GoldFinger: Fast & Approximate Jaccard for Efficient KNN Graph Constructions

Rachid Guerraoui, Anne-Marie Kermarrec, Guilhem Niot, Olivier Ruas, François Taïani

GoldFinger: Fast & Approximate Jaccard for Efficient KNN Graph Constructions

Rachid Guerraoui, *EPFL,* Anne-Marie Kermarrec, *EPFL,* Guilhem Niot, *ENS de Lyon,*
Olivier Ruas, *Pathway,* and François Taïani, *Univ Rennes, Inria, CNRS, IRISA*

Abstract—We propose *GoldFinger*, a new *compact* and *fast-to-compute* binary representation of datasets to approximate Jaccard's index. We illustrate the effectiveness of GoldFinger on the emblematic big data problem of K-Nearest-Neighbor (KNN) graph construction and show that GoldFinger can drastically accelerate a large range of existing KNN algorithms with little to no overhead. As a side effect, we also show that the compact representation of the data protects users' privacy *for free* by providing k-anonymity and l-diversity. Our extensive evaluation of the resulting approach on several realistic datasets shows that our approach reduces computation times by up to 78.9% compared to raw data while only incurring a negligible to moderate loss in terms of KNN quality. We also show that GoldFinger can be applied to KNN queries (a widely-used search technique) and delivers speedups of up to $\times 3.55$ over one of the most efficient approaches to this problem.

Index Terms—KNN graphs, fingerprint, similarity

1 Introduction[1]

K-Nearest-Neighbor (KNN) graphs[2] play a fundamental role in many big data applications, including search [8], [9], recommendation [11], [42], [44] and classification [51]. A KNN graph is a directed graph of entities (e.g., users, documents etc.), in which each entity (or *node*) is connected to its k most similar counterparts or *neighbors*, according to a given *similarity metric*. In many applications, this similarity metric is computed from a second set of entities (termed *items*) associated with each node in a bipartite graph (often extended with weights, such as ratings or frequencies). For instance, in a movie rating database, nodes are users, and each user is associated with the movies (items) she has rated [34].

Being able to compute a KNN graph efficiently is crucial in situations that are constrained, either in terms of time or resources. This is the case of *real time*[3] web applications, such as news recommenders, that must regularly recompute their suggestions in short intervals on fresh data to remain relevant. This is also the case of privacy-preserving personal assistants executing learning tasks on personal devices with limited resources [3].

Computing an *exact* KNN graph rapidly becomes intractable on large datasets. Fortunately, many applications only require a good *approximation* of the KNN graph [37], [40]. Recent KNN construction algorithms [11], [24] have therefore sought to reduce the number of similarity computations by exploiting a *greedy strategy*. These techniques, among the most efficient to date, seem, however, to have reached their limits.

In this paper, rather than reducing an algorithm's complexity (e.g. by computing fewer similarities), we explore an orthogonal strategy. Our take is motivated by the system bottlenecks induced by large data volumes: large data stress complex algorithms and choke the underlying computation pipelines these algorithms execute on [15].

More precisely, we propose to fingerprint the set of items associated with each node into what we have termed a *Single Hash Fingerprint* (SHF), a 64- to 8096-bit vector summarizing a node's profile (e.g. the movies the user have seen, the web pages she visited). SHFs are very quick to construct, provide a sufficient approximation of the similarity between two nodes using extremely cheap bit-wise operations. While the primary purpose of SHFs is efficient computation, it turns out that SHFs also protect the privacy of users by hiding the original clear-text information and providing valuable properties such as k-anonymity and l-diversity. We use these SHFs to rapidly construct KNN graphs, in an overall approach we have dubbed *GoldFinger*.

In this paper, we make the following contributions: (1) we introduce GoldFinger, a *generic* and *efficient* approach to accelerate *any* KNN graph algorithm relying on Jaccard's index, one of the most common metrics used to compute KNN graphs, which can be tuned to trade space and time for accuracy; (2) we propose a formal analysis of GoldFinger's estimation mechanism; (3) we formally analyze the privacy protection granted as a side effect by GoldFinger in terms of k-anonymity and ℓ-diversity; (4) we extensively evaluate our approach on a range of state-of-the-art KNN graph algorithms, such as Locality Sensitive Hashing (LSH), on six representative datasets, and we show that GoldFinger can reduce computation times by up to 78.9% against existing approaches, while only incurring a small loss in terms of quality; (5) finally, we show that GoldFinger can be further improved by using hashing functions adapted to the

1. This paper extends two earlier conference publications [32], [33]. It provides a more in-depth evaluation of the proposed approach, GoldFinger, and applies it to the popular KNN query problem.

2. This paper focuses primarily on the problem of computing a complete KNN graph, but we also explore the related but different problem of answering a sequence of KNN queries in Section 7.2.

3. *Real time* is meant in the sense of *web real-time*, i.e. the proactive push of information to on-line users.

dataset, and we observe that GoldFinger also performs well on the popular KNN query problem, delivering speedups of up to $\times 3.55$ against vanilla HNSW [47], one of the most efficient approaches to this problem.

In the following, we first discuss the related work (Sec. 2), before going into details about the context of our work and our approach (Sec. 3). We then present our evaluation procedure (Sec. 4) and our results (Sec. 5); we report on factors impacting our approach (Sec. 6), before investigating further improvements (Sec. 7), and concluding (Sec. 8).

2 Related work

For small datasets, KNNs can be solved efficiently using specialized data structures [10], [45]. These solutions, unfortunately, do not scale, as computing an exact KNN efficiently remains an open problem. Most practical approaches, therefore, compute an approximation of the KNN graph (ANN), as we do.

A first way to accelerate the computation time is to decrease the number of comparisons between users, taking the risk to miss some neighbors. *Recursive Lanczos Bisection* [18] computes an ANN graph using a divide-and-conquer method, while *NNDescent* [24] and *Hyrec* [11] rely on local search, i.e. they assume that a neighbor of a neighbor is likely to be a neighbor, and thus drastically decrease the scan rate (the proportion of similarities values effectively computed to construct the graph), delivering substantial speedups. *KIFF* [12] computes similarities only when users share an item. KIFF works particularly well on sparse datasets but has more difficulties with denser datasets such as the ones we studied. *Locality Sensitive Hashing* (LSH) [35] allows fast ANN graph computations by hashing users into buckets. The neighbors are selected only between the users of the same buckets. Several hash functions have been proposed, for different metrics [14], [17]. All of the above works can be combined with our approach—as we have demonstrated in the case of *NNDescent*, *Hyrec*, and *LSH*—and are thus complementary to our contribution.

Another strategy to accelerate a KNN graph's construction consists in compacting users' profiles, in order to obtain a fast approximation of the similarity metric. Keeping only a fraction of the profiles speeds-up Jaccard computation [38] but the resulting approach is not as fast as GoldFinger. *Minwise hashing* [7], [43] approximates Jaccard's index by only keeping a small subset of items for each user. It is space efficient but incurs a prohibitive preprocessing cost. (We revisit this point in Section 5.4.) Several works [20], [50] follow a more theoretical strategy to lower the construction cost of sketches. In particular, Fast similarity sketching [22] is a highly competitive approach, which we consider in our evaluation (Section 5.5).

Bloom filters can be used [31] to encode the profiles and then estimates Jaccard's index by using a bitwise `AND`. Despite providing privacy, the resulting loss in precision is prohibitive. *Sketches* [21] are other compacted datastructures, which have been used for instance to find frequent items in data streams [16]. Unfortunately sketches are not optimized for set intersection.

In many applications, such as image or video classifications, properly measuring the similarity between objects is difficult in itself. In such situations, metric learning [57], [60] can be used to learn a distance function between objects, using a set of training examples. Metric learning focuses on accuracy and seeks to generalize well to unseen items. As such, it is largely orthogonal to the problem we consider here, as we aim to approximate the Jaccard similarity as quickly as possible while maintaining an acceptable KNN quality. Closer to our work, hash learning [28], [46], [53], [56], [58] learns a hash function to perform dimension reduction before estimating similarities. Depending on the availability of training examples, this learning might be direct [46], weakly-supervised [56] or unsupervised [28], [58]. In the same vein, learned indexes [23], [39] learn a hierarchy of models to predict where a key might be found in a storage structure, while closely espousing a dataset's distribution. All those techniques rely on a learning step that is conceptually close to the preprocessing of MinHash (cf. Section 5.4). This learning adds a costly preparatory phase, which renders these approaches unattractive in our setting.

Surprisingly, state of the art algorithms for KNN queries [47] do not leverage similarity sketching. GoldFinger significantly improves their performance at a low implementation cost.

3 Problem, Intuition, and Approach

For ease of exposition, we consider in the following that nodes are *users* associated with *items* (e.g. web pages, movies, locations), without loss of generality.

3.1 Notations and problem definition

We note $U = \{u_1, ..., u_n\}$ the set of all users, and $I = \{i_1, ..., i_m\}$ the set of all items. The subset of items associated with user u (a.k.a. its *profile*) is noted $P_u \subseteq I$. P_u is generally much smaller than I (the universe of all items).

Our aim is to approximate a KNN graph G_{KNN} over U relying on some function sim computed over user profiles:

$$\begin{array}{rccc} sim: & U \times U & \rightarrow & \mathbb{R} \\ & (u,v) & & sim(u,v) = f_{sim}(P_u, P_v). \end{array}$$

f_{sim} is any similarity function over sets that is typically positively correlated with the number of common items between the two sets, and negatively correlated with the total number of items present in both sets. We focus on Jaccard's index in the rest of the paper [55].

Formally, a KNN graph G_{KNN} connects each user $u \in U$ with a set $knn(u)$ of k other users that maximize the similarity function $sim(u, -)$:

$$knn(u) \in \underset{v \in U \setminus \{u\}}{\text{argtop}^k} f_{sim}(P_u, P_v) \tag{1}$$

where argtop^k returns the set of k-tuples of $U \setminus \{u\}$ that maximize the similarity function $sim(u, -)$[4].

