Experimental Analysis of Locality Sensitive Hashing Techniques for High-Dimensional Approximate Nearest Neighbor Searches

Omid Jafari[0000−0003−3422−2755] and Parth Nagarkar[0000−0001−6284−9251]

[1] New Mexico State University, Las Cruces, US
[2] {ojafari, nagarkar}@nmsu.edu

Abstract. Finding nearest neighbors in high-dimensional spaces is a fundamental operation in many multimedia retrieval applications. Exact tree-based approaches are known to suffer from the notorious *curse of dimensionality* for high-dimensional data. Approximate searching techniques sacrifice some accuracy while returning *good enough* results for faster performance. Locality Sensitive Hashing (LSH) is a popular technique for finding approximate nearest neighbors. There are two main benefits of LSH techniques: they provide theoretical guarantees on the query results, and they are highly scalable. The most dominant costs for existing external memory-based LSH techniques are algorithm time and index I/Os required to find candidate points. Existing works do not compare both of these costs in their evaluation. In this experimental survey paper, we show the impact of both these costs on the overall performance. We compare three state-of-the-art techniques on six real-world datasets, and show the importance of comparing these costs to achieve a more fair comparison.

Keywords: Locality Sensitive Hashing · High-Dimensional Spaces · Approximate Nearest Neighbor.

1 Introduction

Many large multimedia retrieval applications require efficient processing of nearest neighbor queries in high-dimensional spaces. Exact tree-based indexing structures, such as KD-tree, SR-tree, etc., work well for low-dimensional spaces ($<$ 10) but suffer from the notorious *curse of dimensionality* for high-dimensional spaces. They are often outperformed by brute-force linear scans [4]. One solution to this problem is to search for *good enough* approximate results instead. Approximate techniques sacrifice some accuracy for a significant improvement in the overall processing time. In many applications where 100% is not needed, this tradeoff is very useful in saving time. The goal of the approximate version of the nearest neighbor problem, also called *c-approximate Nearest Neighbor search*, is to return points that are within $c * R$ distance from the query point. Here, $c > 1$ is a user-defined approximation ratio and R denotes the distance of the query point and its nearest neighbor.

1.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [8] is one of the most popular techniques for finding approximate nearest neighbors in high-dimensional spaces. LSH was first introduced in [8] for the Hamming distance, but was later extended to several distances, such as the popular Euclidean distance [6]. LSH uses *random* hash projections to map the original high-dimensional space to the projected low-dimensional space. The main idea behind LSH is that nearby points in the original high-dimensional space will map to similar hash buckets in the low-dimensional space with a higher probability than dissimilar or far away points. Since LSH was first proposed in [8], there have been several works that have focused on improving the search accuracy and/or performance [3,7,9,16,18,23,15].

1.2 Motivation for using LSH

Locality Sensitive Hashing (LSH) is known for two main advantages: its sub-linear query performance (in terms of the data size) and theoretical guarantees on the query accuracy. Additionally, LSH uses random hash functions which are data-independent (i.e. data properties such as data distribution are not needed to generate these random hash functions), and the generation of these hash functions is a simple process that takes negligible time. Additionally, the data distribution does not affect the generation of these hash functions. Hence, in applications where data is changing or where newer data is coming in, these hash functions do not require any change during runtime. While the original LSH index structure suffered from large index sizes (in order to obtain a high query accuracy) [3,18], state-of-the-art LSH techniques [7,9] have alleviated this issue by using advanced methods such as *Collision Counting* and *Virtual Rehashing*. In addition to their fast index maintenance, fast query performance, and theoretical guarantees on the query accuracy, LSH algorithms are easy to implement as external memory-based algorithms, and hence are more scalable than in-memory algorithms (such as graph-based ANN algorithms) [15].

