Distributed approximate KNN Graph construction for high dimensional Data

Mohamed Riadh Trad
INRIA Paris-Rocquencourt
Riadh.Trad@inria.fr

Alexis Joly
INRIA Sophia-Antipolis
Alexis.Joly@inria.fr

Nozha Boujemaa
INRIA Saclay
Nozha.Boujemaa@inria.fr

Abstract

La construction des graphes de plus proches voisins est un problème crucial pour nombre d'applications, notamment celles impliquant des algorithmes d'apprentissage et de fouille de données. Bien qu'il existe certain travaux visant à résoudre le problème dans des environnements centralisés, ils en restent néanmoins limités en raison du volume croissant des données ainsi que leur dimensionalité. Dans cet article, nous proposons une méthode basée sur des fonctions de hachage pour la construction des graphes de plus proches voisins. La méthode proposée est distribuable et scalable, aussi bien en volume qu'en dimensionalité. Par ailleurs, l'utilisation d'une nouvelle famille de fonctions de hachage, RMMH, garantit l'équilibe des charges en environnements parallèles et distribués.

Keywords: KNN-Graph, MapReduce, Similarity Search, Scalable, Distributed, Approximate, Hashing

1 Introduction

Given a set $\mathcal{X}$ of N objects, the K-Nearest Neighbor Graph consists of the vertex set $\mathcal{X}$ and the set of edges connecting each object from $\mathcal{X}$ to its K most similar objects in $\mathcal{X}$ under a given metric or similarity measure. Efficiently constructing the K-NNG of large datasets is crucial for many applications involving feature-rich objects, such as images, text documents or other multimedia content. Examples include query suggestion in web search engines [17], collaborative filtering [1] or event detection in multimedia User Generated Contents. The K-NNG is also a key data structure for many established methods in data mining [4], machine learning [3] and manifold learning [19]. Overall, efficient K-NNG construction methods would extend a large pool of existing graph and network analysis methods to large datasets without an explicit graph structure.

In this paper we investigate the use of high dimensional hashing methods for efficiently approximating the K-NNG, notably in distributed environments. Recent works [21, 9] have shown that the performances of usual hashing-based methods are not as good as expected when constructing the full K-NNG (rather than only considering individual top-K queries). Recently, Dong et al.[9] even show that LSH and other hashing scheme can be outperformed by a radically different strategy, purely based on query expansion operations [9], without relying on any indexing structure or partitioning method. Our work provides evidence to support hashing based methods by showing that such observations might be mitigated when moving to more recent hash function families.

Our new KNN-join method is notably based on RMMH [12], a recent hash function family based on randomly trained classifiers. In this paper, we discuss the importance of balancing issues on the performance of hashing-based similarity joins and show why the baseline approach using Locality Sensitive Hashing (LSH) and collisions frequencies does not perform well (section 3). We then introduce our new K-NNG method based on RMMH (section 4). To further improve load balancing in distributed environments, we finally propose a distributed local join algorithm and describe its implementation within the MapReduce framework.

2 Related work

At first, the K-NNG problem can be seen as a nearest neighbors search problem where each data point itself is issued as a query. The brute-force approach, consisting in N exhaustive scans of the whole dataset, has the cost $O(N^2)$. Its practical usage is therefore limited to very small datasets. Building an index and iteratively processing the N items in the dataset with approximate Nearest Neighbors search techniques is an alternative option that might be more efficient. Approximate nearest-neighbor algorithms have been shown to be an interesting way of dramatically improving the search speed, and are often a necessity [20, 7]. Many approximate NN methods were then proposed including randomized kd-trees [18], hierarchical k-means [16] or approximate spill-trees [15]. Overall, one of the most popular approximate nearest neighbor search algorithms used in multimedia applications is LSH [8]. The basic method uses a family of locality-sensitive hash functions composed of linear projections over randomly selected directions in the feature space.

In addition, the usual approximate Nearest Neighbors search methods, some recent studies focus more specifically on the K-NNG construction problem as a whole. In the text retrieval community, recent studies [2, 21] focused on the ϵ-NNG construction in which one is only interested in finding pairs whose similarity exceeds a predefined threshold. In [21], the authors present a permutation based approach both to filter candidate pairs and to estimate the similarity between vectors. However, their approach is only applicable on sparse vectors. Very recently, Dong et al. [9], proposed the NN-*Descent* algorithm, an approximate K-NNG construction method purely based on query expansion operations and applicable to any similarity measure. Their experiments show that their approach is more efficient than other state-of-the-art approaches. However, designing an efficient distributed version of this method is not trivial limiting its practical scalability as it requires the entire dataset to be loaded into a centralized memory.