Computing an exact KNN graph is particularly expensive: an exhaustive search requires $O(|U|^2)$ similarity computations. Many scalable approaches therefore seek to construct *an approximate KNN graph* $\widehat{G}_{\text{KNN}}$, i.e., to find for each

4. In other words, argtop^k generalizes the concept of argument of the maximum (usually noted argmax) to the k top values of a function over a finite discrete set.

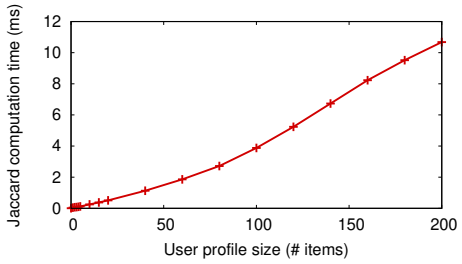


Fig. 1. Cost, averaged over 4.9 millions computations between randomly generated profiles on a Intel Xeon E5420@2.50GHz.

user u a neighborhood $\widehat{knn}(u)$ that is as close as possible to an exact KNN neighborhood [11], [24]. The meaning of 'close' depends on the context, but in most applications, a good approximate neighborhood $\widehat{knn}(u)$ is one whose aggregate similarity (its *quality*) comes close to that of an exact KNN set $knn(u)$.

We capture how well the average similarity of an approximated graph $\widehat{G}_{\text{KNN}}$ compares against that of an exact KNN graph G_{KNN} with the *average similarity* of $\widehat{G}_{\text{KNN}}$:

$$avg\_sim(\widehat{G}_{\text{KNN}}) = \mathop{\mathbb{E}}_{\substack{(u,v)\in U^2:\\ v\in\widehat{knn}(u)}} f_{sim}(P_u, P_v), \tag{2}$$

i.e. the average similarity of the edges of $\widehat{G}_{\text{KNN}}$. We then define the *quality* of $\widehat{G}_{\text{KNN}}$ as

$$quality(\widehat{G}_{\text{KNN}}) = \frac{avg\_sim(\widehat{G}_{\text{KNN}})}{avg\_sim(G_{\text{KNN}})}. \tag{3}$$

A quality close to 1 indicates that the approximate neighborhoods have a quality close to that of ideal neighborhoods, and can replace them with little loss in most applications.

With the above notations, we can summarize our problem as follows: for a given dataset $(U, I, (P_u)_{u\in U})$ and item-based similarity f_{sim}, we wish to compute an approximate $\widehat{G}_{\text{KNN}}$ in the shortest time with the highest overall quality.

3.2 Intuition

A large portion of a KNN graph's construction time often comes from computing individual similarity values (up to 90% of the total construction time [12]). This is because computing explicit similarity values on even medium-sized profiles can be relatively expensive. Figure 1 shows on its y-axis the time required to compute Jaccard's index $J(P_1, P_2) = \frac{|P_1\cap P_2|}{|P_1\cup P_2|}$ between two user profiles of the same size, when this size varies (x-axis). The cost of computing a single index is relatively high even for medium-size profiles: 2.7 ms for two random profiles of 80 items, a typical profile size of the datasets we have considered.

Earlier KNN graph construction approaches have therefore sought to limit the number of similarity computations [11], [24]. They typically adopt a greedy strategy, starting from a random graph, and progressively converging to a better KNN approximation. This strategy dramatically reduces the similarity computations they perform, and are easily parallelizable, but it is now difficult to see how their greedy component could be further improved.

In order to overcome the inherent cost of similarity computations, we propose in this paper to target the data on which computations run, rather than the algorithms that drive these computations. This strategy stems from the observation that *explicit* datastructures (hash tables, arrays) incur substantial costs. To avoid these costs, we advocate the use of *fingerprints*, a *compact*, *binary*, and *fast-to-compute* representation of data. Our intuition is that, with almost no overhead, fingerprints can capture enough of the characteristics of the data to provide a good approximation of similarity values, while drastically reducing the cost of computing these similarities.

TABLE 1
Effect of SHFs on computation time of Jaccard's index, compared to Fig. 1 (80 items).

SHF length (bits)	Comp. Time (ms)	Speedup $\lvert P\rvert = 80$
64	0.011	*253*
256	0.032	*84*
1024	0.120	*23*
4096	0.469	*6*

3.3 GoldFinger and Single Hash Fingerprints

Our approach, dubbed *GoldFinger*, extracts from each user's profile a *Single Hash Fingerprint* (SHF for short). An SHF is a pair $(B, c) \in \{0,1\}^b \times \mathbb{N}$ comprising a bit array $B = (\beta_x)_{x\in[\![0..b-1]\!]}$ of b bits, and an integer c, which records the number of bits set to 1 in B (its L1 norm, which we call the *cardinality* of B in the following). The SHF of a user's profile P is computed by hashing each item of the profile into the array and setting to 1 the associated bit

$$\beta_x = \begin{cases} 1 & \text{if } \exists e \in P : h(e) = x, \\ 0 & \text{otherwise,} \end{cases}$$
$$c = \left\|(\beta_x)_x\right\|_1$$

where $h()$ is a uniform hash function from I to $[\![0..b-1]\!]$, and $\|\cdot\|_1$ counts the number of bits set to 1.

Benefits in terms of space and speed: The length b of the bit array B is usually much smaller than the total number of items, which causes collisions, and a loss of information. This loss is counterbalanced by the highly efficient approximation SHFs can provide of any set-based similarity. The Jaccard's index of two user profiles P_1 and P_2 can be estimated from their respective SHFs (B_1, c_1) and (B_2, c_2) with

$$\widehat{J}(P_1, P_2) = \frac{\|B_1 \textbf{ AND } B_2\|_1}{c_1 + c_2 - \|B_1 \textbf{ AND } B_2\|_1}, \tag{4}$$

where B_1 **AND** B_2 represents the bitwise AND of the bit-arrays of the two profiles. This formula exploits two observations that hold generally with few collisions in the bit arrays (a point we revisit below). First, the size of a set of items P can be estimated from the cardinality of its SHF (B_P, c_p):

$$|P| \approx \|B_P\|_1 = c_p. \tag{5}$$

Second, the bit array $B_{(P_1\cap P_2)}$ of the intersection of two profiles $P_1 \cap P_2$ can be approximated with the bitwise AND of their respective bit-arrays, B_1 and B_2:

$$B_{(P_1\cap P_2)} \approx (B_1 \textbf{ AND } B_2). \tag{6}$$

Equation (4) combines these two observations along with some simple set algebra ($|P_1 \cup P_2| = |P_1| + |P_2| - |P_1 \cap P_2|$).

TABLE 2
Comparison of the time spend (ms) to compute 10000 similarity computations on movielens10M and AmazonMovies using GoldFinger and regular Bloom filters, with $b = 1024$ bits each. GoldFinger achieves speeds-up up to 65%.

	GolFi	BF	gain(%)
ml10M	**2216.32**	6444.52	**65.61**
AM	**2103.98**	5886.98	**64.26**

The computation incurred by (4) is much faster than on explicit profiles and is independent of the actual size of the explicit profiles. This is illustrated in Table 1 which shows the computation time of Eq. (4) on the same profiles as Figure 1 for SHFs of different lengths (as in Fig. 1, the values are averaged over 4.9 million computations). For instance, estimating Jaccard's index between two SHFs of 1024 bits (the default in our experiments) takes 0.120 ms, a 23-fold speedup compared to two explicit profiles of 80 items.

The link with Bloom Filters and collisions: SHFs can be interpreted as a highly simplified form of Bloom filters, and suffer from errors arising from collisions, as Bloom filters do. However, the two structures serve different purposes: whereas Bloom filters are designed to test whether individual elements belong to a set, SHFs are designed to approximate set similarities (in this example Jaccard's index). Still, given the Bloom filters B and B' representing the sets S and S' respectively, we can estimate the size of the set intersection $S \cap S'$ by [54]:

$$n_{S\cap S'} = -\frac{b}{p}\log[1-\frac{||B||_1}{b}]-\frac{b}{p}\log[1-\frac{||B'||_1}{b}] + \frac{b}{p}\log[1-\frac{||B^{S\cup S'}||_1}{b}]$$

with $||B^{S\cup S'}||_1$ being the cardinality of $B^{S\cup S'} = (\beta_x^{S\cup S'})_{x\in[\![1..b]\!]}$ the union of the arrays of B and B' defined by:

$$\forall x \in [\![1..b]\!], \beta_x^{S\cup S'} = \beta_x \vee \beta'_x$$

We compared the computation time of the Jaccard's index using both datastructures: *(i)* a standard implementation of Bloom filters[5] (labeled *BF*) and *(ii)* the SHFs (labeled *GolFi*). Both of them use 1024 bits and the Bloom filters only have one hash function. We compute 10000 similarities between pairs of users of movielens10M and AmazonMovies. The results are displayed in Table 2. SHFs outperform Bloom filters.

In addition to computation time, Bloom filters also use multiple hash functions to minimize false positives when answering set membership queries. Multiple hash functions, however, increase single-bit collisions, and therefore degrade the approximation provided by SHFs.

3.4 Privacy

The noise introduced by collisions brings extra privacy benefits and provides *k*-anonymity and *l*-diversity. More details can be found available in our conference version [33]. It is important to note that SHFs were not designed to provide privacy but faster similarity computation. The above two privacy properties are therefore obtained "for free", as a side effect of our approach. They do not preclude, however, the use of additional strengthening privacy-protection mechanisms—such as the insertion of random noise to the SHF [4] to obtain differential privacy guaranties [26]—albeit typically at the cost of reduced performances.

5. https://github.com/Baqend/Orestes-Bloomfilter

3.5 Formal analysis of the Jaccard estimator

For readability in this section, we will generally treat bit arrays (i.e. belonging to $\{0,1\}^n$) as sets of bit positions (belonging to $\mathcal{P}([\![0, b-1]\!])$). We will also note B_X the set of bit positions set to 1 by the (sub)profile P_X: $B_X = h(P_X)$.

For two given profiles P_1 and P_2, the distribution of $\widehat{J}(P_1, P_2)$ is governed by how the random hash function h maps the items of P_1 and P_2 onto the bit positions $[\![0, b-1]\!]$.