1.3 Motivation of our Experimental Survey

Locality Sensitive Hashing techniques have two dominant costs for finding nearest neighbors: 1) cost of reading the index files from the external memory to the main memory (which we call *Index I/Os*), and 2) cost of finding candidates and removing false positives (which we call *Algorithm time*). As mentioned in Section 1.2, one of the benefits of LSH is that it is a scalable algorithm. Some of the existing LSH techniques (e.g. C2LSH [7] and QALSH [9]) are not entirely external memory-based (i.e. even though the indexes are stored on the disk, their implementations require the entire data and indexes should fit into the main memory during the index creation phase). Thus, existing works (such as [1]) do not compare their results with C2LSH and QALSH on large datasets since they do not fit in the main memory. Additionally, some recent works (such as [15]) only compare the *Index I/Os* without comparing the important *Algorithm*

time. This leads to other recent papers (such as [14,13,25]) to unfairly compare their *Algorithm time* with QALSH or I-LSH [15] since they are deemed as the state-of-the-art LSH techniques.

1.4 Contributions of this Experimental Survey paper

In this paper, we carefully present a detailed experimental analysis on three state-of-the-art *external memory-based* LSH algorithms, C2LSH [7], QALSH [9], and I-LSH [15]. Our contributions are as follows:

- We modify the implementations of C2LSH and QALSH to create fully external memory-based implementations such that the entire dataset and/or the entire index do not need to be in the main memory for the algorithms to work during index generation or query processing.[3]
- We show the importance of experimentally analyzing and comparing the *Index I/Os* and *Algorithm time* of all algorithms.
- We compare these three algorithms on real datasets with different characteristics under differing system parameters.

To the best of our knowledge, we are the first work to present a detailed analysis of these three state-of-the-art LSH techniques.

2 Related Work

Nearest Neighbor problem is an important problem for multimedia applications in many diverse domains such as multimedia retrieval, image processing, machine learning, etc. Since tree-based index structures can be outperformed by a linear scan, due to the *curse of dimensionality*, in high-dimensional spaces, approximate techniques are preferred due to their fast performance at the expense of some accuracy. These techniques can be broadly classified into three main categories: Hashing-based methods, Partition-based methods, and Graph-based methods.[4] Hashing-based methods can be further classified into learning-based hashing techniques and random hashing techniques. The benefit of random hashing techniques, such as Locality Sensitive Hashing [8], are that they are easy to construct, no need for training data, and easy to maintain and update. Additionally, LSH provides a sub-linear (in terms of the data size) query performance and theoretical guarantees on the query accuracy.

Locality Sensitive Hashing and its variants: The main idea of Locality Sensitive Hashing is to create random projections and hash data points in these random projections such that nearby data points in the original high-dimensional space will be mapped to the same hash bucket with a higher probability compared to data points that are far apart from each other. It was originally proposed in [8] for the Hamming distance and then later extended to the popular

[3] These implementations will be made public.

[4] We refer the reader to a recent survey [14] for an in-depth survey on these categories.

Euclidean distance [6]. In this original work on Euclidean distance (E2LSH), instead of a single hash function (or a projection), a hash table consisted of several hash functions (represented by Compound Hash Keys) was built to reduce false positives. But this also generated false negatives. Hence several hash tables had to be used to reduce the number of false positives and false negatives, while keeping the accuracy of the query high. The main drawbacks of this approach were the size of the index structure (since large number of hash tables were required to return the desired number of results with a high accuracy) and the need to determine the width of the hash bucket during index creation (a larger width returned enough results but also with a potential of too many false positives, whereas a smaller width had a potential of misses resulting in insufficient results). This user-defined width, which was mainly dependent on the data distribution, had to be often determined through a trial and error process. LSH-Forest [3] was proposed where the compound hash-keys were hierarchically stored such that the algorithm could stop at a higher level in the tree if more results were needed. In Multi-probe LSH [18], the authors proposed a technique to probe into neighboring buckets when more results were needed. The intuition is that neighboring buckets are more likely to contain nearby points. Hence, if the bucket width was underestimated (which is better than overestimation which can lead to significant wasteful processing), neighboring buckets were probed to find the desired number of results.

