with, possibly, unaligned ones. Consider the remaining $64 - \ell$ bits of each key. These bits are partitioned in two parts: the lower 32 bits and the remaining $64 - \ell - 32$ upper bits. We represent the keys by using two vectors L and H: the vector L stores the lower bits of the keys while H stores the upper bits. Apparently this representation does not solve the issue of the previous naïve approach: reconstructing a key still requires an access to a vector with unaligned entries. However, we can limit the overall number of accesses to unaligned entries as follows. We first compute the Hamming distance between the lower bits of the key and the query. If this Hamming distance is larger than k, we do not need to access the upper bits of the key, saving one unaligned access. Otherwise, we reconstruct the whole key and compute its Hamming distance from the query. Since 32 bits is a sufficiently large part of the key, we expect that the first check suffices for a large portion of the keys. We use SPLIT to refer to this approach.

In Figure 4 we compare SPLIT with the aligned approach and the unaligned one on BA_Sift_Lsh with $k = \{5, 7, 9\}$. We first observe that SPLIT is almost as fast as aligned and two times faster than unaligned for $k = 7$. The reason is that, for this value of k, we have $b = 4$, and, thus, entries in both H and L are aligned. For the remaining cases, SPLIT and unaligned are comparable, with the latter which is faster for larger values of k. The reason for this is that if a key has Hamming distance larger than k, the first check has to filter it out by checking only its first 32 bits. However, as k increases, there are less chances that these bits have more than k errors with the corresponding bits of the query. As the number of keys that pass this filter increases, SPLIT loses wrt unaligned because it has to access entries of two different vectors and any access at H is expensive because it is random and possibly unaligned. Experiments show that only the 1.87% of keys pass the first check for $k = 5$ but this percentage increases to 4.26% for $k = 7$, and to 8.99% for $k = 9$. To overcome this limitation we use the following trick that we refer as XOR. Given a key K, let a_K be the

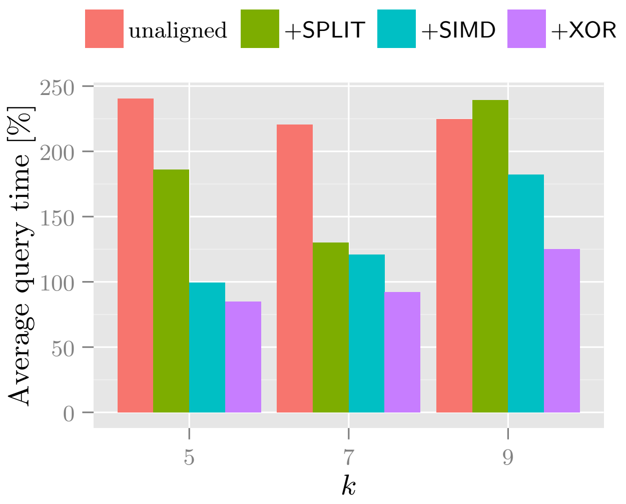

Figure 4: Average query time of the unaligned, SPLIT, SPLIT+ SIMD, and SPLIT+ SIMD+ XOR indexes as fraction of the average query time of the aligned index on BA_Sift_Lsh with $k = \{5, 7, 9\}$.

lower 32 bits of K and b_K be its $64 - \ell - 32$ upper bits. We store $x_K = a_K \oplus b_K$ in the vector L instead of a_K. Note that x_K sketches the content of the key K in 32 bits. Clearly, given x_K and b_K, we can compute a_K. The first check now computes the Hamming distance of x_K and x_Q, where $x_Q = a_Q \oplus b_Q$ with a_Q and b_Q denoting the lower and upper bits of the query Q. This is a better estimate which allows us to reduce the number of keys that pass the first check. Experiments show that this percentage is 0.66% for $k = 5$, 1.27% for $k = 7$, and 3.02% for $k = 9$.