In the conference version [33], we provided the probability distribution of $\widehat{J}(P_1, P_2)$. It can be used to plot the behavior of the estimator $\widehat{J}$ against the real Jaccard index when comparing a profile with other profiles. $\widehat{J}$ is biased but the absolute bias has limited impact on KNN algorithms, which only need to *order* nodes correctly, rather than to predict exact similarities. The probability of such misordering to occur can be computed with the probability distribution and it was shown to be very low in our examples.

In the following, we prove that the estimator $\widehat{J}$ is proportional to the real Jaccard similarity up to some small error introduced by collisions.

Theorem 1. *The estimator $\widehat{J}_{1,2} = \widehat{J}(P_1, P_2)$ of the Jaccard similarity $J_{1,2} = J(P_1, P_2)$ between u_1's and u_2's profiles, P_1 and P_2, is lower bounded by the following inequality*

$$J_{1,2} - \frac{\kappa}{\ell} \leq \widehat{J}_{1,2}, \tag{7}$$

where $\ell = |P_1 \cup P_2|$ is the joint size of the two profiles; $\kappa = \ell - |h(P_1 \cup P_2)|$ is the overall number of collisions occurring when projecting the two profiles onto $[\![1, b]\!]$. If we further assume $\kappa \leq \ell/2$, then we have

$$\widehat{J}_{1,2} \leq J_{1,2} + 3\frac{\kappa}{\ell} + O\left(\frac{\kappa}{\ell}\right)^2. \tag{8}$$

Proof. For brevity, let us note $P_\cap = P_1 \cap P_2$ the set of items that are present in both profiles. Compared to $P_\cap$, collisions may both increase $h(P_1) \cap h(P_2)$ (if they occur between elements of $P_1 \Delta P_2$, the symmetric difference of the users' profiles, i.e. the items that only appear in one of the two profiles), or decrease it (if they occur between elements of $P_\cap$), yielding

$$|P_\cap| - \kappa \leq |h(P_1) \cap h(P_2)| \leq |P_\cap| + \kappa, \tag{9}$$

where κ is the number of collisions caused by h on $P_1 \cup P_2$. Since by definition $|h(P_1 \cup P_2)| = \ell - \kappa$, we get

$$\frac{|P_\cap|/\ell - \kappa/\ell}{1-\kappa/\ell} \leq \widehat{J}_{1,2} \leq \frac{|P_\cap|/\ell + \kappa/\ell}{1-\kappa/\ell} \tag{10}$$

$$\frac{J_{1,2} - \kappa/\ell}{1-\kappa/\ell} \leq \widehat{J}_{1,2} \leq \frac{J_{1,2} + \kappa/\ell}{1-\kappa/\ell}. \tag{11}$$

Since $\kappa < \ell$, we trivially have $\frac{J_{1,2}-\kappa/\ell}{1-\kappa/\ell} \geq J_{1,2} - \frac{\kappa}{\ell}$, thus proving (7). If we assume $\kappa \leq \ell/2$, we can use the fact that

$\frac{1}{1-x} \leq 1+2x$ when $x \in [0, \frac{1}{2}]$, which yields, together with $J_{1,2} \leq 1$

$$\frac{J_{1,2}+\kappa/\ell}{1-\kappa/\ell} \leq \left(J_{1,2}+\frac{\kappa}{\ell}\right)\left(1+2\frac{\kappa}{\ell}\right) \leq J_{1,2}+3\frac{\kappa}{\ell}+O\left(\frac{\kappa}{\ell}\right)^2$$

□

We now bound the effect of collisions with the following concentration bound.

Theorem 2. *The collision density κ/ℓ is upper bounded by the value $(1+d)\frac{\ell-1}{2b}$ with a probability bounded by the following formula*

$$\mathbb{P}\left[\frac{\kappa}{\ell} < (1+d)\frac{\ell-1}{2b}\right] \geq 1-\left(\frac{e^d}{(1+d)^{(1+d)}}\right)^{\frac{\ell(\ell-1)}{2b}}, \tag{12}$$

where $d > 0$ is a real positive value, and the other variables are defined as in Theorem 1.

For space reason, we only give a sketch of the proof. This result is obtained by considering the projection of every item as an random variable. The realizations of these random variables and their sum is then upper bounded. We finally apply a multiplicative Chernoff bound to obtain the concentration bound. See for instance [30] for more details on a similar proof.
As an example, if we use the case $\ell = 256$, $b = 1024$ (typical values of our experiments) and set $d = 0.5$ we obtain that

$$J_{1,2} - 0.187 \leq \widehat{J}_{1,2} \leq J_{1,2} + 0.374$$

with probability 0.968.

4 Experimental Setup

4.1 Datasets

We evaluate GoldFinger on six publicly available datasets (Table 3). To apply Jaccard's index, we binarize each dataset by only keeping in a user profile P_u those items that user u has rated higher than 3.

Movielens: Movielens [34] is a group of anonymous datasets containing movie ratings collected on-line between 1995 and 2015 by GroupLens Research [52]. The datasets (before binarization) contain movie ratings on a 0.5-5 scale by users who have at least performed 20 ratings. We use 3 versions of the dataset, movielens1M (ml1M), movielens10M (ml10M) and movielens20M (ml20M), containing between 575,281 and 12,195,566 positive ratings (i.e. higher than 3).

AmazonMovies: AmazonMovies [48] (AM) is a dataset of movie reviews from Amazon collected between 1997 and 2012. We restrain our study to users with at least 20 ratings (before binarization) to avoid users with not enough data (this problem, the *cold start problem*, is generally treated separately [41]). After binarization, the dataset contains 57,430 users; 171,356 items; and 3,263,050 ratings.

DBLP: DBLP [59] is a dataset of co-authorship from the DBLP computer science bibliography. In this dataset, both the user set and the item set are subsets of the author set. If two authors have published at least one paper together, they are linked, which is expressed in our case by both of them rating each other with a rating of 5. As with AM, we only consider users with at least 20 ratings. The resulting dataset contains 18,889 users, 203,030 items; and 692,752 ratings.

Gowalla: Gowalla [19] (GW) is a location-based social network. As DBLP, both the user set and the item set are subsets of the set of the users of the social network. The undirected friendship link from u to v is represented by u rating v with a 5. As previously, only the users with at least 20 ratings are considered. The resulting dataset contains 20,270 users, 135,540 items; and 1,107,467 ratings.

4.2 Baseline algorithms and competitors

We apply GoldFinger to four existing KNN algorithms: Brute Force (as a reference point), NNDescent [24], Hyrec [11] and LSH [35]. We compare the performance and results of each of these algorithms in their native form (*native* for short) and when accelerated with GoldFinger (*GolFi* for short). For completeness, we also consider three direct competitors *b-bit minwise hashing* [43] and *Fast similarity sketching* [22].

b-bit minwise hashing (MinHash): A standard technique to approximate Jaccard's index values between sets is the *MinHash* [14] algorithm. MinHash creates multiple independent permutations on the IDs of an item universe, and keeps for each profile the item with the smallest ID, after each permutation. The Jaccard's index between two profiles can be estimated by counting the proportion of minimal IDs that are equal in the compacted representation of the two profiles. *MinHash* was extended to keep only the lowest b bits of each minimal element [43]. This approach, called *b-bit minwise hashing* (MinHash for short), creates very compact binary summaries of profiles, comparable to our SHFs, from which a Jaccard's index can be estimated.

Fast similarity skeching (FastSim): Fast similarity sketching [22] was designed to alleviate the expensive preprocessing cost of MinHash. Instead of sampling with replacement one item for each position of the sketch, as MinHash does, FastSim mixes sampling with replacement and sampling without replacement. For a sketch of length t, FastSim uses $2t$ random hash functions that are used to assign the items to a position in the sketch and to a random value. The first t hash functions randomly assign items to positions while the last t hash functions deterministically assign items to a set position –one position per hash function– to ensure that every position contains at least one item. At each position of the sketch, the minimum value of the associated items is stored. To improve the performances, the sketch is filled with items one hash function at a time and the process is stopped whenever every position has at least one value. The similarity using FastSim is estimated in a similar way as with MinHash: by counting the proportion of equal values in the sketch, position-wise. Constructing a sketch of length t from a set S has a complexity of $t + |S|$, against $t \times |S|$ for MinHash. Albeit its lower complexity, FastSim is not as compact as the densified variation of MinHash as it stores floats instead of bits.

Brute force: The Brute Force algorithm computes the similarities between every pair of profiles, performing a constant number of similarity computations equal to $\frac{n\times(n-1)}{2}$. While this is computationally intensive, this algorithm produces an exact KNN graph.

TABLE 3
Description of the datasets used in our experiments

Dataset	Users	Items	Scale	Ratings > 3	$\lvert P_u \rvert$	$\lvert P_i \rvert$	Density
movielens1M (ml1M) [34]	6,038	3,533	1-5	575,281	95.28	162.83	2.697%
movielens10M (ml10M) [34]	69,816	10,472	0.5-5	5,885,448	84.30	562.02	0.805%
movielens20M (ml20M) [34]	138,362	22,884	0.5-5	12,195,566	88.14	532.93	0.385%
AmazonMovies (AM) [48]	57,430	171,356	1-5	3,263,050	56,82	19.04	0.033%
DBLP [59]	18,889	203,030	5	692,752	36.67	3.41	0.018%
Gowalla (GW) [19]	20,270	135,540	5	1,107,467	54.64	8.17	0.040%

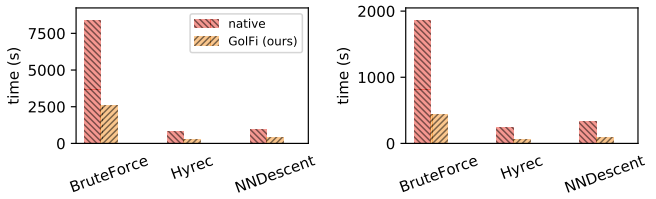
NNDescent: NNDescent [24] constructs an approximate KNN graph (or ANN) by relying on a local search and by limiting the number of similarities computations. NNDescent starts from an initial random graph, which is then iteratively refined to converge to an ANN graph. During each iteration, for each user u, NNDescent compares all the pairs (u_i, u_j) among the neighbors of u, and updates the neighborhoods of u_i and u_j accordingly. NNDescent includes a number of optimizations: it exploits the order on user IDs, and maintains update flags to avoid computing several times the same similarities. It also reverses the current KNN approximation to increase the space search among neighbors. The algorithm stops either when the number of updates during one iteration is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations.