Later, C2LSH [7] introduced two main concepts of *Collision Counting* and *Virtual Rehashing* that solved the two main drawbacks of E2LSH [6]. In C2LSH, the authors proposed to create m base hash functions and choose candidate points based on how many times a data point collides with the query point (and hence instead of creating several hash tables of several hash functions, only 1 table of m base hash functions is needed), which reduced the size of the index structure. Additionally, in *Virtual Rehashing*, the neighboring buckets in each hash function are read incrementally when sufficient number of results are not found. In SK-LSH [16], the authors propose a linear ordering on the Compound Hash Keys (using a space-filling curve) such that nearby Compound Hash Keys are stored on the same (or nearby) page on the disk, thus reducing the total number of I/Os. The design of SK-LSH is still build on the original E2LSH, and hence suffers from the parameter tuning problem, where the user is expected to enter important parameters such as number of hash functions and the radius at which k results will be found. QALSH [9] was later proposed that built query-aware hash functions such that the hash value of the query point is considered as the anchor bucket during query processing and this idea would solve the issue when close points to a query were partitioned into different buckets when query was near the bucket boundaries. Additionally, B+trees are built on each hash function for efficient lookups into neighboring buckets (which translate to range queries). QALSH utilizes the concepts of *Collision Counting* and *Virtual Rehashing*. HD-Index [1] was introduced which generated Hilbert keys of the dataset points and also stored the distances of points to each other to efficiently prune the results based on distance filters. Due to the reliance on space-filling

curves (Hilbert curves) and B+-trees, HD-Index cannot scale for moderately high-dimensional datasets [1]. SRS [22] uses the Euclidean distance between two points in the projected space to estimate their distance in the original space. In order to find the next nearest neighbor in the projected space, SRS uses an R-tree to index the points in the projected space. This incremental finding of the NN is similar to I-LSH. The main goal of SRS is to introduce a very lightweight index structure to solve the ANN problem. SRS is shown to suffer from memory leaks and slow running times as compared with C2LSH [1], and hence not included in our work. Recently, I-LSH [15], which is considered to be the state-of-the-art LSH technique [13], was proposed to improve the Virtual Rehashing process of QALSH (where the range of the lookups are incremented exponentially). In I-LSH, the authors propose to increase the range of the lookups based on the distance to the nearest point (in the projected space) instead of increasing the range exponentially. While this strategy results in less disk I/Os, it also leads to high disk seeks (random I/Os) and algorithm time as we show in Section 4. Very recently, an in-memory LSH algorithm, PM-LSH [25] was proposed where the idea was to estimate the Euclidean distance based on a tunable confidence interval value such that the overall query processing time is reduced.

3 State-of-the-art Techniques

In this section, we will introduce the concepts introduced by the three state-of-the-art external memory-based LSH techniques, C2LSH [7], QALSH [9], and I-LSH [15]. We primarily use the terminologies and formulations introduced in E2LSH [6] and C2LSH [7]. Due to space limitations, we ask the reader to refer to [7] for detailed formulations. C2LSH [7] introduced the concepts of *Collision Counting* and *Virtual Rehashing*. In [7], authors theoretically show that two close points x and y collide in at least l hash layers (out of m hash layers) with a probability $1-\delta$. Further, only those points that collide at least l times with the query point, where l is the collision count threshold, are chosen as candidates. C2LSH creates only one hash function per hash table, and hence the number of hash functions are equal to the number of hash table.

Instead of assuming a *magic* radius (which traditional LSH methods did), C2LSH sets the initial radius R to 1. It is possible that with $R = 1$, there are not enough results for a top-k query to be returned. C2LSH increases the radius of the query in the following sequence: $R = 1, c, c^2, c^3$.... If at *level-R*, enough candidates are not found, the radius is increased until enough query results are found. This exponential expansion process is called *Virtual Rehashing*.

Moreover, C2LSH uses two terminating conditions to stop the algorithm. These conditions specify that 1) at the end of each virtual rehashing at least k candidates should have been found whose Euclidean distance to the query are less than or equal to cR, and 2) at any point, $k + \beta n$ candidates are found.

QALSH introduces *query-aware* hash functions. For a query q, once the query projection is found by computing $h_{\boldsymbol{a}}(q)$, QALSH uses the query as the "anchor" to find the anchor bucket with width w with the interval $|h_{\boldsymbol{a}}(q) - \frac{w}{2}, h_{\boldsymbol{a}}(q) + \frac{w}{2}|$.