The use of 32-bit entries in the vector H allows a further optimization which speeds up the computation of the Hamming distance needed at the first check. Indeed, it is possible to use SIMD operations to perform 4 Hamming distance computations at once on 32-bit entries. Due to space limitations we do not enter into details of this technique, we just point out that it is not applicable to 64-bit entries. We use SIMD to refer to this optimization.

As shown in Figure 4, the combined use of SPLIT, SIMD and XOR improves the speed of the aligned representation for k up to 7 and almost cancels the gap for larger values of k. Thus, our layout uses a fraction between 0.67 (with $k = 5$) and 0.8 (with $k = 9$) of the space of the aligned representation without any time penalty.

4.3 Overall comparison

In our comparison we compare the following solutions.

- Original is an implementation of multi-index* which uses binary search in the searching phase and keys are stored in a vector with aligned 64-bit entries. We validated this implementation with the source code of the implementations in [20] and [26]. However, we were not able to compare directly with these implementations because 1) the implementation in [20] does not scale with the dataset size and, indeed, [20] tested it on a dataset with less than 0.5 million keys (i.e., three orders of magnitudes smaller than ours); 2) implementation in [26] actually solves nearest neighbor queries which is a slightly different problem. However, as far as the implementation in [26] is concerned, we point out that i) their function to compute Hamming distance is slower than ours[3] and, since these computations are the dominant cost in the checking phase, our implementation is faster; ii) our implementation is also more space efficient as, for example, their index with $b = 2$ blocks on BA_Sift_Lsh and BA_Sift_Mlh uses 28 Gbytes of space while ours uses 15 Gbytes.

- Compact is an implementation of multi-index* which uses all the time and space improvements described in Subsection 4.2, namely Succinct for the searching phase and SPLIT+ SIMD+ XOR for the checking phase.

- Clustered is an implementation which uses all the improvements of the previous implementation combined with the Clustered strategy described in Subsection 3.2.

- BoundedClustered is an implementation which uses all the improvements in Compact combined with the BoundedClustered strategy described in Subsection 3.2. The threshold τ is set to 32 for $k = \{4, 5\}$, to 64 for $k = \{6, 7\}$ and to 128 for $k = \{8, 9\}$. These thresholds have been chosen after an experimental evaluation whose results are not reported due to space limitations.

	Original		Compact	
	$[\mu s]$	$[\frac{\|I\|}{d}]$	$[\mu s]$	$[\frac{\|I\|}{d}]$
$k = 2$				
CW09_SimHash	48	2.0	17(2.9×)	1.4(-30%)
CW09_OddSketch	51	2.0	17(3.0×)	1.4(-31%)
BA_Sift_Lsh	67	2.0	25(2.7×)	1.2(-40%)
BA_Sift_Mlh	62	2.0	22(2.8×)	1.2(-40%)
$k = 3$				
CW09_SimHash	49	2.0	17(2.9×)	1.4(-30%)
CW09_OddSketch	49	2.0	16(3.0×)	1.4(-31%)
BA_Sift_Lsh	70	2.0	28(2.5×)	1.2(-40%)
BA_Sift_Mlh	64	2.0	25(2.5×)	1.2(-40%)

Table 2: Average query time and space usage of Original and Compact on all the datasets for $k = 2$ and $k = 3$. The space usage is reported as the increasing factor with respect to the size of the original dataset. We also report in brackets the improvement of Compact wrt Original.

In our overall comparison we distinguish two cases based on the value of k.

The results for small values of k, namely $k = 2$ and $k = 3$, are reported in Table 2. Due to space limitations, we only report results for Original and Compact. Anyway, the higher level of sophistication of Clustered and BoundedClustered and their small improvements on the query time do not justify their use on these values of k. Indeed, their main objective is that of improving the checking phase but, as we discussed in Subsection 4.2, the time of the searching phase dominates the overall query time with these values of k.