Hyrec: Hyrec [11] uses a strategy similar to that of NNDescent, exploiting the fact that a neighbor of a neighbor is likely to be a neighbor. As NNDescent, Hyrec starts with a random graph which is then refined. Hyrec primarily differs from NNDescent in its iteration strategy. At each iteration, for each user u, Hyrec compares all the neighbors' neighbors of u with u, rather than comparing u's neighbors between themselves. Hyrec also does not reverse the current KNN graph. As NNDescent, it stops when the number of changes is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations.

LSH: Locality-Sensitive-Hashing (LSH) [35] reduces the number of similarity computations by hashing each user into several buckets. Neighbors are then selected among users found in the same buckets. To ensure that similar users tend to be hashed into the same buckets, LSH uses min-wise independent permutations of the item set as its hash functions, similarly to the MinHash algorithm [14].

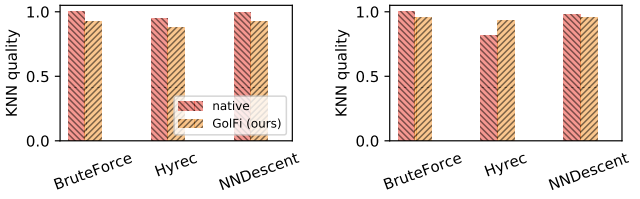
4.3 Experimental settings

The value of k controls a trade-off between computation-time and quality for NNDescent and Hyrec: the larger k the higher the quality but the longer the computation. For a given quality, some datasets may require a larger k such as DBLP [24]. This trade-off is studied in details in the original papers of the algorithms. It is orthogonal to our study as we compare GoldFinger and its competitors using the same parameters. For simplicity, we chose the same default value for all datasets and $k = 30$ offered a good balance across the board. The parameters of Hyrec and NNDescent are the default ones from the original papers [11], [24]: the parameter δ of Hyrec and NNDescent is set to 0.001, and their maximum number of iterations to 30. The number of hash functions for LSH is 10, as this provides a good trade-off between execution time and graph quality, and is in-line with values used in earlier studies [25]. GoldFinger uses 1024 bits long SHFs computed with Jenkins' hash function [36].



(a) movielens20M (b) AmazonMovies

Fig. 2. Execution time using a 1024-bit SHF (lower is better). GoldFinger (GolFi) outperforms Brute Force, Hyrec and NNDescent in their native version.



(a) movielens20M (b) AmazonMovies

Fig. 3. KNN quality using a 1024-bit SHF (higher is better). GoldFinger (GolFi) only experiences a small decrease in quality.

Evaluation metrics: We measure the effect of GoldFinger on Brute Force, Hyrec, NNDescent and LSH along with two main metrics: *(i)* their computation *time* (measured from the start of the algorithm, once the dataset has been prepared), and *(ii)* the *quality* of the resulting KNN (Sec. 3.1). Throughout our experiments, we use a 5-fold cross-validation, and average results on the 5 resulting runs.

Implementation details and hardware: We have implemented Brute Force, Hyrec, NNDescent and LSH (with and without GoldFinger) in Java 1.8. Our experiments run on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads, 32GB of memory, and a HHD of 750GB. Unless stated otherwise, we use all 8 threads. Our code is available online[6].

5 Evaluation Results

We first discuss in detail the impact of GoldFinger on Brute Force, Hyrec, NNDescent, and LSH, before comparing it against MinHash and FastSim.

5.1 Brute Force, Hyrec, NNDescent, and LSH

The impact of GoldFinger (*GolFi*) on these four algorithms is summarized in Table 4 in terms of execution time and KNN

6. https://gitlab.inria.fr/oruas/SamplingKNN

TABLE 4
Computation time and KNN quality with native algorithms (*nat.*) and GoldFinger (GolFi). GoldFinger yields the shortest computation times across all datasets (in bold), yielding gains (*gain*) of up to 78.9% against native algorithms. The loss in quality is at worst moderate, ranging from 0.22 to an improvement of 0.11.

datasets		*comp. time (s)*			*KNN quality*		
	algo	**nat.**	**GolFi**	*gain%*	**nat.**	**GolFi**	*loss*
ml1M	Brute Force	19.0	4.0	*78.9*	1.00	0.93	*0.07*
	Hyrec	14.4	4.4	*69.4*	0.98	0.92	*0.06*
	NNDescent	19.0	11.0	*42.1*	1.00	0.93	*0.07*
	LSH	9.5	**3.0**	*68.4*	0.98	0.92	*0.06*
ml10M	Brute Force	2028	606	*70.1*	1.00	0.94	*0.06*
	Hyrec	314	**110**	*65.0*	0.96	0.90	*0.06*
	NNDescent	374	147	*60.7*	1.00	0.93	*0.07*
	LSH	689	255	*63.0*	0.99	0.94	*0.06*
ml20M	Brute Force	8393	2616	*68.8*	1.00	0.92	*0.08*
	Hyrec	842	**289**	*65.7*	0.95	0.88	*0.07*
	NNDescent	919	383	*58.3*	0.99	0.92	*0.07*
	LSH	2859	1060	*62.9*	0.99	0.93	*0.06*
AM	Brute Force	1862	435	*76.6*	1.00	0.96	*0.04*
	Hyrec	235	**62**	*73.6*	0.82	0.93	*-0.11*
	NNDescent	324	91	*71.9*	0.98	0.95	*0.03*
	LSH	144	141	*2.1*	0.98	0.96	*0.02*
DBLP	Brute Force	100	46	*54.0*	1.0	0.82	*0.18*
	Hyrec	46	27	*41.3*	0.86	0.81	*0.05*
	NNDescent	31	**24**	*22.6*	0.98	0.82	*0.16*
	LSH	40	38	*2.6*	0.87	0.86	*0.01*
Gowalla	Brute Force	160	54	*66.3*	1.0	0.78	*0.22*
	Hyrec	39	**22**	*43.6*	0.95	0.78	*0.17*
	NNDescent	45	26	*42.2*	1.0	0.79	*0.21*
	LSH	30	27	*3.7*	0.87	0.82	*0.05*

quality. The columns marked *nat.* indicate the results with the native algorithms, while those marked *GolFi* contain those with GoldFinger. The columns in italics show the gain in computation time brought by GoldFinger (*gain* %), and the loss in quality (*loss*). The fastest time for each dataset is shown in bold. Excluding LSH for space reasons, the same results are shown in Figures 2 (time) and 3 (quality).

Overall, GoldFinger delivers the fastest computation times across all datasets, for a small loss in quality ranging from 0.22 (with Brute Force on Gowalla) to an improvement of 0.11 (Hyrec on AmazonMovies). Excluding LSH on AmazonMovies, DBLP, and Gowalla for the moment, GoldFinger is able to reduce computation time substantially, from 42.1% (NNDescent on ml1M) to 78.9% (Brute Force on ml1M), corresponding to speedups of 1.72 and 4.74 respectively. Experiments show that the most important parameter is the distribution of ratings, rather than the sparsity or the number of items/users. The more ratings are concentrated onto a small subset of items, reducing collisions, the higher the accuracy provided by GoldFinger. Ratings appear to be more concentrated of few items in movie datasets which explain the differences between the datasets.

On DBLP and Gowalla, the use of GoldFinger reduces quality more significantly than on other datasets. This varying impact on quality stems from the characteristics of individual datasets, whose influence is studied in more depth in Section 6. In general, this effect can be reduced by using larger SHFs. Overall, however, and despite the distortion that GoldFinger inherently causes, Hyrec, NNDescent, and LSH still provide fair approximations of the KNN graph obtained by Brute Force with GoldFinger.

TABLE 5
L1 stores and L1 loads with the native algorithms (*nat.*) and GoldFinger (GolFi) on ml10M. GoldFinger drastically reduces the number of L1 accesses, yielding reductions (*gain*) of up to 87.9%.

	L1 stores ($\times 10^{12}$)			*L1 loads* ($\times 10^{12}$)		
algo	**nat.**	**GolFi**	gain%	**nat.**	**GolFi**	gain%
Brute Force	2.82	0.34	87.9	8.26	1.08	86.9
Hyrec	0.35	0.08	77.1	1.14	0.28	75.4
NNDescent	0.57	0.16	71.9	1.93	0.59	69.4
LSH	0.84	0.85	-1.19	2.96	2.90	2.03

The improved KNN quality observed when GoldFinger is applied to Hyrec on AmazonMovies arises from the interplay between Hyrec's internal workings and the noise introduced by GoldFinger. On AmazonMovies, native Hyrec stops exploring new neighbors too early, resulting in a degraded KNN quality. The use of GoldFinger delays Hyrec's convergence, and forces it to perform additional iterations. The cost of these additional iterations, however, is more than offset by the speed gains GoldFinger brings. In addition, this extra exploration significantly improves KNN quality, and overcompensates the loss usually caused by GoldFinger. We revisit this effect in Section 6.2.

GoldFinger only has a limited effect on the execution time of LSH on the AmazonMovies, DBLP, and Gowalla datasets. This lack of impact can be explained by the characteristics of LSH and the datasets. LSH must first create user buckets using permutations on the item universe, an operation that is proportional to the number of items. Because AmazonMovies, DBLP, and Gowalla are comparatively very sparse (Table 3), the buckets created by LSH tend to contain few users. As a result, the overall computation time is dominated by the bucket creation, limiting the impact of GoldFinger.

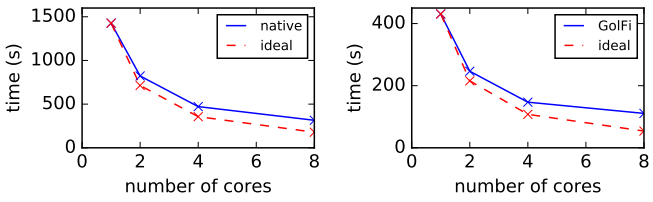
Despite these results, GoldFinger consistently outperforms native LSH on these datasets, for instance, taking 62s (with Hyrec) instead of 141s with LSH on AmazonMovies (a speedup of $\times 2.27$), for a comparable quality.