If the projected location for a point x falls in the same anchor bucket as q, i.e., $|h_{\boldsymbol{a}}(o) - h_{\boldsymbol{a}}(q)| \leq \frac{w}{2}$, then QALSH considers that o has collided with q under $h_{\boldsymbol{a}}$. QALSH [9] also utilizes these concepts of Collision Counting and Virtual Rehashing to build *query-aware* hash functions. Another main difference of QALSH is that it uses B+-trees to represent the hash tables. An exponential expansion in each hash table is thus the same as a range query on a B+-tree. By using *query-aware* hash functions and B+-trees, QALSH improves the theoretical bounds by reducing the total number of hash functions required to satisfy the quality guarantee. Additionally, QALSH can work for any approximation ratio, c, greater than 1, while C2LSH can only work for $c \geq 2$. While the reduction in number of hash functions generates a smaller index, the overhead of using B+-trees makes QALSH much slower as we experimentally show in Section 4.

I-LSH [15] uses query-aware hash functions (proposed by QALSH) and proposes an incremental expansion strategy to reduce overall index I/Os. In order to do that, I-LSH finds the next closest point in each projection. While this process leads to less overall index I/Os, it still requires disk seeks and (as we show in Section 4) the algorithm overhead is far more than the savings in the disk I/Os.

4 Experimental Analysis

In this section, we first explain our experimental evaluation plan. We experimentally analyze C2LSH, QALSH, and I-LSH on different datasets and report the results for varying criteria. All experiments were run on the nodes of the Bigdat cluster [5] with the following specifications: two Intel Xeon E5-2695, 256GB RAM, and CentOS 6.5 operating system. All codes were written in C++11 and compiled with gcc v4.7.2 with the -O3 optimization flag. As mentioned in Section 1.4, we extend the implementations of C2LSH and QALSH to be completely external-memory based implementations (i.e. the entire dataset or the index files are not needed to be in the main memory in order to construct the LSH indexes).

4.1 Datasets

We use the following six diverse high-dimensional datasets in our experiments:

- **P53**[5] consists of $31,002$ 5409-dimensional points which are generated based on the biophysical features of mutant p53 proteins.
- **LabelMe**[19] consists of $181,093$ 512-dimensional points which were generated by running the GIST feature extraction algorithm on annotated images.
- **Sift1M**[10] consists of $1,000,000$ 128-dimensional points that were created by running the SIFT feature extraction algorithm on real images.
- **Deep1M** consists of $1,000,000$ 96-dimensional points sampled from the Deep1B dataset introduced in [2].
- **Mnist8M**[17] This dataset contains $8,100,000$ 784-dimensional points that represent images of the digits 0 to 9 which are grayscale and of size 28×28.
- **Tiny80M**[24] This dataset contains $79,302,017$ 384-dimensional points generated using Gist feature extraction algorithm on 80 million colored images.

[5] Supported by NSF Award #1337884

4.2 Evaluation Criteria and Parameters

The goal of our paper is to present a detailed analysis of the performance and accuracy of the state-of-the-art LSH techniques. We randomly choose 50 queries and report the average of the results. We used the same parameters suggested in their papers ($w = 2.781$ for QALSH and $w = 2.184$ for C2LSH). We choose $\delta = 0.1$ and $c = 2$ (since C2LSH cannot give guarantees for $c < 2$). Since I-LSH uses the same hash functions as QALSH, their index size and index construction time are the same. [9] shows the difference between these two criteria for C2LSH and QALSH for different datasets, and hence we avoid it in this paper.

After careful analysis of performance of LSH techniques, we present the following breakdown of the query processing time (QPT):

- **Index Read Cost:** LSH techniques need to read index files (from the external memory) in order to find the candidates. This dominant cost of reading index files can be further broken down into the number of disk seeks (i.e. random I/Os) and the total amount of data read. Following [15], we also consider the number of disk seeks and amount read in our cost formulation.
- **Algorithm Time:** Another dominant cost in LSH processing is the processing of index files once they are read into the main memory. LSH techniques need to find points that are considered as candidates. Techniques such as Collision Counting (explained in 3) are included in this cost.
- **False Positive Removal Cost:** Once a point is deemed as a candidate, its Euclidean distance with the query point is calculated. Since the state-of-the-art LSH techniques have an upper bound of the number of candidates (which is set to $k + 100$), this cost is negligible as compared to the previous two costs. Due to space limitations and since we observed that this cost is less than 0.5 ms for all algorithms, we do not show the results of this cost.