[3] A switch statement dependent on the word length prohibits compile optimizations.

		Original		Compact		Clustered		BoundedClustered	
		$[\mu s]$	$[\lvert I\rvert/d]$	$[\mu s]$	$[\lvert I\rvert/d]$	$[\mu s]$	$[\lvert I\rvert/d]$	$[\mu s]$	$[\lvert I\rvert/d]$
CW09_SimHash	$k=4$	102	3.0	39(2.6×)	2.1(-32%)	37(2.7×)	2.1(-31%)	41(2.5×)	2.3(-25%)
CW09_OddSketch		94	3.0	34(2.8×)	2.1(-32%)	33(2.8×)	2.1(-31%)	37(2.6×)	2.3(-25%)
BA_Sift_Lsh		557	3.0	271(2.1×)	2.0(-32%)	220(2.5×)	2.1(-31%)	163(3.4×)	2.2(-26%)
BA_Sift_Mlh		368	3.0	149(2.5×)	2.0(-32%)	126(2.9×)	2.1(-31%)	112(3.3×)	2.2(-26%)
CW09_SimHash	$k=5$	104	3.0	42(2.4×)	2.1(-32%)	43(2.4×)	2.1(-31%)	47(2.2×)	2.3(-25%)
CW09_OddSketch		94	3.0	34(2.7×)	2.1(-32%)	35(2.7×)	2.1(-31%)	40(2.3×)	2.3(-25%)
BA_Sift_Lsh		588	3.0	329(1.8×)	2.0(-32%)	307(1.9×)	2.1(-31%)	266(2.2×)	2.2(-26%)
BA_Sift_Mlh		387	3.0	179(2.2×)	2.0(-32%)	171(2.3×)	2.1(-31%)	168(2.3×)	2.2(-26%)
CW09_SimHash	$k=6$	871	4.0	401(2.2×)	3.1(-23%)	327(2.7×)	3.1(-23%)	286(3.1×)	3.1(-22%)
CW09_OddSketch		791	4.0	348(2.3×)	3.1(-23%)	276(2.9×)	3.1(-23%)	254(3.1×)	3.1(-22%)
BA_Sift_Lsh		4 668	4.0	3 252(1.4×)	3.1(-23%)	2 878(1.6×)	3.1(-23%)	1 465(3.2×)	3.1(-22%)
BA_Sift_Mlh		3 165	4.0	1 975(1.6×)	3.1(-23%)	1 703(1.9×)	3.1(-23%)	899(3.5×)	3.1(-22%)
CW09_SimHash	$k=7$	875	4.0	429(2.0×)	3.1(-23%)	385(2.3×)	3.1(-23%)	358(2.4×)	3.1(-22%)
CW09_OddSketch		787	4.0	359(2.2×)	3.1(-23%)	312(2.5×)	3.1(-23%)	300(2.6×)	3.1(-22%)
BA_Sift_Lsh		4 945	4.0	4 036(1.2×)	3.1(-23%)	3 897(1.3×)	3.1(-23%)	2 488(2.0×)	3.1(-22%)
BA_Sift_Mlh		3 272	4.0	2 356(1.4×)	3.1(-23%)	2 228(1.5×)	3.1(-23%)	1 400(2.3×)	3.1(-22%)
CW09_SimHash	$k=8$	4 737	5.0	3 930(1.2×)	4.1(-18%)	3 448(1.4×)	4.1(-18%)	2 422(2.0×)	4.1(-18%)
CW09_OddSketch		4 501	5.0	3 588(1.3×)	4.1(-18%)	3 033(1.5×)	4.1(-18%)	2 107(2.1×)	4.1(-18%)
BA_Sift_Lsh		21 936	5.0	24 148(0.9×)	4.1(-18%)	23 548(0.9×)	4.1(-18%)	12 118(1.8×)	4.1(-19%)
BA_Sift_Mlh		15 652	5.0	15 818(1.0×)	4.1(-18%)	14 881(1.1×)	4.1(-18%)	6 809(2.3×)	4.1(-19%)
CW09_SimHash	$k=9$	4 778	5.0	4 804(1.0×)	4.1(-18%)	4 388(1.1×)	4.1(-18%)	3 218(1.5×)	4.1(-18%)
CW09_OddSketch		4 499	5.0	4 400(1.0×)	4.1(-18%)	3 871(1.2×)	4.1(-18%)	2 890(1.6×)	4.1(-18%)
BA_Sift_Lsh		24 249	5.0	31 679(0.8×)	4.1(-18%)	31 962(0.8×)	4.1(-18%)	20 929(1.2×)	4.1(-19%)
BA_Sift_Mlh		16 611	5.0	20 173(0.8×)	4.1(-18%)	19 873(0.8×)	4.1(-18%)	11 190(1.5×)	4.1(-19%)

