qwLSH: Cache-conscious Indexing for Processing Similarity Search Query Workloads in High-Dimensional Spaces*

Omid Jafari
ojafari@nmsu.edu
New Mexico State University
Las Cruces, New Mexico

John Ossorgin
osso09@nmsu.edu
New Mexico State University
Las Cruces, New Mexico

Parth Nagarkar
nagarkar@nmsu.edu
New Mexico State University
Las Cruces, New Mexico

ABSTRACT

Similarity search queries in high-dimensional spaces are an important type of queries in many domains such as image processing, machine learning, etc. Locality Sensitive Hashing (LSH) has been shown to be an effective approximate search method for solving similarity search queries in high-dimensional spaces. Often, queries in real-world settings arrive as part of a query workload. LSH and its variants are particularly designed to solve single queries effectively. They suffer from one major drawback while executing query workloads: they do not take into consideration important data characteristics for effective cache utilization while designing the index structures. In this paper, we present *qwLSH*, an index structure that intelligently divides a given cache during processing of a query workload by using novel cost models. Experimental results show that, given a query workload, *qwLSH* is able to perform faster than existing techniques due to its unique cost models and strategies to reduce cache misses.

CCS CONCEPTS

• Information systems → Nearest-neighbor search.

KEYWORDS

Nearest-neighbor Search, Locality Sensitive Hashing, High-dimensional Search

ACM Reference Format:

Omid Jafari, John Ossorgin, and Parth Nagarkar. 2019. qwLSH: Cache-conscious Indexing for Processing Similarity Search Query Workloads in High-Dimensional Spaces. In *International Conference on Multimedia Retrieval (ICMR '19)*, June 10–13, 2019, Ottawa, ON, Canada. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3323873.3325048>

1 INTRODUCTION

The similarity search problem in high-dimensional spaces is a well-known problem with wide-ranging applications in domains such as information retrieval, artificial intelligence, machine learning, etc. Exact tree-based spatial indexing techniques, such as KD-tree,

R-tree, SR-tree, etc., are effective for improving searching in low-dimensional spaces, but as the dimensions increase (~ 10), they all suffer from the popular *curse of dimensionality* problem (where they are often outperformed even by a brute force linear scan) [15]. One approach to addressing this *curse of dimensionality* problem is to search for *approximate* solutions instead of exact solutions. Approximate solutions sacrifice accuracy for a much faster performance. Formally, the goal of the approximate version of the similarity search problem, also called *c-approximate Nearest Neighbor search* [21], is to return the objects that are within the distance $c \times R$ from the query object, where $c > 1$ is an approximate ratio and R is the distance from the query to the true nearest neighbor.

Locality Sensitive Hashing: Locality Sensitive Hashing, first proposed in [21], is one of the most popular solutions for approximate searching in high-dimensional spaces. The purpose of Locality Sensitive Hashing (LSH) is to map high-dimensional data to lower dimensions while preserving the distances in the original space. The lower dimensional space is generated through a series of *random projections*. In this lower dimensional space, data objects are mapped to individual buckets based on a hash function, with the intuition that nearby data points in the original space are mapped to the same hash buckets in the lower dimensional space with a higher probability than mapping dissimilar or far away points to the same buckets, which may lead to *misses* and *false positives*.

Motivation: Often, in real-world settings, queries arrive as part of a query workload. Several research works have shown the benefits of designing index structures particularly for efficient handling of query workloads [8, 14, 24, 25, 27, 35]. Most of these works focus on partitioning the data space such that regions that are queried with high frequency are partitioned with more granularity. e.g., in 2D geographical spaces, points of interest in downtown areas are queried with much more frequency than places in the suburbs. Similarly, consider the following two scenarios for high-dimensional data: (1) Genomic DNA data, often represented as high-dimensional vectors, require similarity search queries to find similar genomes given a query genome [11, 13, 29]. Certain common genome sequences are often queried more times than the rest [29]. (2) Similarly, earthquake detection or satellite image data, both high-dimensional data [12, 37], often query certain similar regions of the space that are of more interest. These queries can be viewed as part of a query workload. While LSH has been shown to be effective in processing single queries in high-dimensional spaces, it is not particularly designed to handle query workloads efficiently. Assuming that the index and the data are stored on an external storage, Locality Sensitive Hashing and its variants have two main IO operations: (1) accessing the index (denoted by *IndexIO*) in order to find the candidates, and (2) once the candidates are found, the