3 Hashing based K-NNG construction

3.1 Notations

Let us first introduce some notations. We consider a dataset $\mathcal{X}$ of N feature vectors $\mathbf{x}$ lying in a Hilbert space $\mathbb{X}$. For any two points $\mathbf{x}, \mathbf{y} \in \mathcal{X}$, we denote as $\kappa : \mathbb{X}^2 \to \mathbb{R}$ a symmetric kernel function satisfying Mercer's theorem, so that κ can be expressed as an inner product in some unknown Hilbert space through a mapping function Φ such that $\kappa(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x}).\Phi(\mathbf{y})$.

If we denote as $\mathcal{N}_K(\mathbf{x})$ the set of the K nearest neighbors of $\mathbf{x}$ in $\mathcal{X}$ according to κ, then the K-Nearest Neighbor Graph on $\mathcal{X}$ is a directed graph $\mathbf{G}_K(\mathcal{X}, E)$ connecting each element to its K Nearest Neighbors, thus: $E = \{(\mathbf{u}, \mathbf{v}), \mathbf{u} \in \mathcal{X}, \mathbf{v} \in \mathcal{N}_K(\mathbf{u})\}$.

We generally denote as $\mathcal{H}$, a family of binary hash functions $h : \mathbb{X} \to \{-1, 1\}$. If we consider hash function families based on random hyperplanes we have: $h(\mathbf{x}) = sgn(\mathbf{w}.\mathbf{x} + b)$, where $\mathbf{w} \in \mathbb{X}$ is a random variable distributed according to p_w and b is a scalar random variable distributed according to p_b. When working in the Euclidean space $\mathbb{X} = \mathbb{R}^d$ and choosing $p_w = \mathcal{N}(0, \mathbf{I})$ and $b = 0$, we get the popular LSH function family sensitive to the inner product [5, 13]. In this case, for any two points $\mathbf{q}, \mathbf{v} \in \mathbb{R}^d$ we have:

$$Pr[h(\mathbf{q}) = h(\mathbf{v})] = 1 - \frac{1}{\pi} cos^{-1}\left(\frac{\mathbf{q}.\mathbf{v}}{\|\mathbf{q}\| \, \|\mathbf{v}\|}\right)$$

Thus, the collision probability of any two items increases with their inner product ($\kappa(\mathbf{q},\mathbf{v}) = \mathbf{q}.\mathbf{v}$). More generally, any LSH function family has the property:

$$Pr[h(\mathbf{q}) = h(\mathbf{v})] = f\left(\kappa(\mathbf{q},\mathbf{v})\right) \tag{1}$$

where $f(\kappa)$ is the *sensitivity function*, increasing with κ.

3.2 LSH based K-NNG approximation

Let us consider L hash tables, each constructed from the concatenation of p hash functions built from an LSH family $\mathcal{H}$. The collision probability of any two items $\mathbf{q},\mathbf{v}$ in *one* table is:

$$Pr[\mathbf{h}(\mathbf{q}) = \mathbf{h}(\mathbf{v})] = \left[f\left(\kappa(\mathbf{q},\mathbf{v})\right)\right]^p \tag{2}$$

where $\mathbf{h}(\mathbf{q})$ and $\mathbf{h}(\mathbf{v})$ denote p-length binary hash codes. The total number of collisions in *any set* of L hash tables, denoted as $n_{\mathbf{q},\mathbf{v}}$, is a random variable distributed according to a binomial distribution with parameters L (the number of experiments) and f^p (the probability of success of each Bernouilli experiment). The *expected* number of collisions in L hash tables is therefore:

$$\bar{n}_{\mathbf{q},\mathbf{v}} = E[n_{\mathbf{q},\mathbf{v}}] = L.\left[f\left(\kappa(\mathbf{q},\mathbf{v})\right)\right]^p$$