Table 3: Average query time and space usage of all the solutions on all the datasets for $k = \{4, 5, 6, 7, 8, 9\}$. The space usage is reported as the increasing factor with respect to the size of the original dataset. We also report in brackets the improvement of our solutions wrt Original.

Results show that Compact improves the query time of Original by a factor that ranges between 2.5 and 3.0. This is mainly due to the replacement of binary search with our Succinct solution. The space usage is also improved by at least 30% and up to 40% with better results for larger datasets. This means that while the space usage of Original is two times the size of the original datasets, Compact uses slightly more space than the original datasets. Note that the space usage does not depend on k because with both values we have to use $b = 2$.

The results for larger values of k, namely k between 4 and 9, are reported in Table 3. We first observe that our solutions improve the space usage of Original by 17% up to 32% depending on the solution, the value of k and the dataset size. Note that Compact is always slightly more space efficient than Clustered and BoundedClustered. This is due to the auxiliar information stored by Clustered and BoundedClustered to describe and locate their clusters. However, the impact of this space overhead decreases with the size of the dataset and the value of k.

As far as the query time is concerned, we first observe that Compact and Clustered are faster (factor 1.4×—2.9×) than Original up to $k = 7$. For $k = \{8, 9\}$, these three solutions have a quite close time efficiency with one which is faster than the others depending on the dataset. We also observe that Clustered always outperforms Compact even if for some values of k the improvement is below the 10%. This is expected because Clustered always checks a subrange of the keys checked by Compact by using the same scanning procedure. Finally, BoundedClustered is the fastest solution with margin. Indeed, it outperforms Original by a factor that ranges between 2 and 3.4 for k up to 7. For $k = \{8, 9\}$, the factor of improvement is at least 1.5 for all datasets but BA_Sift_Lsh with $k = 9$ where it is 1.2.

k	Original or Compact	Clustered	BoundedClustered	
4	322 795	216 026(1.5×)	38 347	(8.4×)
5	322 795	253 511(1.3×)	60 151	(5.4×)
6	3 754 240	3 106 170(1.2×)	485 530	(7.7×)
7	3 754 240	3 346 000(1.1×)	701 792	(5.3×)
8	21 287 200	19 704 300(1.1×)	2 784 240	(7.6×)
9	21 287 200	20 356 400(1.0×)	3 903 500	(5.5×)

Table 4: Average number of checked candidates per query on BA_Sift_Lsh by varying the value of k. In brackets we report the reduction factor with respect to Original and Compact .

Finally, we report in Table 4 the average number of checked candidates per query on BA_Sift_Lsh. Clearly, Original and Compact always check the same set of candidates because they differ only on the representation of the keys. These numbers partially explain the time results in Table 3. It is apparent that BoundedClustered is more efficient also on this respect and, indeed, it reduces the number of checked candidates by a factor that ranges from 5.3 to 8,4. However, these high reduction factors are not reflected on equally high speed up factors. This is due to the more predictable memory access patterns of the other solutions which scan a whole range of consecutive candidates, while BoundedClustered skips subranges of the whole range. Thus, the much smaller number of checked candidates in BoundedClustered is partially balanced by the higher benefits that the other solutions obtain by the CPU prefetching.