*This work is supported by NSF #1633330

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICMR '19, June 10–13, 2019, Ottawa, ON, Canada

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6765-3/19/06...\$15.00

<https://doi.org/10.1145/3323873.3325048>

Figure 1: Effect of Cardinality and Dimensionality over the ratio IndexIO/TotalIO and DataIO/TotalIO respectively

candidate points need to be brought from the external storage (denoted by *DataIO*) into the main memory to remove false positives. We observe that these two operations, however, have different costs:

- the *IndexIO* cost is dependent on the size of the index, which is in turn dependent on the size of the dataset (also referred to as *cardinality* of the dataset), and
- the *dataIO* cost is dependent on the size of each data point, i.e. the number of dimensions of the dataset, and the desired number of objects in the result (top-*k*).

Figure 1 shows our observation that as the cardinality of the dataset increases (when the dimensionality is fixed), the ratio of *IndexIO/TotalIO* (where *TotalIO* = *IndexIO* + *DataIO*) increases, because the cost of *IndexIO* increases. Similarly, as the dimensionality increases, the ratio of *DataIO/TotalIO* increases, because the cost of *DataIO* increases. By using this important observation, the efficiency of an LSH algorithm can be improved by intelligently utilizing the cache (and thereby reducing cache misses) for a workload.

Contributions of this paper: In this paper, we present our proposed index structure, *qwLSH*, that can efficiently process similarity search query workloads in high-dimensional spaces. The following are the contributions of this paper:

- Given the important observation that different data have different query processing needs, we intelligently divide a cache for efficient processing of a workload to reduce overall cache misses
- We present novel cost models for intelligently dividing a cache given a query workload
- We present different cache utilization strategies for LSH-based query workloads
- Finally, we use real datasets to show the efficiency of our proposed index structure, *qwLSH*, by comparing against state-of-the-art algorithms.

2 RELATED WORK

Locality Sensitive Hashing [21] is a popular approximate technique for solving similarity search queries in high-dimensional spaces. Several works were subsequently proposed [10, 15–17, 20, 22, 23, 32, 34, 39] to improve upon the original work. Please see the survey paper [36] for details about these algorithms. [16] and [22] are considered as the state-of-the-art LSH-based algorithms. Our proposed index structure, *qwLSH*, is orthogonal to these approaches and can work with both these algorithms.

Query Workloads: Many index structures and systems have been particularly designed to execute query workloads efficiently [8, 14, 24, 25, 27, 35] for 1D [14, 24, 26], 2D and low-dimensional data [6–8, 25, 35]. [18] deals with recommending indexes in a high-dimensional database based on a given dynamic query workload.

Cache-Conscious Querying: There have been several works that design techniques to reduce cache misses to improve the query performance [19, 28, 31]. An experimental survey on cache conscious algorithms can be found at [38]. In the domain of LSH and its variants, [33] attempts to make LSH cache-conscious, but their main focus is on presenting a caching strategy to improve the index construction cost, whereas our main goal is to speed up the processing speed of the entire query workload.

3 PROBLEM SPECIFICATION

In this paper, we will be relying mainly on the terminologies presented in E2LSH [3], C2LSH [16], and QALSH [17]. Let us consider we are given a query workload Q . For each query $q \in Q$, given k and an approximation ratio $c > 1$, we return the top- k results such that $\|o_i, q\| \leq c \times \|o_i^*, q\|$, where o_i^* is the true i th nearest neighbor of q . Given a cache C , our goal is to divide the cache into two parts: C_I and C_D , where C_I is the part of the cache that stores the index files (i.e. data from the hash buckets) and C_D is the part of the cache that stores the data objects, such that $size(C) = size(C_I) + size(C_D)$. Assume $cost(C)$ denotes the cost of bringing all necessary files (index files + data objects) into the cache C . Similarly, $cost(C_I)$ denotes the cost of bringing the index files into the cache, and $cost(C_D)$ denotes the cost of bringing the data objects into the cache, such that $cost(C) = cost(C_I) + cost(C_D)$.