The *empirical* number of collisions in L tables, denoted as $\hat{n}_{\mathbf{q},\mathbf{v}}$, can be seen as an estimator of this value. And since the sensitivity function f is supposed to be an increasing function with κ, it is easy to show that:

$$\kappa(\mathbf{q},\mathbf{v}_1) < \kappa(\mathbf{q},\mathbf{v}_2) \Leftrightarrow E[\hat{n}_{\mathbf{q},\mathbf{v}_1}] < E[\hat{n}_{\mathbf{q},\mathbf{v}_2}] \tag{3}$$

The top-K neighbors of any item $\mathbf{x} \in \mathcal{X}$ according to κ can therefore be approximated by the top-K items ranked according to their collision frequency with $\mathbf{x}$ (as suggested in [14]). Consequently the whole K-NNG on $\mathcal{X}$ can be approximated by simply counting the number of collisions of item pairs, without any distance computation.
More formally, we define the *hashing based approximation* of a K-NNG $\mathbf{G}_K(\mathcal{X},E)$, as a new directed graph $\hat{\mathbf{G}}_K(\mathcal{X},\hat{E})$ where $\hat{E}$ is a set of edges connecting any item $\mathbf{x}$ to its K most frequently colliding items in the L hash tables. In practice, since the number of collisions is a discrete variable, more than K items might have the same number of collisions and have to be kept in the graph produced. The hash-based approximation of a K-NNG should therefore rather be seen as a filtering step of the all-pairs graph. A brute-force refinement step can be applied on $\hat{\mathbf{G}}_K(\mathcal{X},\hat{E})$ to get a more accurate approximation during a second stage.

3.3 Balancing issues of LSH-based K-NNG

The LSH-based K-NNG approximation is very attractive in the sense that it does not require any kernel (or metric) computation. It simply requires building L hash tables and post-processing all collisions occurring in these tables. Unfortunately, balancing issues strongly affect the performance of this scheme in practice. The cost of the method is, in fact, mainly determined by the total number of collisions in all hash tables, i.e.

$$T_{\mathcal{H}}(\mathcal{X},L,p) = \sum_{l=1}^{L}\sum_{b=1}^{2^p}\frac{n_{l,b}.(n_{l,b}-1)}{2} \tag{4}$$

where $n_{l,b}$ is the number of items in the b-th bucket of the l-th table. For an ideally balanced hash function and $p \sim log_2(N)$, the cost complexity would be $O(L.N)$. But for highly unbalanced hash functions, the cost complexity tends rather to be $O(L.N^2)$ because the most filled buckets concentrate a large fraction of the whole dataset (i.e. $n_{l,b} = \alpha N$).

4 Proposed method

We now describe our K-NNG approximation method. It can be used either as a filtering step (combined with a brute-force refinement step applied afterwards), or as a direct approximation of the graph, depending on the quality of the application required. The method holds for centralized settings as well as for distributed or parallelized settings, as discussed below.

4.1 Random Maximum Margin Hashing

Rather than using classical LSH functions, our method is based on Random Maximum Margin Hashing [12], an original hash function family introduced recently and one that is suitable for any kernelized space. In addition to its nice embedding properties, the main strength of RMMH for our problem is its load balancing capabilities. The claim of this method is actually that the lack of independence between hash functions is the main issue affecting the performance of data dependent hashing methods compared to data independent ones. Indeed, the basic requirement of any hashing method is that the hash function provide a **uniform** distribution of hash values, or at least one that is as uniform as possible. Non-uniform distributions increase the overall expected number of collisions and therefore the cost of resolving them. The uniformity constraint should therefore not be relaxed too much even if we aim to maximize the collision probability of close points.

4.2 RMMH-based K-NNG approximation

Our K-NNG approximation algorithm now works in the following way:
STEP1 - Hash tables construction: For each item $\mathbf{x} \in \mathcal{X}$, we compute L p-length binary hash codes $\mathbf{h}(\mathbf{x})$ (using $L.p$ distinct RMMH functions) and insert them in L distinct hash tables.
STEP2 - Local joins: Non-empty buckets of each hash table are processed independently by a *local join* algorithm. For each non-empty bucket b, the local join algorithm generates the $n_c = \frac{n_b.(n_b-1)}{2}$ possible pairs of items computed from the n_b items contained in the bucket. Notice that this algorithm ensures that buckets can be processed separately and therefore facilitate the distribution of our method. Each emitted pair is simply coded by a pair of integer identifiers (id_i, id_j) such that $id_i < id_j$ (as a coding convention) with $0 < i < n_b - 1$ and $0 < i.n_b + j < n_c - 1$.
STEP3 - Reduction: All pairs (id_i, id_j) are mapped onto an *accumulator* in order to compute the occurrence of each pair (within the $T_{\mathcal{H}}$ emitted pairs). Notice that the occurrence matrix produced does not depend on the mapping sequence so that each pair can be inserted independently from the other ones, at any time during the algorithm. This ensures that this reduction step can be easily distributed.
STEP4 - Filtering: Once the full occurrence matrix has been computed, it is filtered in order to keep only the most similar items and compute our approximate graph $\hat{\mathbf{G}}_K(\mathcal{X}, \hat{E})$.We recall here that the hashing-based K-NNG produced by the above algorithms could still be refined by a brute-force algorithm applied on the remaining pairs.