5.2 Memory and cache accesses

By compacting profiles, GoldFinger reduces the amount of memory needed to process a dataset. To gauge this effect, we use the `perf`[7] command line tool to profile the memory accesses of GoldFinger. `perf` uses hardware counters to measure accesses to the cache hierarchy (L1, LLC, and physical memory). To eliminate accesses performed during the dataset preparation, we subtract the values returned by `perf` when only preparing the dataset from the values obtained on a full execution.

Table 5 summarizes the measures obtained on Brute Force, Hyrec, NNDescent and LSH on ml10M, both without (*native*) and with GoldFinger (*GolFi*). We only show L1 accesses for space reasons, since LLC and RAM accesses are negligible in comparison. Except on LSH, GoldFinger significantly reduces the number of L1 cache loads and stores,

7. https://perf.wiki.kernel.org/index.php/Main_Page



(a) Native Hyrec (b) Hyrec + GoldFinger

Fig. 4. Effect of the number of cores on Hyrec for ml10M. GoldFinger (GolFi) preserves the scalability of the algorithm.

TABLE 6
Preprocessing time of each dataset for the native approach, b-bit minwise hashing (MinHash) & GoldFinger. The preprocessing consists in loading the data and creating the datastructure. The associated computation time does not include any KNN graph computation. GoldFinger is orders of magnitude faster than MinHash, up to $\times 3255.2$ faster than MinHash on DBLP, whose overhead is prohibitive.

Dataset	Native	MinHash	GoldFinger	speedup (×)
ml1M	0.37s	6.24s	0.31s	20.1
ml10M	3.90s	203s	3.24s	62.7
ml20M	8.71s	820s	7.06s	116.1
AM	3.40s	3250s	1.92s	1692.7
DBLP	0.42s	944s	0.29s	3255.2
GW	0.47s	594s	0.40s	1485.0

confirming the benefits of GoldFinger in terms of memory footprint. For LSH, L1 accesses are almost not impacted by GoldFinger. Again, we conjecture this is because memory accesses are dominated by the creation of buckets.

5.3 Scalability: number of cores

Because GoldFinger modifies both the memory accesses and the breakdown between different computation activities, it could perturb the native scalability of the algorithm it is applied to. Figure 4 shows this is not the case, by plotting the execution time of Hyrec on ml10M both without (Figure 4a, left) and with GoldFinger (Figure 4b, right) when increasing the number of available cores from 1 to 8. The dotted-line (labeled *ideal*) on both charts represents an algorithm that would scale perfectly: for a given number of cores, its represented value is the value for one core divided by the number of cores. Comparing Figures 4a and 4b shows that GoldFinger, in addition to the speedup, preserves the native scalability of the underlying algorithm (here Hyrec).

5.4 MinHash

We use $b = 4$ and 256 permutations for BBHM (a configuration that provides the best trade-off between time and KNN quality). MinHash decreases the computation time on ml10M from 2028s to 1028s with a Brute Force approach, while delivering a quality of 0.93. GoldFinger (with 1024 bits) is both substantially faster (producing a graph in 606s, a 41.1% improvement compared to MinHash), and better (delivering a quality of 0.94). Similar results are obtained on other datasets, or when using LSH (with an average temporal gain of 34.8% compared to MinHash, temporal gains ranging from 2.7% to 58.8%, and while producing better graphs across the board). When used on AmazonMovies with NNDescent and Hyrec, MinHash causes these two algorithms to collapse, yielding qualities below 0.11. Together these results show GoldFinger outperforms MinHash by a wide margin. Furthermore, computing MinHash summaries is extremely costly (as it requires creating a large number of permutations on the entire item set), which renders the approach self-defeating in our context. Table 6 summarizes the time required to load and construct the internal representation of each dataset when using a native (explicit) approach, GoldFinger (using Jenkins' hash function [36]), and MinHash. MinHash is used as baseline for the speedup computation to stress the difference of speed between the two approaches. Whereas GoldFinger is slightly faster than a native approach (as it does not need to create extensive in-memory objects to store the dataset), MinHash is one to 3 orders of magnitude slower than GoldFinger (1692 times slower on AmazonMovies for instance). This kind of overhead makes it impractical for environments with limited resources.

TABLE 7
Preprocessing time for the native approach, FastSim with $t = 64$ and GoldFinger. GoldFinger is slightly faster but FastSim is competitive.

Dataset	Nat.	FastSim	GolFi	speedup (×)
ml10M	5.27s	6.45s	4.01s	1.6
AM	2.68s	3.65s	2.12s	1.7
DBLP	0.27s	0.47s	0.21s	2.3

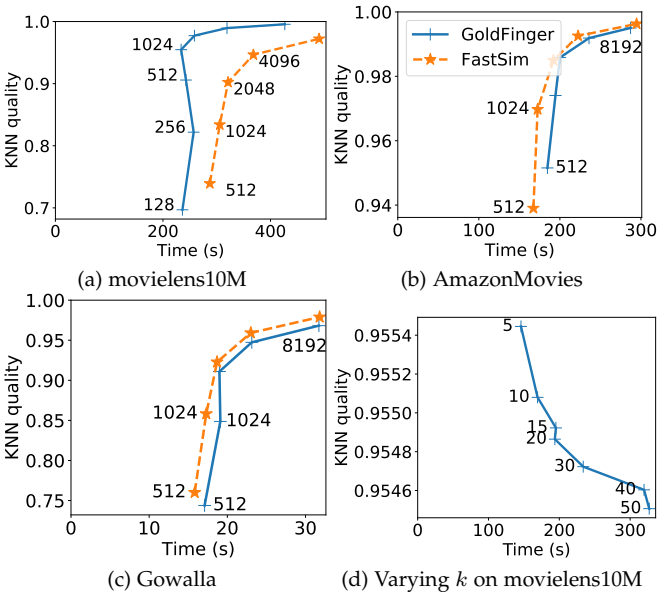
5.5 Fast similarity sketching

The experiments in this section[8] were run on an Intel Xeon E5-2630@2.40GHz using 8 cores (out of the 16 available cores) with 125GB of memory.

Table 7 summarizes the time required to construct the sketches for FastSim, GoldFinger and the speedup of GoldFinger compared to FastSim on movielens10M, AmazonMovies and DBLP. We used FastSim with $t = 64$ as it offered the best trade-off between quality and computation time. As each stored value is encoded as a float, the resulting sketches were $64 \times 32 = 2048$ bits long. During the preprocessing step, GoldFinger is slightly faster than Fast similarity sketching ($\times 1.6$) for the initialization part. Still, FastSim remains an acceptable competitor as the preprocessing step is negligible compared to the total KNN graph computation.

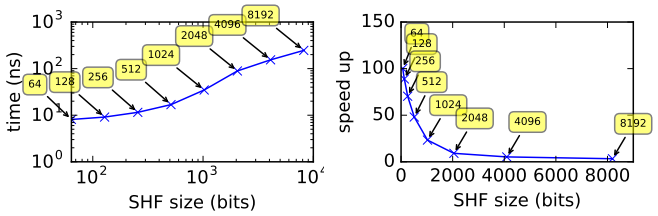
Figures 5a, 5b and 5c explore the trade-off between computation time and KNN quality when increasing the space available to GoldFinger and FastSim to summarize individual items. We use a brute force approach on movielens10M, AmazonMovies and Gowalla. The bit-size of item summaries ranges over $\{512, 1024, 2048, 4096, 8192\}$ for both GoldFinger and FastSim, except on movielens10M on which we also consider 128 and 256 bits. For FastSim, those values correspond to a number of hash functions ranging from 16 to 256. The results are averaged over four runs. On movielens10M, which is comparatively dense, GoldFinger significantly outperforms FastSim, even for small space budgets. On AmazonMovies and Gowalla, FastSim performs slightly

8. Code is available at https://gitlab.aliens-lyon.fr/gniot/samplingknn



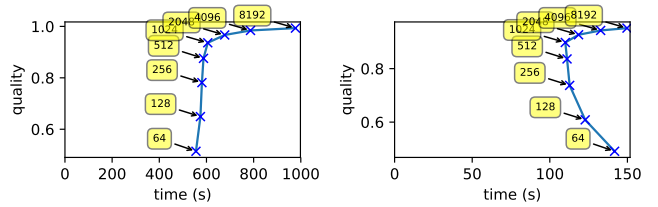
(a) movielens10M (b) AmazonMovies (c) Gowalla (d) Varying k on movielens10M

Fig. 5. Relation between the computation time and the KNN quality. Figures 5a, 5b and 5c display the relation for different sizes using FastSim and GoldFinger. GoldFinger outperforms FastSim on movielens10M, a dense dataset. Figure 5d shows that k has close to no impact on the quality of the resulting KNN graph when using GoldFinger (1024 bits, on movielens10M).



(a) Time to compute one similarity (b) Speedup

Fig. 6. Effect of the size of the SHF on the similarity computation time, on ml10M.



(a) Brute Force (b) Hyrec

Fig. 7. Relation between the execution time and the quality in function of the size of SHF.

better on low space budgets, with this advantage decreasing as more bits are used. This difference can be explained by how GoldFinger behaves on different datasets: GoldFinger is more efficient on denser datasets. Additional experiments show that if we artificially increase the density of a dataset —by randomly merging items— the KNN quality increases for both approaches but GoldFinger widens the gap with FastSim and is the clear winner for dense datasets.

Overall, these results show that GoldFinger is competitive against one of the most up-to-date sketching approaches and clearly outperforms this approach on a dense dataset.

5.6 Impact of k

The figure 5d shows the impact of k on the trade-off between computation time and quality when computing a KNN with a brute force approach and GoldFinger with 1024-bit SHFs on movielens10M. Note that the axis for the quality has a small range: k has a negligible impact on the quality. The larger k, the longer the computation, as expected as the cost to maintain a sorted neighborhood of size k for each user increases with k.

6 Sensitivity Analysis

The size of SHFs (b) determines the number of collisions occurring when computing SHFs, and when intersecting them. It thus affects the obtained KNN quality. Shorter SHFs also deliver higher speedups, resulting in an inherent trade-off between execution time and quality. In this section we focus on ml10M to explore this trade-off.