cost(C_I): The amount of data that is read from the index files is a function of the cardinality of the dataset (we denote this constant as α_{card}) (Figure 1). For all m projections in the LSH index, the cost of bringing all necessary index files into C_I for a single query q : $cost(C_I) = \alpha_{card} \times \sum_{i=1}^m (cost(C_{I_i}^q))$

cost(C_D): Similarly, let us first consider the cost of bringing all necessary data objects into C_D for a single query, q . For a single query, the worst case scenario is reading $w = k + v$ data objects from the external storage (where v is the number of allowed false positives [16]). Let us denote the cost of reading a single j th data object as $cost(C_{D_j}^q)$. The size of each data object is a function of the dimensionality of the dataset (Figure 1). Let us denote this constant as α_{dim} . Thus, we have: $cost(C_D) = \alpha_{dim} \times \sum_{j=1}^w (cost(C_{D_j}^q))$

When we have more than 1 query in the given query workload Q , assuming that there will be at least 1 index file that will be reused within the queries, $cost(C_I)$ will depend on the size of C_I . If a large amount of C is dedicated to C_I , then the $cost(C_I)$ will be less and vice-versa. Conversely, $cost(C_D)$ will depend on the size of C_D . Thus, given a query workload Q , we want to find $size(C_I)$ (or $size(C_D)$) such that:

$$\begin{aligned} & \text{minimize} && \sum_{g=1}^{|Q|} \left(\alpha_{card} \times \sum_{i=1}^m (cost(C_{I_i}^g)) + \alpha_{dim} \times \sum_{j=1}^w (cost(C_{D_j}^g)) \right) \\ & \text{subject to} && size(C) = size(C_I) + size(C_D). \end{aligned}$$

4 QWLSH

We intelligently divide the cache based on the cardinality and dimensionality of the given dataset by leveraging the cost models presented in Section 3. In this section, we will present two strategies for dividing the cache, along with unique cost models.

Figure 2: (a) Optimal Setting for Index Cache Size (in % of the Total Available Cache Size) on different versions of the Deepsat dataset [9], (b) Effect of Index Cache Size Setting on the IndexIO, DataIO, and TotalIO for P53 dataset, (c) Effect of varying top-k on P53 dataset. [16MB Cache Size, $|Q|=250$], (d) Effect of caching overhead on Time (in ms)

Naive Solution: For a given query workload, the cache (using the default cache replacement policy, such as MRU) treats the index files and the data object files equal, regardless of the cardinality and the dimensionality of the dataset. The cardinality and the dimensionality of the dataset affects the total size of index files and data objects that are read into the cache respectively. We leverage this observation that naive solutions do not.

Design of qwLSH: Once we determine the appropriate split between C_I and C_D , we use the popular cache replacement policy, MRU, to decide which index files need to be evicted from C_I and which data files need to be evicted from C_D . With the intuition that queries in query workloads are often near each other [8, 25], we choose MRU. We first present two strategies based on how the index files are accessed and cached. **Strategy 1:** In this strategy, the index cache, C_I , is further divided uniformly between the m hash functions. In each of the *sub-index cache*, we have a replacement policy specific to the *sub-index cache*. **Strategy 2:** The index cache, C_I , is not further divided uniformly between the m hash functions. We have a single MRU replacement policy for the index cache that stores (and if needed, evicts) index files from different hash functions. In both strategies, there is a separate MRU replacement policy for the data cache, C_D . The intuition behind Strategy 1 is that each projection will use the cache uniformly. The intuition behind Strategy 2 is that some projections might bring more hash buckets into the cache than others.

Cost Models of qwLSH: In order to determine the appropriate split between the index cache, C_I , and the data cache, C_D , we train a model based on the size of index files and data files that are read from the external storage for different settings (cardinality and dimensionality) on the DeepSat dataset [9]. Figure 2 (a) shows the optimal setting of the index cache size for different dataset characteristics that returns the least amount of total IO. This model validates our observation that datasets with different data characteristics utilize the cache differently. Datasets with low dimensionality ($=50$) require more index cache utilization ($\geq 80\%$) since the data points are small in size. On the contrary, datasets with low cardinality ($\leq 100K$) and high dimensionality (≥ 500) require less index cache utilization since the data points are large in size and the number of index files needed to be brought in memory are low. Hence, it is beneficial to cache more data points than index files to reduce the total amount of IO. Figure 2 (b) shows the effect of different Index Cache size setting for P53 dataset. As the Index Cache size setting increases (i.e. more % of the cache is allocated to the index cache), as expected, the DataIO cost increases while the IndexIO cost decreases. When Index Cache size setting is at 40%, the system incurs

the least amount of IO while processing the entire query workload. Our model also estimates the best Index Cache size setting for P53 dataset to be at 40%. Our model is dependent on the cache size and the number of queries in the query workload.