4.3 Split local joins

Although local joins can easily be distributed, large buckets still affect the overall performance. The local join algorithm has quadratic space complexity ($n_c = \frac{n_b.(n_b-1)}{2} = O(n_b^2)$) and is therefore, likely to raise memory exceptions as well as expensive swapping phases. Moreover, we want our distributed framework to support a wide variety of hash functions, even those with lower balancing capabilities. In the following, we extend the local join algorithm to process large buckets in parallel and/or distributed architectures with guaranties on the runtime and memory occupation. In practice, if the number of collisions generated by a given local join exceeds a fixed threshold (i.e. $n_c > c_{max}$), then the local join is split into $n_s = \lceil \frac{n_c}{c_{max}} \rceil$ *sub-joins*, each being in charge of at most c_{max} collisions. Since the number of generated pairs at iteration k is $n - k$, the number of generated pairs of an s-length iteration block starting at the i^{th} iteration of the external loop is:

$$\sum_{k=0}^{s-1}(n_b - i - k) = \frac{1}{2} * s^2 + (n_b - i - \frac{1}{2}) * s \tag{5}$$

which must be less than or equal to c_{max}. The thus defined inequality has two roots of opposite signs s_1 and s_2 ($s_1 > s_2$); we require s to be equal to $\lfloor s_1 \rfloor$ as long as $s + i$ remains less than or equal to s, $n_b - i$ otherwise.

4.4 MapReduce Implementation

As explained in the previous section, all the steps of our hashing-based K-NNG approximation framework can be easily distributed. A first *MapReduce* job performs the hash tables construction **STEP1** and then, a second MapReduce job computes **STEP2** and **STEP3**. **STEP4** can be easily distributed by using the occurrence matrix line numbers as input keys to a third job.

4.4.1 Hash tables construction (STEP1)

The first MapReduce job splits the input dataset $\mathcal{X}$ into independent chunks of equal sizes to be processed in parallel. A mapper iterates over the set of its assigned objects features and computes $L.p$ hash values for each feature. Hash values are concatenated into L p-length hash codes corresponding to L bucket identifiers for the L hash tables). Each hash code is then emitted along with the table identifier (*intermediate key*) and the associated feature identifier (intermediate value). The Reduce function merges all the emitted identifiers for a particular intermediate key (*i.e.* bucket identifier within a specific table). The resulting buckets are provided as input to the second MapReduce job.

4.4.2 Occurrence matrix computation (STEP2 & 3)

The second job processes buckets separately. The map function generates all possible pairs of identifiers of the processed bucket and issues each pair (*intermediate key*), possibly with a null intermediate value. The reduce function counts the number of intermediate values for each issued pair. For efficiency concerns, map outputs are locally combined before being sent to the reducer. This requires intermediate values to store the cumulated pair occurrences. With such an optimization, the mapper issues each pair along with its initial occurrence. Combine and reduce functions simply sum the intermediate values for each issued pair.

5 Experimental setup

This section provides details about the experimental setup including datasets, performance measures and system environment. Experimental results are reported in Section 6.

5.1 Datasets & Baselines

Our method was evaluated on 2 datasets of different dimensions and sizes:
Shape: a set of 544-dimensional feature vectors extracted from 28775 3D polygonal models.
Audio: a set of 54387 192-dimensional feature vectors extracted from the DARPA TIMIT collection.

All features were L_2 normalized and compressed into 3072-dimensional binary hash codes using RMMH and LSH. These datasets were first used in [10] to evaluate the LSH method and more recently in [9] to evaluate the NN-*Descent* algorithm against the Recursive Lanczos Bisection[6] and LSH. We rely on these datasets to evaluate our method against the NN-*Descent* method [9].