6.1 Impact on the similarity computation time

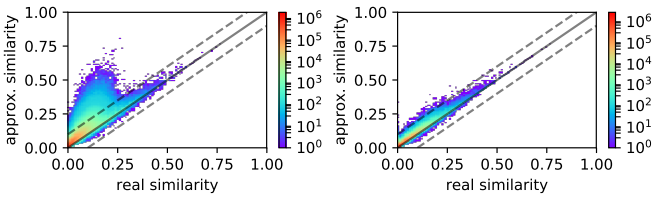
SHFs aim at drastically decreasing the cost of individual similarity computations. To assess this effect, Figure 6 shows the average computation time of one similarity computation when using SHFs (Eq. 4) and its corresponding speedup. The measures were obtained by computing with a multithreaded program the similarities between two sets of 5×10^4 users, sampled randomly from ml10M. The first set of users is partitioned into as many subsets as there are threads. On each thread, each user of the partition from the first set is compared to every user of the second set. The total time required is divided by the total number of similarities computed, 2.5×10^9, and then averaged over 4 runs. The computation time is linear in the size of the SHF. Computation time spans from 8 nanoseconds to 250 nanoseconds using SHF, against 800 nanoseconds with real profiles. The other datasets show similar results.

6.2 Impact on the execution of the algorithm

Figure 7 shows how the overall execution time and the quality of Brute Force and Hyrec evolve when we increase the size of the SHFs. (LSH presents a similar behavior to that of Brute Force, and NNDescent to that of Hyrec.)

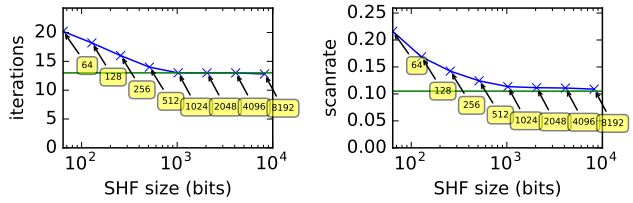
As expected, larger SHFs cause Brute Force to take longer to compute, while delivering a better KNN quality (Fig. 7a). The overall computation time does not exactly follow that of individual similarity computations (Figure 6a), as the algorithm involves additional bookkeeping work, such as maintaining the KNN graph and iterating over the dataset.

The KNN quality of Hyrec shows a similar trend, increasing with the size of SHFs. The computation time of Hyrec presents, however, an unexpected pattern: it first *decreases* when SHFs grow from 64 to 1024 bits, before increasing again from 1024 to 4096 bits (Figure 7b). This difference is due to the different nature of the two approaches. The Brute Force algorithm computes a fixed number of similarities, which is independent of the distribution of similarities between users. By contrast, Hyrec adopts a greedy approach: the number of similarities computed depends on the iterations performed by the algorithm, and these iterations are highly dependent on the distribution of similarity values



(a) 1024 bits (b) 4096 bits

Fig. 8. Heatmaps similarities on ml10M. The distortion of the similarity decreases when the size of SHF augments.



(a) Number of iterations (b) Scan rate

Fig. 9. Effect of compression on the convergence of Hyrec on ml10M. GoldFinger converges to the native approach when the size of SHF augments.

between pairs of users (what we have termed the *similarity topology* of the dataset), a phenomenon we return to in the following section.

6.3 Impact on estimated similarity values

Figure 8 shows how SHFs tend to distort estimated similarity values between pairs of users in ml10M, when using 1024 and 4096 bits . The x-axis represents the real similarity of a pair of users $(u, v) \in U^2$, the y-axis represents its similarity obtained with GoldFinger, and the z-axis represents the number of pairs whose real similarity is x and estimated similarity y. (Note the log scale for z values.) The solid diagonal line is the identity function ($x = y$), while the two dashed lines demarcate the area in which points are at a distance lower than 0.1 from the diagonal. These figures were obtained by sampling 10^8 pairs of users of ml10M. Due to technical reasons we had to divide each z-value by 10. The closer points lay to the diagonal ($x = y$), the more accurate the estimation of their similarity by GoldFinger.

Figure 8 shows that the size of SHFs strongly influences the accuracy of the Jaccard estimation: while points tend to cluster around $x = y$ with 4096-bits SHFs (Figure 8a), the use of 1024 bits generate collisions that lead GoldFinger to overestimate low similarities (Figure 8b).

Still, most of the pairs of users have both their exact and approximated similarities below 0.1: 92% for 1024 bits and 94% for 4096 bits. This confirms our initial intuition (Section 3): two users with low similarity are likely to get a low approximation using GoldFinger. The large majority of the pairs of users in the KNN (as directed edges) do not see their similarity changed much by the use of SHFs. The pairs that experience a large variation between their real and their estimated similarity are too few in numbers to have a decisive impact on the quality of the resulting KNN graph.

This explains why the Brute Force algorithm experiments a decrease in execution time along with a small drop in quality with GoldFinger. Hyrec and NNDescent, however, iterate recursively on node neighborhoods and are therefore more sensitive to the overall distribution of similarity values. The recursive effect is the reason why Hyrec and NNDescent's execution time first decreases as SHFs grow in size (as mentioned earlier, in Fig. 7).

To shed more light on this effect, Figure 9 shows the number of iterations and the corresponding scan rate performed by Hyrec for SHF sizes varying from 64 to 8192 bits. The scan rate is the number of similarity computations executed by Hyrec+GoldFinger divided by the number of pairs of users. The green horizontal line represents the results when using native Hyrec. As expected, the behavior of the GoldFinger version converges to that of native Hyrec as the size of the SHFs increases. Interestingly, short SHFs ($<$ 1024 bits) cause Hyrec to require more iterations to converge, leading to a higher scan rate. When this occurs, the performance gain on individual similarity computations (Figure 6) does not compensate for this higher scan rate, explaining the counter-intuitive pattern observed in Figure 7b.

6.4 The hash function

The hash function used to compute SHFs is central to GoldFinger's behavior, as it determines the index of each item in an SHF. In particular, the hash function controls the number of collisions, and thus the distortion on the similarity. Such a distortion has an impact on the resulting KNN graph and the execution time as seen in the previous sections.

We studied the influence on ml10M of three distinct common hash functions: Jenkins' hash function [36] (which we have used in our experiments), the modulo function applied to item IDs, and SHA-512. The main influence of the hash function is on the creation time: using SHA-512 increases by more than 10 times the dataset creation time reported in Table 8 because of the high cost of hashing items. All other metrics (execution time, quality and recall) remain however similar across all three hashing methods.

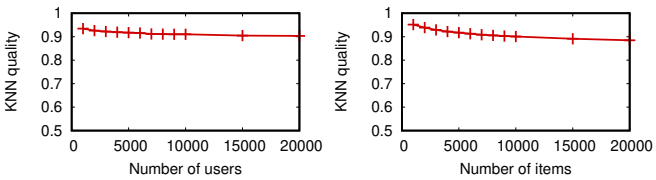
TABLE 8
Results depending on the hash function used. The hash function is critical for the creation of the dataset.

Function	Creation	Execution	Quality	Recall
Modulo	3.93s	610s	0.953	0.652
Jenkins	4.06s	606s	0.936	0.569
SHA-512	48.8s	599s	0.938	0.574

Let us note that uniform random hashing is not the only available strategy. In Section 7.1, we explore how a hash function that is tailored to a specific dataset can help increase KNN graph quality.

6.5 Impact of the dataset

To better understand how the characteristics of a dataset impact the behavior of GoldFinger, we apply GoldFinger to a series of synthetic datasets in which we modify only one parameter at a time.



(a) Impact of $|U|$ (b) Impact of $|I|$

Fig. 10. GoldFinger maintains a high quality when increasing the number of users or items. (Brute Force+GoldFinger, $b = 1024$ bits, synthetic datasets with default values $|U| = 5,000$, $|I| = 5,000$, and Zipfian distribution of users and items, with exponent 1)

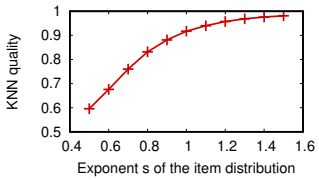


Fig. 11. GoldFinger exploits the concentration of ratings among a few items found in many datasets, shown here when increasing the exponent s of the item distribution.

6.5.1 Experimental methodology

We generate synthetic datasets by fixing a number of users $|U|$ and a number of items $|I|$. We then generate ratings by drawing user-item pairs $(u, i) \in I \times U$ according a Zipfian distribution on both items and users. We use a Zipfian exponent of 1 for users, and s for items. To avoid the cold start problem, we further add 20 ratings to every users (still drawing items according to a Zipfian distribution). In total, we draw $80 \times |U|$ ratings.

By default, we set $|U| = 5,000$, $|I| = 5,000$, and $s = 1$. Starting from this default configuration, we vary $|U|$, $|I|$ and s while keeping the other parameters at their default value.

We study the impact of the changes in the dataset on the KNN quality by using the brute force algorithm with GoldFinger and SHFs of $b = 1024$ bits. The brute force algorithm allows us to focus on the raw impact of the changes in the dataset, without any interfering mechanism such as the convergence speed of Hyrec or NNDescent that we discussed in Section 6.

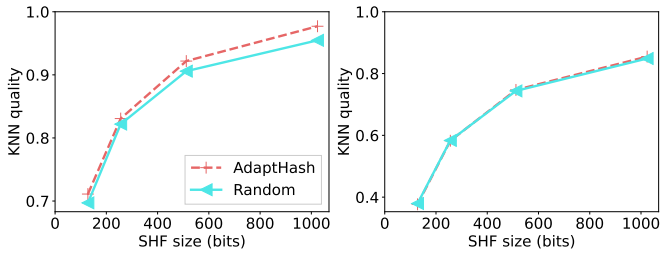
6.5.2 Items matter more than users

Figures 10a and 10b show the impact of the number of users and items on the KNN quality.