Workflow of qwLSH: In order to find the optimal index cache size for efficient utilization of the cache, we have to first generate a training model that shows the behavior of different cache sizes for different dataset settings (Figure 2 (a)). This process is done offline and does not require the knowledge of the query workloads beforehand. During query processing, once we know the characteristics of the dataset (i.e. the cardinality and dimensionality), we refer to our model to determine the index cache size and the data cache size. The underlying LSH index (on the disk) is not changed.

5 EXPERIMENTAL EVALUATION

All experiments were run on machines with Intel Core i7-6700, 16GB RAM, 2TB HDD, and Ubuntu 16.04 OS. The reported results are an average of 5 runs. We implemented a cache on top of the state-of-the-art QALSH code [20] (*qwLSH* can be implemented over any state-of-the-art LSH variant). All codes were written in C++11. We use the default settings of QALSH ($c = 2$, $w = 2.719$, $\delta = 1/e$) [20]. Since there is no work that directly aims at solving our problem, we compare our work with: **QALSH_Naive** (the baseline QALSH algorithm), **QALSH_Ci** (where we allocate 99% of the total cache size to C_I . Hence, C_D will have 1% of the total cache size allocation), **QALSH_Cd** (where we allocate 99% of the total cache size to C_D), **QALSH_CiCd** (where we allocate 50% to C_I and 50% to C_D), and **QALSH_Opt** (where we report the best setting for C_I where $C_I = 1, 10, 20, \dots, 90, 99\%$).

Datasets: In order to train our models, we needed a large dataset with high-cardinality and high-dimensionality. Due to the lack of such datasets, we followed the same technique used to generate the Mnist [1] dataset by downloading 324,000 images from the Deepsat dataset [9], and then converting each image to a 3136-dimensional point. From this dataset, we generated 20 different datasets (with different cardinalities and dimensionalities) to create our model. In order to test our model and the effectiveness of *qwLSH*, we used the following four commonly used real high-dimensional datasets: **Audio**[4] (consists of 54387 192-dim. points), **LabelMe**[30] (consists of 181093 512-dim. points), **P53**[5] (consists of 31008 5409-dim. points). Since our training dataset consisted of only 3136 dimensions, we reduce the dimensionality of each point to 3000), **Sift**[2] (consists of 250,000 128-dim. points).

Evaluation Criteria and Parameters: Since we do not change the logic of the LSH algorithm, the accuracy of each query, the

Figure 3: Comparison of *qwLSH* (TotalIO) against its alternatives (for different real datasets) and default settings Figure 4: Comparison of *qwLSH* (TotalIO) for varying cache size against its alternatives (for Audio and P53 datasets) Figure 5: Comparison of *qwLSH* (TotalIO) for varying number of queries against its alternatives (for Audio and P53 datasets)

index size and construction cost are the same as the underlying LSH algorithm (and hence not reported). We evaluate the effectiveness of *qwLSH* by comparing the size of the data (i.e. the index files and the data objects) that is needed to be brought into the cache. We found that Strategy 1 and Strategy 2 (Section 4) gave similar results, and hence we present only results using Strategy 1. We chose the following parameter settings (default are in bold): $|Q|$ (50, 100, **250**), and Total Cache Size (8, **16**, 20) in MB. We generate query workloads by choosing queries from dense regions of the dataset (that reflects our motivation and prior works [8]). We chose the maximum cache size setting based on an L3 cache size [33].

Discussion of the Performance Results: We first show the effect of different values of top- k for the P53 dataset in Fig. 2 (c). It can be seen that k does not have a noticeable effect on the performance of the algorithms. Hence, in further charts, we only consider $k=50$ due to space limitations and for simplicity purposes. In our experiments, we also observed that the alternative, QALSH_Cd, always gave the worst result (because of insufficient space to cache the index files). Hence we also omit QALSH_Cd from the charts. In Figure 3, we compare *qwLSH* against all alternatives on the 4 real datasets for the default settings [16MB cache, $|Q|=250$]. While for Audio, LabelMe, and Mnist, our proposed model always matches with the optimal setting (QA_Opt), QA_Opt for P53 returns a slightly lower IO than *qwLSH* (154.66 vs. 157.22). This is because our model returns an Index Cache size setting of 45.41%. Since we only compare the setting sizes with increments of 10 (i.e. for 40% and 50%), our model does not return the best answer. Even with this small drawback, the difference between the IOs is small. For LabelMe and Sift, the Index Cache size setting of 99% is always the best option because

the IndexIO cost is always very dominant. Due to this reason (and space limitations), we only present the results for Audio and P53 datasets in our next experiments.