It is well-known that random I/Os are much more expensive than sequential I/Os [12]. Additionally, the difference in the cost changes significantly depending on whether the external storage medium is an HDD or an SSD. The difference in the costs of random I/Os and sequential I/Os is significantly more in HDDs than in SSDs (mainly because random disk seeks are faster in SSDs than HDDs) [11]. We noticed that the number of disk seeks are significantly different in these LSH techniques due to how they find neighboring points in projected spaces. Hence, we model the overall *Query Processing Time* (QPT) for both HDDs and SSDs. For an HDD, we use the reported benchmarks for Seagate Barracuda HDD with 7200 RPM and 1TB: average disk seek requires 8.5 ms and the average data read rate is 0.156 MB/ms [21]. Similarly, for an SSD, we use the reported benchmarks for the Seagate Barracuda 120 SSD with 1TB storage: average disk seek requires 0.01 ms and the average data read rate is 0.56 MB/ms [20].

We use the same accuracy measure, the overall ratio, used in several prior works [7,9,16,15]: $\frac{1}{k}\sum_{i=1}^{k}\frac{||o_i,q||}{||o_i^*,q||}$. Here, o_i is the ith point returned by the technique and o_i^* is the true ith nearest point from q (ground truth). The closer the ratio is to 1, the higher is the accuracy of the LSH technique.