Although in both cases larger sets lower the KNN quality, the impact remains limited: increasing the number of items from 1000 to 20000 decreases the quality by 0.066, while increasing the number of users over the same range causes a drop of 0.031. The impact of items is also stronger than that of users. This difference explains why GoldFinger performs particularly well on movielens10M, while showing higher quality losses on DBLP and Gowalla.

6.5.3 Unbalanced item popularity is better

Figure 11 shows the impact of the distribution of the ratings among items on the KNN quality. GoldFinger leverages the inherently skewed distribution of many entity-item datasets, which often contain a few highly popular items and a long tail of less frequent items. With higher s values, more ratings are concentrated into a small subset of items, reducing the frequency of collisions, and helping GoldFinger achieve a higher quality.



(a) movielens10M (b) Gowalla

Fig. 12. Evolution of KNN quality with a random hash function and AdaptHash for GoldFinger using the brute force approach. AdaptHash outperforms the use of a random hash function.

7 Going Beyond

In this section, we answer the two following questions: *(i)* can we further improve the performances of GoldFinger, and *(ii)* can GoldFinger be applied to other problems? We first present a heuristic to improve the repartition of the items over the bits, and we then show that GoldFinger performs well for the KNN queries problem.

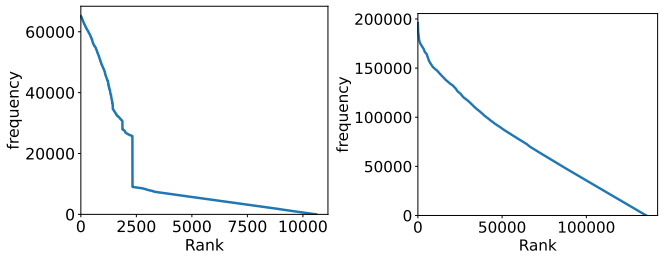
7.1 An adaptive hash function for better quality

Hashing items randomly tends to create detrimental collisions, which explains, for instance, why 2048-bit GoldFinger does not provide a perfect KNN quality. This section explores how a hash function that is tailored to a dataset can help reduce the impact of these collisions.

Intuitively, collisions occur when two items share the same hash, so decreasing the total frequency of the items associated with a hash should reduce collisions. We consider a new hashing strategy, dubbed AdaptHash, which exploits item frequencies. AdaptHash iterates over all items sequentially in order of decreasing frequency and greedily assigns to each item the hash value that has been least used so far. This strategy is a $\frac{4}{3}$-approximation [29] of the classical scheduling problem that seeks to minimize the "makespan" of all items by "scheduling" them on p processors based on their frequency.

In terms of computation, this new hashing strategy primarily impacts initialization, when the sketches are computed, as the method to estimate the similarity remains unchanged. However, the initialization time remains very small compared to the overall KNN graph computation time. For instance, on movielens10M, AdaptHash doubles the initialization time, from 4s to 8s (independently of the sketch size), but this still represents only a fraction of the computation time of the KNN graph (110s with Hyrec).

Figure 12 shows how this new hashing strategy impacts KNN graph quality on movielens10M and Gowalla. AdaptHash clearly outperforms random hash functions on movielens10M, while on Gowalla, there seems to be no discernible difference. AdaptHash delivers similar improvements to that of movielens10M on ml1M and DBLP. Its



(a) movielens10M (b) Gowalla

Fig. 13. Frequency distribution of movielens10M and Gowalla

behavior on AmazonMovies is close to that of Gowalla: it neither hurts nor improves GoldFinger's behavior.

Our investigations show that AdaptHash seems to shine when the frequency distribution is not uniform. For instance, Figure 13 shows the frequency distribution of movielens10M and Gowalla. On movielens10M, where the gain is substantial, the frequency distribution is skewed towards a long tail of low-frequency items. In contrast, the same distribution is much more uniform in Gowalla.

More generally, AdaptHash can be interpreted as a basic form of unsupervised hash learning [28], [58]. Because AdaptHash is simple, its overhead remains limited, an essential prerequisite in our context. One enticing avenue for future research is whether this strategy can be further refined to develop better yet similarly lightweight hashing techniques. One line of attack could for instance combine the simplicity of AdaptHash with the intuitions of existing unsupervised hash learning solutions.

7.2 Fast KNN queries with GoldFinger

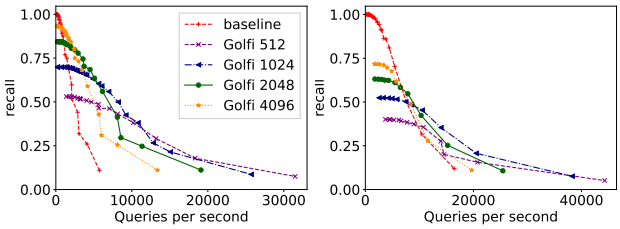
Resolving KNN queries [5], [47] is a fundamental problem related but different from the KNN graph problem. In the KNN graph problem, the k closest neighbors of every user present in a dataset must be returned. The graph can be built user by user (as LSH does) or as a whole (as Hyrec does). In that setting, the computation times of both the profiles and the KNN graph are central performance metrics. By contrast, the KNN queries problem aims to find the k closest neighbors of users (the *queries*) that are unknown beforehand and may not be part of the initial set of users. In this case, the primary metric for KNN queries is the number of queries that can be answered per second. Complex indexes and datastructures can be built offline, once and for all, to speed up the KNN queries. We evaluate the impact of the use of GoldFinger on KNN queries.

For our evaluation, we used ANN-benchmarks [6], a benchmarking tool that compares ANN querying algorithms using state-of-the-art implementations. ANN-benchmarks evaluates algorithms with parameters optimized empirically to meet different tradeoffs between quality and performance. For the Jaccard Similarity metric, ANN-benchmarks[9] compares the following implementations: NMSLIB[10] [13], PyNNDescent [24], [49], Datasketch [27] and PUFFINN [5]. In the following, we focus on the implementation of HNSW (*Hierarchical Navigable Small World graphs*) [47] from NMSLIB as it was significantly outperforming the other competitors in all our experiments, in both its base version and the version with a GoldFinger pre-treatment. The library NMSLIB was adopted by Amazon [1] and HNSW was implemented at Facebook [2]. ANN-benchmarks provides a list of good parameter configurations for the implementation HNSW in NMSLIB: this list of configurations will be used for all our experiments.

Figures 14 and 15 investigate the impact of GoldFinger on HNSW when applied to the datasets movielens10M

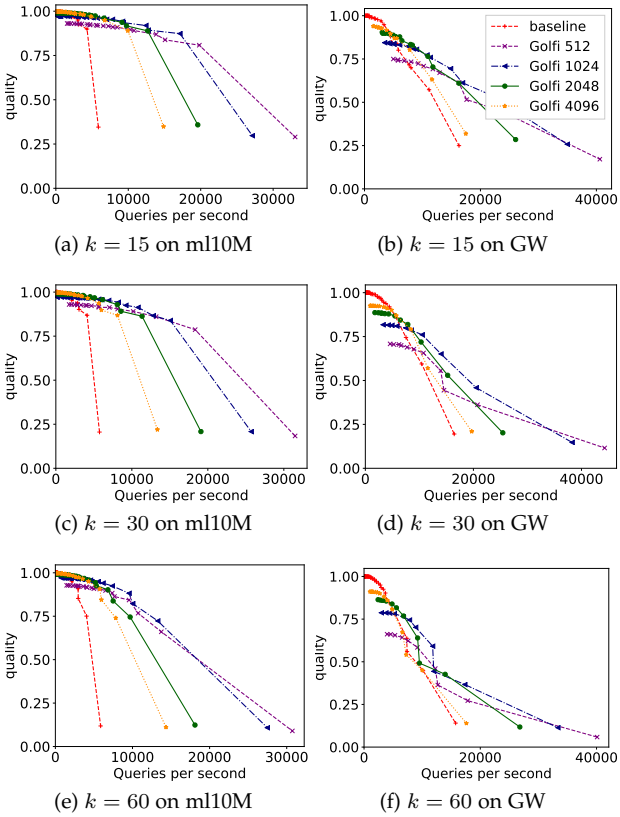
9. Our full experimental code is available at https://github.com/GuilhemN/ann-benchmarks/tree/PAPER. It leverages some improvements we contributed to the NMSLIB library.

10. Our full experimental code is available at https://github.com/GuilhemN/nmslib/tree/PAPER. It integrates GoldFinger in the NMSLIB library itself.



(a) Recall on ml10M (b) Recall on GW

Fig. 14. Impact of GoldFinger on the recall of HNSW using ANN-benchmarks, on movielens10M and Gowalla for different parameter configurations. Using GoldFinger improves throughput at the expense of recall, with GoldFinger's benefit being stronger on movielens10M, which is denser than Gowalla.

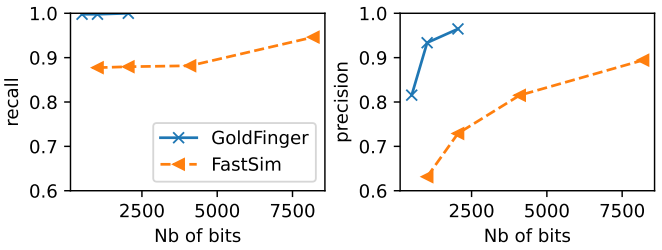


(a) $k = 15$ on ml10M (b) $k = 15$ on GW

(c) $k = 30$ on ml10M (d) $k = 30$ on GW

(e) $k = 60$ on ml10M (f) $k = 60$ on GW

Fig. 15. Impact of GoldFinger on the KNN quality of HNSW using ANN-benchmarks, on movielens10M and Gowalla, for different values of k (15, 30 and 60). Across the board, GoldFinger significantly improves throughput. The drop of quality is close to negligible on movielens10M, which is denser. On Gowalla, GoldFinger provides the better option as soon as quality drops below a reasonable threshold (0.906 for $k = 15$, 0.866 for $k = 60$), this threshold being smaller for smaller values of k.