5.2 Performance measures

We use *recall* and *precision* to measure the **accuracy** of our approximate KNN Graphs against the exact KNN Graphs. The exact K-NN Graphs were computed on each dataset using a brute-force exhaustive search to find the K-NN of each node. The default K is fixed to 100. The default similarity measure between feature vectors is the inner product. Note that, since all features are L_2-normalized, the inner product K-NNG is equivalent to the Euclidean distance K-NNG. The *recall* of an approximate K-NNG is computed as the number of correct Nearest Neighbors retrieved, divided by the number of edges in the exact K-NNG. Similarly, we define the *precision* of an approximate K-NNG as the number of exact Nearest Neighbors retrieved, divided by the total number of edges in the approximate K-NNG.

As for **efficiency**, we use the number of generated pairs and the Gini coefficient to study the impact of the different parameters on hash functions used. Finally, we use the Scan-Rate as an architecture and method-independent measure of the filtering capabilities of approximate KNNG construction methods. It is defined in [9] as the ratio of the number of item pairs processed by the algorithm to the total number of possible pairs (i.e $\frac{N(N-1)}{2}$).

5.3 System environment

The NN-*Descent* code is the same as in [9] and was provided by the authors. It is an openMp parallelized implementation. To allow fair comparison, we used an openMp-based centralized version of our code. It iteratively performs steps 1 to 4 (Section 4.1) and finally applies a brute-force refinement step on the remaining pairs. Experiments were conducted on an $X5675$ 3.06 *Ghz* processor server with 96 *Gbytes* of memory.

6 Experimental results

We first evaluate the impact of the hash functions used on load distributions (Section 6.1). We then evaluate the overall performance of our method in centralized settings (Section 6.2.1) and compare it against the NN-*Descent* algorithm (Section 6.2.2).

6.1 Hash functions evaluation

In Figure 1, we report the Gini coefficient for different values of M (Section 4.1). The plots show that hash tables produced by RMMH quickly converge to fair load balancing when M increases. Gini coefficients below 0.6 are, for instance, known to be a strong indicator of a fair load distribution [11]. As a proof of concept, very high values of M even provide near-perfect balancing. As we will see later, such values are not usable in practice since too much scattering of the data also degrades the quality of the generated approximate graph. The parameter M is actually aimed at tuning the compromise between hash functions independence and the generalization capabilities of the method [12]. For typical values of M greater than 15, RMMH outperforms LSH on both datasets.

Figure 2 plots the number of collisions to be processed for increasing values of L and different hash functions. The results show that the RMMH-based approach generates up to two orders of magnitude fewer collisions than

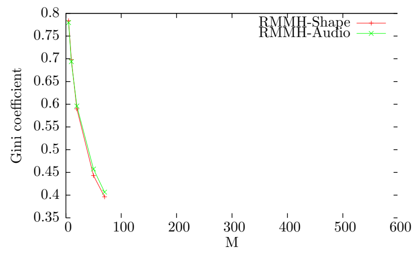


Figure 1: Gini coefficient - RMMH-Based Hashing

the LSH-based approach for values of M greater than 15. The number of generated pairs for both datasets do not exceed 10^9 pairs and therefore, can be processed in centralized settings. In the following, unless otherwise stated, the default value of M is 10.