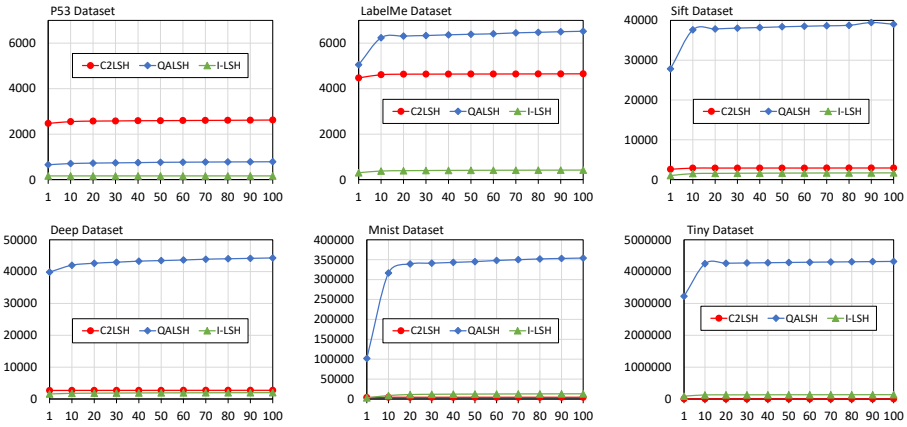

Fig. 1: Number of Disk Seeks (Y axis) for different k (X Axis) on 6 datasets

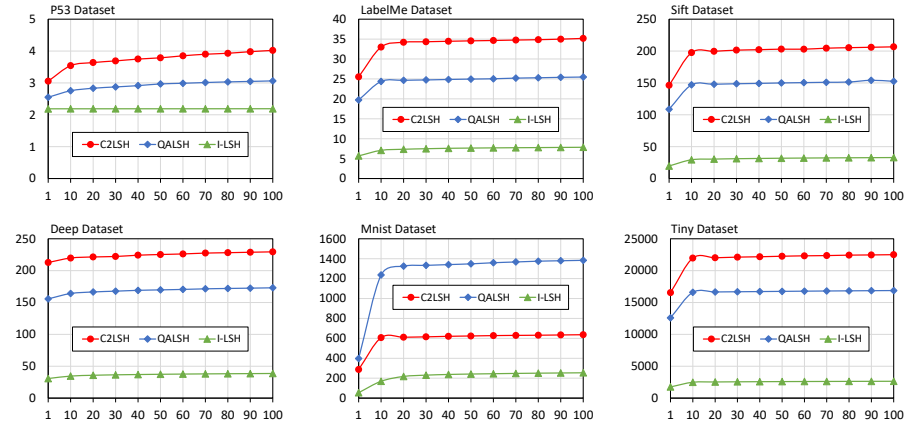

Fig. 2: Amount of Data Read (in MB) (Y axis) for k (X Axis) on 6 datasets

4.3 Discussion of the Performance Results

Number of Disk Seeks: Figure 1 shows the required number of disk seeks (random I/Os). We observed that the performance of I-LSH degrades as dataset size becomes large. This is because I-LSH needs to find the closest projected point each time the radius needs to be expanded, which further requires reading the indexed points from the disk several times. We also observe that QALSH has a better performance compared to C2LSH for smaller datasets, but as the dataset size increases, the number of seeks are significantly higher than C2LSH and I-LSH. This is happening because the search radiuses of QALSH are larger than C2LSH in larger datasets, which results in higher disk seeks.

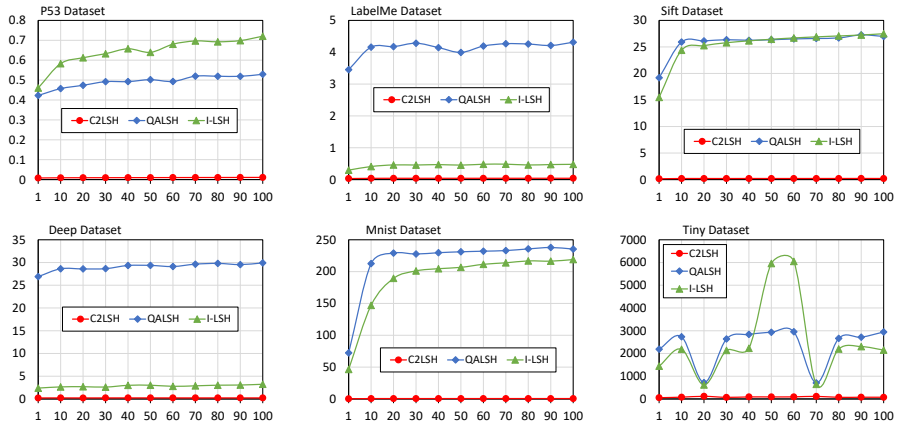

Fig. 3: Algorithm Time (in s) (Y axis) for k (X Axis) on 6 datasets

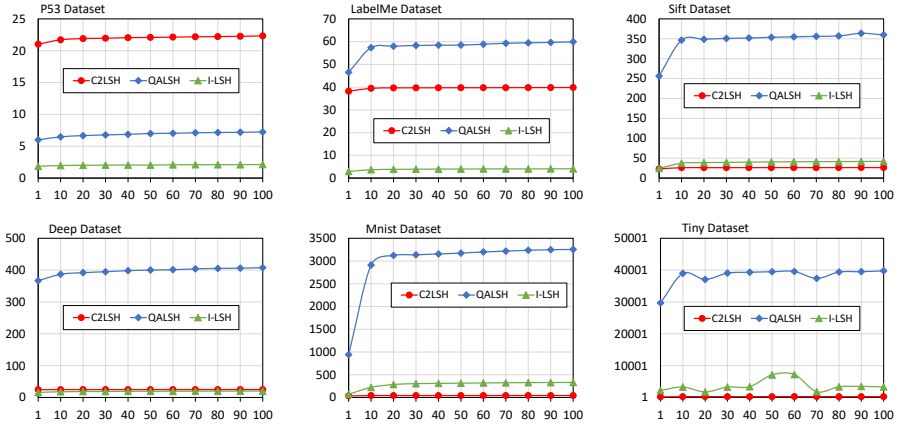

Fig. 4: HDD Query Processing Time (in s) (Y axis) for k (X Axis) on 6 datasets

Amount of Data Read: Figure 2 shows the total amount of data that was read from the index files. I-LSH always has the least amount of data read for all datasets because it incrementally searches for the nearest points in the projections instead of having buckets and fixed widths. However, we later show that these I/O savings are offset by the processing time of finding these nearest points. C2LSH reads more data than QALSH for most datasets (except Mnist) since QALSH uses less hash projections because they are query-aware.