(a) Recall (b) Precision

Fig. 16. Quality of the approximate similarity join queries with GoldFinger and Fast similarity sketching on DBLP. GoldFinger achieves a high quality with a low number of bits, outperforming Fast similarity sketching.

and Gowalla for three values of k (15, 30, and 60). The two figures chart the tradeoff between quality and execution time for KNN queries for the different parameter configurations provided for HNSW in ANN-benchmarks. Query quality is measured using recall (Figure 14) and KNN quality (Figure 15), while execution time is captured using throughput, expressed as queries per second. We plot all results obtained using the original HNSW implementation (dubbed *baseline*), and with a pre-treatment performed using GoldFinger, with SHFs sizes ranging from 512 to 4096. On movielens10M, GoldFinger delivers significant speedups for similar levels of quality. For instance, with 1024 bits and $k = 30$, GoldFinger improves throughput by 255% ($\times 3.55$ speedup) for a quality of 0.91 (Figure 15c). On Gowalla, the benefits of GoldFinger require lower quality values (graphically, the tip-off point occurs when the baseline dives under any GolFi curves), but the drop is reasonable. For instance, for $k = 15$ (Figure 15b), GoldFinger is faster as soon as the target quality is below 0.906; and at a quality of 0.75, the speedup provided by GoldFinger reaches $\times 1.70$.

For both datasets, the benefits of GoldFinger are more pronounced for smaller values of k, and decrease as k grows, yet remain significant across the board.

These experiments shows that GoldFinger is not limited to KNN graph computation but is applicable to other problems that use Jaccard's similarity, such as KNN queries.

8 Conclusion

We have proposed *GoldFinger*, a new *compact* and *fast-to-compute* representation of datasets which accelerates the computation of Jaccard's index. GoldFinger drastically speeds up the construction of KNN graphs against the native versions of prominent KNN construction algorithms such as NNDescent or LSH while incurring a small to moderate loss in quality, and close to no overhead in dataset preparation compared to the state of the art. GoldFinger is applicable to KNN queries and can help accelerate HNSW, one of the most efficient KNN query algorithms to date.

Due to its versatility, we conjecture GoldFinger can be used to solve further problems. We plan, for instance, to explore how GoldFinger can help compute *set similarity joins*. Figure 16 shows a first glimpse of how GoldFinger and Fast similarity sketching alter the quality of the results, hinting at GoldFinger's strong potential in this context.

Acknowledgments

This work was partially funded by the PAMELA project of the French National Research Agency (ANR-16-CE23-0016).

References

[1] Build k-nearest neighbor (k-nn) similarity search engine with amazon elasticsearch service. https://aws.amazon.com/about-aws/whats-new/2020/03/build-k-nearest-neighbor-similarity-search-engine-with-amazon-elasticsearch-service/. last accessed July 23, 2021.
[2] Faiss: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss. last accessed July 23, 2021.
[3] Snips: The ai platform for voice-enabled devices. https://snips.ai/. last accessed April 19, 2017.
[4] M. Alaggan, S. Gambs, and A.-M. Kermarrec. Blip: non-interactive differentially-private similarity computation on bloom filters. In *SSS*.
[5] M. Aumüller, T. Christiani, R. Pagh, and M. Vesterli. Puffinn: Parameterless and universally fast finding of nearest neighbors. In *ESA*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[6] M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.
[7] Y. Bachrach and E. Porat. Sketching for big data recommender systems using fast pseudo-random fingerprints. In *ICALP*, 2013.
[8] X. Bai, M. Bertier, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. Gossiping personalized queries. In *EDBT*, 2010.
[9] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. The gossple anonymous social network. In *Middleware*, 2010.
[10] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, 2006.
[11] A. Boutet, D. Frey, R. Guerraoui, A.-M. Kermarrec, and R. Patra. Hyrec: Leveraging browsers for scalable recommenders. In *Middleware*, 2014.
[12] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taïani. Being prepared in a sparse world: the case of knn graph construction. In *ICDE*, 2016.
[13] L. Boytsov and B. Naidan. Engineering efficient and effective non-metric space library. In N. Brisaboa, O. Pedreira, and P. Zezula, editors, *Similarity Search and Applications*, 2013.
[14] A. Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of SEQUENCES 1997*.
[15] R. Bruno, D. Patrício, J. Simão, L. Veiga, and P. Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *EuroSys*, 2019.
[16] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, 2002.
[17] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.
[18] J. Chen, H.-r. Fang, and Y. Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10(9), 2009.
[19] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *KDD*, 2011.
[20] T. Christiani, R. Pagh, and J. Sivertsen. Scalable and robust set similarity join. In *ICDE*, 2018.
[21] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.
[22] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup. Fast similarity sketching. In *FOCS*, 2017.
[23] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, et al. Alex: an updatable adaptive learned index. In *SIGMOD*, 2020.
[24] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.
[25] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *CIKM*, 2008.
[26] C. Dwork. Differential privacy: A survey of results. In *TAMC*.
[27] Z. E. Datasketch. https://github.com/ekzhu/datasketch.
[28] C. D. Erkun Yang, T. Liu, W. Liu, and D. Tao. Semantic structure-based unsupervised deep hashing. In *IJCAI*, 2018.

[29] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, Feb. 1978.

[30] G. Giakkoupis, A.-M. Kermarrec, O. Ruas, and F. Taïani. Cluster-and-conquer: When randomness meets graph locality. In *ICDE*. IEEE, 2021.

[31] M. Gorai, K. Sridharan, T. Aditya, R. Mukkamala, and S. Nukavarapu. Employing bloom filters for privacy preserving distributed collaborative knn classification. In *2011 World Congress on Information and Communication Technologies*, 2011.

[32] R. Guerraoui, A. Kermarrec, O. Ruas, and F. Taïani. Fingerprinting big data: The case of knn graph construction. In *ICDE*, April 2019.

[33] R. Guerraoui, A.-M. Kermarrec, O. Ruas, and F. Taïani. Smaller, faster & lighter knn graph constructions. In *WWW*, 2020.

[34] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TIIS*, 5(4):1–19, 2015.

[35] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.

[36] B. Jenkins. Hash functions. *Dr Dobbs Journal*, 22(9), 1997.

[37] K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. Lazybase: freshness vs. performance in information management. *ACM SIGOPS Operating Systems Review*, 44(1).

[38] A.-M. Kermarrec, O. Ruas, and F. Taïani. Nobody cares if you liked star wars: Knn graph construction on the cheap. In *European Conference on Parallel Processing*, 2018.

[39] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, 2018.

[40] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *The VLDB Journal*, 13(3).

[41] X. N. Lam, T. Vu, T. D. Le, and A. D. Duong. Addressing cold-start problem in recommendation systems. In *IMCOM*.

[42] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. Lars: A location-aware recommender system. In *ICDE*, 2012.

[43] P. Li and A. C. König. Theory and applications of b-bit minwise hashing. *Communications of the ACM*, 54(8), 2011.

[44] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[45] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2005.

[46] W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. In *IEEE CVPR*, 2012.

[47] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE TPAMI*, 42(4), 2020.

[48] J. J. McAuley and J. Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *WWW*, 2013.

[49] L. McInnes. PyNNDescent. https://github.com/lmcinnes/pynndescent.

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

[51] N. Nodarakis, S. Sioutas, D. Tsoumakos, G. Tzimas, and E. Pitoura. Rapid aknn query processing for fast classification of multidimensional data in the cloud. *CoRR*, abs/1402.7063, 2014.

[52] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW*, 1994.

[53] S. Su, C. Zhang, K. Han, and Y. Tian. Greedy hash: Towards fast optimization for accurate hash coding in cnn. *NeurIPS*, 2018.

[54] S. J. Swamidass and P. Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3), 2007.

[55] C. J. van Rijsbergen. *Information retrieval*. Butterworth, 1979.

[56] J. Wang, T. Zhang, N. Sebe, H. T. Shen, et al. A survey on learning to hash. *IEEE PAMI*, 40(4):769–790, 2017.

[57] E. Xing, M. Jordan, S. J. Russell, and A. Ng. Distance metric learning with application to clustering with side-information. *NeurIPS*, 2002.

[58] E. Yang, T. Liu, C. Deng, W. Liu, and D. Tao. Distillhash: Unsupervised deep hashing by distilling data pairs. In *CVPR*, 2019.

[59] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1), 2015.

[60] J. Yu, X. Yang, F. Gao, and D. Tao. Deep multimodal distance metric learning using click constraints for image ranking. *IEEE transactions on cybernetics*, 47(12):4014–4024, 2016.

Rachid Guerraoui Rachid Guerraoui is professor in computer science at EPFL where he leads the laboratory of Distributed Computing. He worked in the past with Ecole des Mines de Paris, CEA Saclay, HP Labs in Palo Alto and MIT. He has been elected ACM fellow and professor of the College de France. He was awarded a Senior ERC Grant and a Google Focused Award.

Anne-Marie Kermarrec Anne-Marie Kermarrec is Professor at EPFL where she leads the Scalable Computing Systems Lab. In the past, she founded and led the Mediego startup. She was a senior director at Inria (F, 2004-2015), and has worked at Vrije Universiteit (NL) and Microsoft Research Cambridge (UK). She has received an ERC grant, an ERC Proof of Concept Grant, a Montpetit Award and an Innovation Award, both from the French Academy of Science. She has been elected to the European Academy and ACM Fellow. Her research interests are large-scale distributed systems, peer-to-peer networks and system support for machine learning.

Guilhem Niot is currently a Master student at EPFL in Switzerland where he prepares a double diploma with École Normale Supérieure de Lyon. Guilhem holds a bachelor degree from École Normale Supérieure de Lyon. He is actively authoring and contributing to open source libraries for KNN querying, hardware security and web applications. His main interests lie in distributed systems, cryptography and software engineering.

Olivier Ruas is currently R&D engineer at Pathway. He was previously a postdoctoral fellow at Inria (Lille, France) and the Peking University (PKU, Beijing, China). Olivier holds a PhD from University of Rennes 1 (2018), a master degree from École Normale Supérieure de Cachan. His research interests focus on the intersection between machine learning, data management and distributed systems.

François Taïani has been a Professor in Distributed Computer Systems at the University of Rennes 1 and IRISA/Inria in Brittany, France since 2012. François holds a PhD from Université Toulouse III (2004), a Diplom der Informatik from Universität Stuttgart (1998), and a Diplôme d'Ingénieur from Ecole Centrale Paris (France). His main interest lies in the scalability and programmability of complex distributed systems.