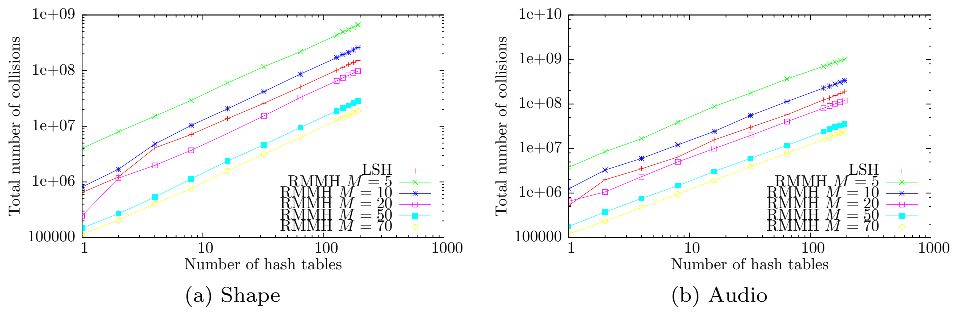


Figure 2: Total number of collisions

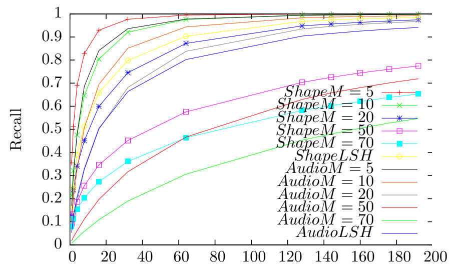
6.2 Experiments in centralized settings

In this section, we first evaluate the overall performance of our method in centralized settings (Section 6.2.1) and compare it with the NN-*Descent* algorithm (Section 6.2.2).

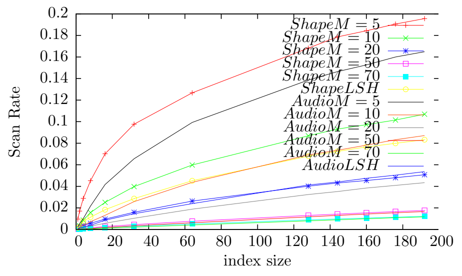
6.2.1 Overall performance

Figure 3a summarizes the recall of our method on both datasets for different hash functions and index sizes (i.e. the number L of hash tables used). The best results are observed for low values of M. The results also show high recall values even with a small number of hash tables ($L = 16$ for 90% recall) whereas higher recall values require a higher number of hash tables (128 hash tables for $M = 10$, whereas only 64 hash tables are required for $M = 10$ for 99% recall).

The results also show that high recall values can be achieved using different values of M. As discussed in 6.1, the higher M is, the fewer collisions are generated and the more hash tables are needed. Figure 3b plots the scan rate for different hash functions and index sizes. The results shows that 99% recall can be achieved while considering less than 0.05 of the total $N * (N - 1)/2$ comparisons (for $M = 20$) for both datasets. It is worth noticing that even with low values of M, and therefore low generalization properties ($M = 5$), the scan rate did not exceed several percent of the total number of comparisons. This suggests that intermediate



(a) Recall vs # of hash tables used



(b) Scan rate variation vs # of hash tables used

Figure 3: Impact of the number of hash tables used

values of M generate more accurate approximations of the KNN Graph as they require fewer comparisons for the same degree of accuracy. In practice, intermediate values of M with a high number of tables appears to be a reasonable trade-off between accuracy and approximation cost. Conversely, very high values of M degrade the accuracy of the KNNG approximation.

In Table 1, we compare RMMH against LSH in both efficiency and effectiveness. RMMH clearly outperforms LSH with fewer hash tables.

		Shape		Audio	
	L	Recall	Time	Recall	Time
RMMH	128	0.994	3.078	0.982	5.440
LSH	128	0.967	2.616	0.903	3.912
LSH	192	0.984	3.358	0.941	5.382

Table 1: Total Running Time - LSH vs RMMH ($M = 10$)

Table 2 summarizes the Recall, Scan Rate and CPU time for different values of the minimum collision frequency considered in the approximate graph. Note that threshold values cannot exceed the number of hash tables used. The results show that low collision thresholds saved more than 50% of feature comparisons on both datasets for less than a 4% recall loss.

	Shape			Audio		
thresold	Rec.	SR	CPU	Rec	SR	CPU
1	0,995	0,087	3,078	0,983	0,069	5,440
2	0,981	0,043	1,817	0,951	0,026	3,051
4	0,929	0,022	1,271	0,861	0,009	2,008

Table 2: Impact of the filtering parameter ($M = 10$, $L = 128$)

6.2.2 Comparison against State-of-the-art

We used the same NN-*Descent* settings as in [10] (($\rho = 0.5$) for speed and ($\rho = 1$) for accuracy). We use $M = 10$ and two different thresholds for the post processing phase ($t = 1$ (default) and $t = 2$) and up to 192 hash tables for high recall rates as discussed in Section 6.2.

Table 3 summarizes the recall and CPU time of both methods under those defined settings. The results show that our approach yields similar results to the NN-*Descent* algorithm for both recall rates and CPU costs using the default threshold. Although our method performs many fewer comparisons than the NN-*Descent* approach, the results show similar running times considering the Local Joins and Reduction costs. Higher threshold values are likely to further reduce the scan rate and CPU costs accordingly. By putting a threshold on the frequency collisions, our method achieves both higher recall and faster speed ($t = 1$). Actually, our frequency-based approximation beats the NN-*Descent* high-accuracy setting in all cases. Here again, the results suggest that higher threshold values achieve better approximations of the KNN Graph.