Algorithm Time: Figure 3 shows the time needed to find the candidates (excluding the I/O times). This figure shows the huge overhead of I-LSH which is caused due to their incremental searching strategy. Also, since I-LSH and QALSH both use B+-trees, which become huge for the larger datasets, their

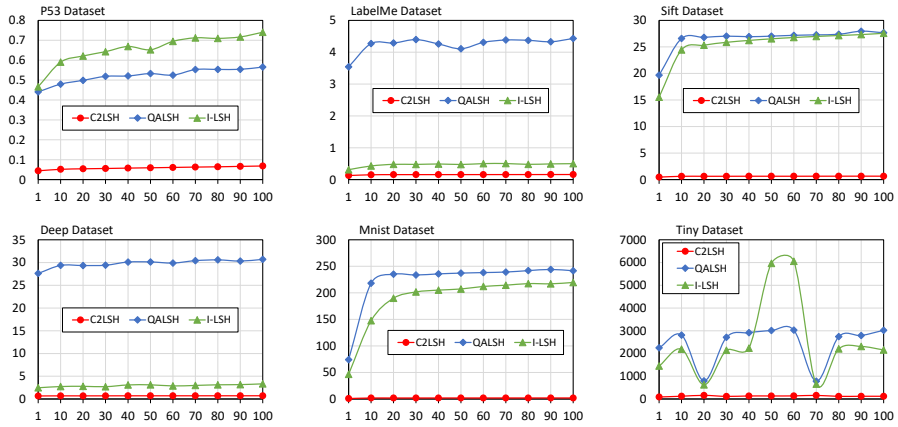

Fig. 5: SSD Query Processing Time (in s) (Y axis) for k (X Axis) on 6 datasets

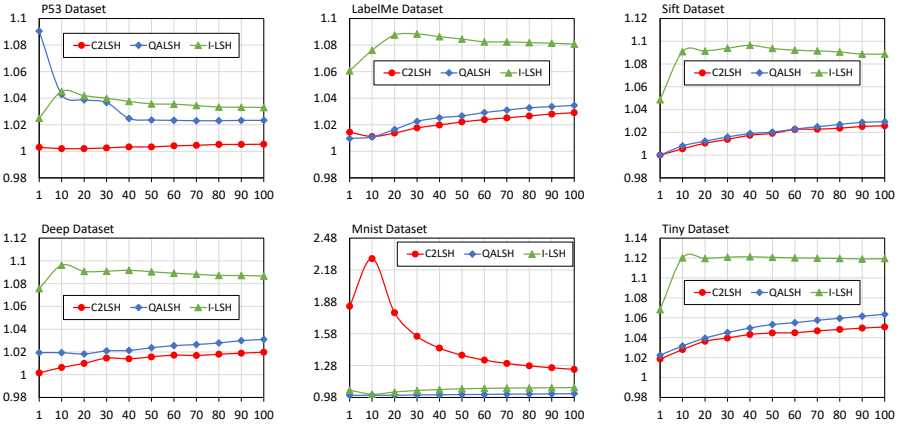

Fig. 6: Accuracy Ratio (Y axis) for different k (X Axis) on 6 datasets

performance degrades heavily in these cases. Since C2LSH does not have any overhead of additional index structures (such as B+-tree), it has the least Algorithm time for all datasets. In terms of Algorithm Time, I-LSH is faster than QALSH (except for the P53 dataset - which is the smallest dataset in our experiments) mainly because it has to process less hash functions than QALSH [15].

Query Processing Time (on HDD): Figure 4 shows the overall time required to solve a given k-NN query on a Hard Disk Drive. I-LSH performs the best for smaller datasets (P53 and LabelMe) because its Algorithm Time overhead is small, but as the dataset size increases, the Algorithm Time overhead offsets the savings in disk seeks and performs worse than C2LSH (but better

than QALSH). Except for the smallest dataset (P53), QALSH is the slowest of the three algorithms. It works good for smaller datasets (P53) but does not scale well for moderate and large sized datasets. For larger datasets, C2LSH is always the fastest technique since its having better algorithm time and number of disk seeks compared to the other two algorithms.

Query Processing Time (on SSD): Figure 5 shows the overall time required to solve a given k-NN query on a Solid State Drive. In SSDs, I/O operations are much faster and the overall Query Processing Time is mainly dominated by the algorithm time. Therefore, C2LSH (which has the best Algorithm time) always performs the best on SSDs (for all datasets) followed by I-LSH (except for the smallest dataset, P53).