Effect of Varying Total Cache Size: Figure 4 shows the effect of different cache sizes on *qwLSH*. While *qwLSH* does not always return the best setting, it is still always very close to the optimal setting. For some scenarios (e.g. Audio, 8MB), the optimal answer is close to QA_Ci, whereas for some scenarios (e.g. P53, 20MB), the optimal answer is closer to QA_CiCd. In both scenarios, *qwLSH* is able to adapt and return results closer to the optimal setting.

Effect of Varying Number of Queries in the Query Workload: Figure 5 shows that *qwLSH* can adapt to different number of queries in the query workload while returning the best answer or the answer that is close to the best setting.

Overhead of *qwLSH*'s Cache Implementation: Figure 2 (d) shows that overhead of *qwLSH* is negligible for Audio and P53 datasets. *qwLSH* is still faster than QALSH_Naive because of the efficient cache utilization and the resultant savings in the total IO.

6 CONCLUSION

In this paper, we presented a novel cache-conscious index structure, *qwLSH*, for efficient execution of query workloads in high-dimensional spaces. Based on important observations about the effect of cardinality and dimensionality of a dataset on cache utilization, we intelligently divided a given cache during processing of a query workload by using novel cost models. Experimental analysis over different real datasets under different settings showed the effectiveness of *qwLSH*.