			Shape			Audio		
	ρ		Rec.	CPU	S.R.	Rec.	CPU	S.R.
NND	1		0,978	2,044	0,096	0,942	3,387	0,054
NND	0.5		0,958	2,33	0,057	0,903	4,834	0,033
	t	L	Rec.	CPU	S.R.	Rec.	CPU	S.R.
Ours	1	64	0,976	1,943	0,060	0,943	3,288	0,044
Ours	1	128	0,995	3,078	0,087	0,983	5,440	0,069
Ours	1	192	0,998	3,943	0,107	0,992	6,854	0,087
Ours	2	64	0,925	1,026	0,027	0,857	1,591	0,013
Ours	2	128	0,981	1,817	0,043	0,951	3,051	0,026
Ours	2	192	0,993	2,597	0,057	0,976	4,335	0,036

Table 3: Comparison against State-of-the-art

Table 4 shows the impact of varying K (i.e. the number of Nearest Neighbors considered in the exact K-NNG) on both datasets ($L = 128$ and $M = 10$). It shows that high recall values can be obtained on both smaller($K = 1$) and larger graphs ($K = 20$) whereas a sufficiently large K is needed for the NN-*Descent* to achieve recall rates ($> 90\%$) as stated in [9].

K	1	5	10	20	100	Scan-rate
Shape	0.999	0.998	0.997	0.994	0.978	0.086
Audio	0.996	0.991	0.988	0.982	0.957	0.069

Table 4: Recall for varying values of K

As a conclusion, our method achieves similar or better performances than the most performant state-of-the-art approximate K-NNG construction method in centralized architectures. Contrary to this method, our method has the advantage to be easily distributable and therefore more scalable.

7 Conclusion

This paper introduced a new hashing-based K-NNG approximation technique, which is easily distributable and scalable to very large datasets. To the best of our knowledge, no other work has reported full K-NN graphs results on such large datasets. Our study provides some evidence that balancing issues explain the low performances obtained with a classical LSH-based approach for approximating K-NN graphs. It also shows that using alternative new hash functions that handle hash tables uniformity can definitely change those conclusions. We finally described a distributable implementation of our method under a MapReduce framework and improved the load balancing of this scheme through a split local join strategy. Other perspectives include: multi-probe accesses to the hash tables, query expansion strategies and metadata management.

References

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17:734–749, June 2005.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[3] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1–8, 2008.

[4] M. R. Brito, E. L. Chávez, A. J. Quiroz, and J. E. Yukich. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, Aug. 1997.

[5] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[6] J. Chen, H. ren Fang, and Y. Saad. Fast approximate *k*nn graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10:1989–2012, 2009.

[7] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 244 –255, 2000.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational geometry*, pages 253–262, 2004.

[9] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[10] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 669–678, New York, NY, USA, 2008. ACM.

[11] P. Haghani, S. Michel, P. Cudré-Mauroux, and K. Aberer. Lsh at large - distributed knn search in high dimensions. In *WebDB*, 2008.

[12] A. Joly and O. Buisson. Random maximum margin hashing. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2011, Colorado Springs, CO, USA, 20-25 June 2011*, pages 873–880. IEEE, 2011.

[13] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *IEEE Int. Conf. on Computer Vision (ICCV*, 2009.

[14] K. Ling and G. Wu. Frequency based locality sensitive hashing. In *Multimedia Technology (ICMT), 2011 International Conference on*, pages 4929 –4932, july 2011.

[15] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. pages 825–832. MIT Press, 2004.

[16] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[17] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 377–386, New York, NY, USA, 2006. ACM.

[18] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*. IEEE Computer Society, 2008.

[19] S. Yan, D. Xu, B. Zhang, H.-J. Zhang, Q. Yang, and S. Lin. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:40–51, 2007.

[20] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *The VLDB Journal*, 7:275–293, December 1998.

[21] J. Zhai, Y. Lou, and J. Gehrke. Atlas: a probabilistic algorithm for high dimensional similarity search. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 997–1008, New York, NY, USA, 2011. ACM.