Accuracy Ratio: Figure 6 shows the accuracy of the compared techniques. Ratios are always greater than or equal to 1 and having a ratio equal to 1 equates to the highest accuracy. Except for the Mnist dataset, C2LSH produces the best accuracy among the three algorithms. QALSH is more accurate than I-LSH, which we believe is mainly because it uses more hash functions than I-LSH. Except for C2LSH's accuracy on the Mnist dataset, all three algorithms produce accurate results for all datasets.

Overall, we find that C2LSH can find k-NN results faster than QALSH and I-LSH, mainly because of the simplicity of their hash functions (i.e. an additional index structure, B+-tree, is not used). Additionally, all three algorithms produce accurate results (with C2LSH producing slightly better accurate results than QALSH and I-LSH for most datasets).

5 Conclusion

Locality Sensitive Hashing is a popular technique for solving Approximate Nearest Neighbor queries in high-dimensional spaces. In this paper, we presented a detailed experimental analysis on three popular state-of-the-art LSH algorithms, C2LSH, QALSH, and I-LSH. We presented our analysis on diverse datasets with varying characteristics (cardinality and dimensionality). We show that while reducing disk seeks is important, it cannot be at the expense of Algorithm time, which can be a dominant cost in the overall query processing time for large datasets. We importantly show that improvements in one aspect of the LSH workflow (e.g. disk seeks), does not necessarily result in overall query processing performance improvement.

References

1. Arora, A., et al., "Hd-index: Pushing the scalability-accuracy boundary for approximate knn search," *VLDB* 2018.

2. Babenko, A., et al., "Efficient indexing of billion-scale datasets of deep descriptors," *CVPR* 2016.
3. Bawa, M., et al., "Lsh forest: Self-tuning indexes for similarity search," *WWW* 2005.
4. Chávez, E., et al., "Searching in metric spaces," *CSUR* 2001.
5. Danziger, S.A., et al., "Predicting positive p53 cancer rescue regions using most informative positive (mip) active learning," *PLoS computational biology* 2009.
6. Datar, M., et al., "Locality-sensitive hashing scheme based on p-stable distributions," *SOCG* 2004.
7. Gan, J., et al., "Locality-sensitive hashing scheme based on dynamic collision counting," *SIGMOD* 2012.
8. Gionis, A., et al., "Similarity search in high dimensions via hashing," *VLDB* 1999.
9. Huang, Q., et al., "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *VLDB* 2015.
10. Jegou, H., et al., "Product quantization for nearest neighbor search," *TPAMI* 2010.
11. Kim, A., et al., "Optimally leveraging density and locality for exploratory browsing and sampling," *HILDA* 2018.
12. Leis, V., et al., "Query optimization through the looking glass, and what we found running the join order benchmark," *VLDB* 2018.
13. Li, M., et al., "I/o efficient approximate nearest neighbour search based on learned functions," *ICDE* 2020.
14. Li, W., et al., "Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement," *TKDE* 2019.
15. Liu, W., et al., "I-lsh: I/o efficient c-approximate nearest neighbor search in high-dimensional space," *ICDE* 2019.
16. Liu, Y., et al., "Sk-lsh: An efficient index structure for approximate nearest neighbor search," *VLDB* 2014.
17. Loosli, G., et al., "Training invariant support vector machines using selective sampling," *Large scale kernel machines* 2007.
18. Lv, Q., et al., "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," *VLDB* 2007.
19. Russell, B.C., et al., "Labelme: a database and web-based tool for image annotation.," *IJCV* 2008.
20. Seagate Barracuda 120 SSD Manual.: `https://www.seagate.com/www-content/datasheets/pdfs/barracuda-120-sata-DS2022-1-1909US-en_US.pdf`
21. Seagate ST2000DM001 Manual.: `https://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf`
22. Sun, Y., et al., "Srs: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *VLDB* 2014.
23. Tao, Y., et al., "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *TODS* 2010.
24. Torralba, A., et al., "80 million tiny images: A large data set for nonparametric object and scene recognition," *TPAMI* 2008.
25. Zheng, B., et al., "Pm-lsh: A fast and accurate lsh framework for high-dimensional approximate nn search," *VLDB* 2020.