REFERENCES

- [1] 1998. *Mnist*. <http://yann.lecun.com/exdb/mnist/>
- [2] 2004. *Sift*. <http://corpus-textmex.irisa.fr/>
- [3] 2005. *E2LSH*. <https://www.mit.edu/~andoni/LSH/>
- [4] 2006. *Audio*. <http://www.cs.princeton.edu/cass/audio.tar.gz>
- [5] 2006. *P53*. <https://archive.ics.uci.edu/ml/datasets/p53+Mutants>
- [6] Danian Achachev, Bernhard Seeger, and Peter Widmayer. 2012. Sort-based Query-adaptive Loading of R-trees. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*. ACM, New York, NY, USA, 2080–2084. <https://doi.org/10.1145/2396761.2398577>
- [7] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid Aref. 2016. Kangaroo: Workload-Aware Processing of Range Data and Range Queries in Hadoop. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining (WSDM '16)*. ACM, New York, NY, USA, 397–406. <https://doi.org/10.1145/2835776.2835841>
- [8] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamer Qadah. 2015. AQWA: Adaptive Query Workload Aware Partitioning of Big Spatial Data. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2062–2073. <https://doi.org/10.14778/2831360.2831361>
- [9] Saikat Basu, Sangram Ganguly, Supratik Mukhopadhyay, Robert DiBiano, Manohar Karki, and Ramakrishna Nemani. 2015. DeepSat: A Learning Framework for Satellite Imagery. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '15)*. ACM, New York, NY, USA, Article 37, 10 pages. <https://doi.org/10.1145/2820783.2820816>
- [10] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH Forest: Self-tuning Indexes for Similarity Search. In *Proceedings of the 14th International Conference on World Wide Web (WWW '05)*. ACM, New York, NY, USA, 651–660. <https://doi.org/10.1145/1060745.1060840>
- [11] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P. Drake, Jane M. Landolin, and Adam M. Philipp. 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology* 33, 6 (Jun 2015), 623–630. <https://doi.org/10.1038/nbt.3238>
- [12] Ruben Buaba, Mohamed Gebri, Abdollah Homaifar, Eric Kihn, and Mikhail Zhizhin. 2010. Locality Sensitive Hashing for satellite images using texture feature vectors. In *2010 IEEE Aerospace Conference*. IEEE, 1–10. <https://doi.org/10.1109/AERO.2010.5447003>
- [13] Jeremy Buhler. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17, 5 (May 2001), 419–428. <https://doi.org/10.1093/bioinformatics/17.5.419>
- [14] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. <https://doi.org/10.14778/1920841.1920853>
- [15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SCG '04)*. ACM, New York, NY, USA, 253–262. <https://doi.org/10.1145/997817.997857>
- [16] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive Hashing Scheme Based on Dynamic Collision Counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 541–552. <https://doi.org/10.1145/2213836.2213898>
- [17] Jinyang Gao, H.V. Jagadish, Beng Chin Ooi, and Sheng Wang. 2015. Selective Hashing: Closing the Gap Between Random Search and k-NN Search. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 349–358. <https://doi.org/10.1145/2783258.2783284>
- [18] M. Gibas, G. Canahuat, and H. Ferhatosmanoglu. 2008. Online Index Recommendations for High-Dimensional Databases Using Query Workloads. *IEEE Transactions on Knowledge and Data Engineering* 20, 2 (Feb 2008), 246–260. <https://doi.org/10.1109/TKDE.2007.190690>
- [19] Richard A. Hankins and Jignesh M. Patel. 2003. Effect of Node Size on the Performance of Cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '03)*. ACM, New York, NY, USA, 283–294. <https://doi.org/10.1145/781027.781063>
- [20] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware Locality-sensitive Hashing for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 9, 1 (Sept. 2015), 1–12. <https://doi.org/10.14778/2850469.2850470>
- [21] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*. ACM, New York, NY, USA, 604–613. <https://doi.org/10.1145/276698.276876>
- [22] Yingfan Liu, Jiangtao Cui, Zi Huang, Hui Li, and Heng Tao Shen. 2014. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 7, 9 (May 2014), 745–756. <https://doi.org/10.14778/2732939.2732947>
- [23] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 950–961. <http://dl.acm.org/citation.cfm?id=1325851.1325958>
- [24] Parth Nagarkar and K. Selçuk Candan. 2014. HCS: Hierarchical Cut Selection for Efficiently Processing Queries on Data Columns using Hierarchical Bitmap Indices. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24–28, 2014*. 271–282. <https://doi.org/10.5441/002/edbt.2014.26>
- [25] Parth Nagarkar, K. Selçuk Candan, and Aneesa Bhat. 2015. Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads. *PVLDB* 8, 12 (2015), 1382–1393. <http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf>
- [26] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 61–72. <https://doi.org/10.1145/2213836.2213844>
- [27] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 430–441. <https://doi.org/10.1145/2452376.2452427>
- [28] Jun Rao and Kenneth A. Ross. 2000. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 475–486. <https://doi.org/10.1145/342009.335449>
- [29] Zeehasham Rasheed, Huzefa Rangwala, and Daniel Barbará. 2013. 16S rRNA metagenome clustering and diversity estimation using locality sensitive hashing. *BMC Systems Biology* 7, Suppl 4 (Oct 2013), S11. <https://doi.org/10.1186/1752-0509-7-S4-S11>
- [30] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. 2008. LabelMe: A Database and Web-Based Tool for Image Annotation. *International Journal of Computer Vision* 77, 1 (01 May 2008), 157–173. <https://doi.org/10.1007/s11263-007-0090-8>
- [31] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2017. Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs. In *Data Management on New Hardware*, Spyros Blanas, Rajesh Bordawekar, Tirthankar Lahiri, Justin Levandoski, and Andrew Pavlo (Eds.). Springer International Publishing, Cham, 1–17.
- [32] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving C-approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 1–12. <https://doi.org/10.14778/2735461.2735462>
- [33] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over One Billion Tweets Using Parallel Locality-sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1930–1941. <https://doi.org/10.14778/2556549.2556574>
- [34] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-dimensional Space. *ACM Trans. Database Syst.* 35, 3, Article 20 (July 2010), 46 pages. <https://doi.org/10.1145/1806907.1806912>
- [35] Kostas Tzoumas, Man Lung Yiu, and Christian S. Jensen. 2009. Workload-aware Indexing of Continuously Moving Objects. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1186–1197. <https://doi.org/10.14778/1687627.1687761>
- [36] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. *CoRR abs/1408.2927* (2014). <http://arxiv.org/abs/1408.2927>
- [37] C. E. Yoon, O. O'Reilly, K. J. Bergen, and G. C. Beroza. 2015. Earthquake detection through computationally efficient similarity search. *Science Advances* 1, 11 (Dec 2015), e1501057–e1501057. <https://doi.org/10.1126/sciadv.1501057>
- [38] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnel, and Fred Gustavson. 2007. An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM, New York, NY, USA, 93–104. <https://doi.org/10.1145/1248377.1248394>
- [39] Yuxin Zheng, Qi Guo, Anthony K.H. Tung, and Sai Wu. 2016. LazyLSH: Approximate Nearest Neighbor Search for Multiple Distance Functions with a Single Index. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2023–2037. <https://doi.org/10.1145/2882903.2882930>