5. FUTURE WORK

We conclude with two interesting open questions. First, we point out that the use of the Elias-Fano representation [11, 12] to store the sorted keys of an index requires $n \log \frac{2^{64}}{n} + 2n + o(n)$ bits. This space bound implies that each key requires between 36 and 38 bits on our data. This means that it always almost halves the space usage of Original and improves the ones of the other solutions regardless the value of k. However, scanning a portion of a vector in Elias-Fano representation is much more inefficient than with the others. Thus, a solution that uses Elias-Fano would only introduce a time/space tradeoff. The design of a representation which achieves the space bound of Elias-Fano still preserving the time efficiency of the other solutions is left as a future work.

For the second one, we observe that the greedy selection of the pivots in both Clustered and BoundedClustered may be suboptimal. It would be very interesting to design an efficient optimization algorithm which selects the pivots to minimize the expected execution time of the subsequent queries. One of the challenges here is to provide a precise model to estimate the query execution time.

6. ACKNOWLEDGEMENT

This work was partially supported by the project PRA_2016_64 *Through the fog* funded by the University of Pisa and by the EU H2020 Program under the scheme *INFRAIA-1-2014-2015: Research Infrastructures* grant agreement #654024 *SoBigData: Social Mining & Big Data Ecosystem*.

7. REFERENCES

[1] D. Belazzougui. Faster and space-optimal edit distance 1 dictionary. In *CPM*, pages 154–167, 2009.

[2] D. Belazzougui and R. Venturini. Compressed string dictionary search with edit distance one. *Algorithmica*, 2016 (to appear).

[3] L. Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1), 2011.

[4] G. S. Brodal and V. Srinivasan. Improved bounds for dictionary look-up with one error. *Information Processing Letters*, 75(1-2):57–59, 2000.

[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[6] H. Chan, T. W. Lam, W. Sung, S. Tam, and S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010.

[7] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[8] I. Chegrane and D. Belazzougui. Simple, compact and robust approximate string dictionary. *J. Discrete Algorithms*, 28:49–60, 2014.

[9] D. Clark. *Compact Pat Trees.* PhD thesis, Department of Computer Science, University of Waterloo, 1996.

[10] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[11] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[12] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA*, 1971.

[13] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, 2014.

[14] D. H. Greene, M. Parnas, and F. F. Yao. Multi-index hashing for information retrieval. In *FOCS*, pages 722–731, 1994.

[15] R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *SEA*, pages 5–17, 2013.

[16] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[17] M. L. Hetland. The basic principles of metric indexing. In *Swarm intelligence for multi-objective problems in data mining*, pages 199–232. Springer, 2009.

[18] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, pages 861–864, 2011.

[19] P. Li and A. C. König. Theory and applications of b-bit minwise hashing. *Commun. ACM*, 54(8):101–109, 2011.

[20] A. X. Liu, K. Shen, and E. Torng. Large scale hamming distance query processing. In *ICDE*, pages 553–564, 2011.

[21] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[22] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[23] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1969.

[24] M. Mitzenmacher, R. Pagh, and N. Pham. Efficient estimation for high similarities using odd sketches. In *WWW*, pages 109–118, 2014.

[25] M. Norouzi, D. J. Fleet, and R. Salakhutdinov. Hamming distance metric learning. In *NIPS*, pages 1070–1078, 2012.

[26] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(6):1107–1119, 2014.

[27] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *SIGMOD*, 2013.

[28] A. C.-C. Yao and F. F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25(1):194–202, 1997.

[29] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu. Hmsearch: an efficient hamming distance query processing algorithm. In *SSDBM*, pages 19:1–19:12, 2013.