Large-scale distributed similarity search with Locality-Sensitive Hashing

João Pedro Ferro Queimado - 55977
Estudo Orientado em Engenharia Informática
Mestrado em Engenharia Informática
Faculdade de Ciências, Universidade de Lisboa
fc55977@alunos.fc.ul.pt

Abstract

Locality Sensitive Hashing (LSH) is an efficient hash-based solution that solves the Approximate Nearest Neighbor problem in high-dimensional data environments. The recent growth in user-generated data led to the creation of large datasets that require faster and more efficient methods to search entities. LSH achieves fast search times by creating an index that stores similar data objects close to each other, but it incurs extra computational steps and additional data to be stored. The high computational and storage requirements of traditional LSH algorithms in big datasets vouch for the development of resource-efficient LSH solutions. In this work, we propose a distributed LSH solution that focuses on balancing storage, processing, and network requirements, without compromising LSH's search efficiency. Furthermore, we present a distributed storage solution, based on erasure codes, that allows data objects to be stored directly in the LSH's multi-maps with low storage overhead, storage fault-tolerance, and uniform storage balance across nodes.

Keywords Locality Sensitive Hashing (LSH), Client-Server, Big-data, Approximate Nearest Neighbour, Erasure codes

1 Introduction

1.1 Motivation

Due to the recent growth in popularity of the Internet, the number of users increases each day. This rise in users translates to an increase in user-generated data, which accumulate into large datasets with multiple features or even dimensions. Since users also consume information from the Internet, they need faster and more efficient algorithms to keep search times reasonable [15].

Nearest Neighbor (NN) algorithms, also known as Similarity search algorithms, compose the group of algorithms that search datasets for objects based on their similarity to a given query object. Similarity search algorithms are a well-studied subject in multiple data science areas, such as databases studies, computer vision, and computer linguistics, but the most common use for these types of algorithms is in machine learning [15].

The push for faster similarity search algorithms led to the development of Approximate Nearest Neighbor (ANN) algorithms, that further improve search times from the previously mentioned NN algorithms, by making probabilistic assumptions related to objects' similarities. Locality-Sensitive Hashing (LSH) has been widely recognized as the most effective method to solve the ANN problem [10]. However, some of the traditional LSH solutions come with poor management of the available machine resources, mainly the storage ones, which aggravates with more data.

1.2 Problem statement

Several issues need to be considered when dealing with large quantities of data. The first issue involves how and where to store such datasets since storing large quantities of data usually requires expensive storage solutions. The second main issue involves how to perform fast and efficient searches in such datasets. These problems tend to aggravate if we consider that data can be added to datasets during the service lifespan, therefore growing over time.

Most datasets are feature-rich, meaning that each data object contains several features. These datasets behave similarly to a set of vectors with multiple dimensions. Large multi-dimensional datasets also fall under the previously mentioned issues. In addition to these issues, multi-dimensional data also fall under the curses of dimensionality, which refers to the changes of behavior as the number of features increase.

LSH implementations present a fast solution to the ANN problem, but this is achieved by adding extra pre-processing steps that generate LSH values and store them together with the original data objects in an index. Since most LSH implementations define this index as a group of multi-maps, naturally, the number of storage requirements in an LSH system will poorly scale with the amount of data to be stored.

1.3 Goals

In this work, we propose a distributed solution for LSH in big-data environments. Our solution uses erasure codes to store data objects near the distributed LSH index with low storage overhead, removing the need of additional storage solutions to maintain data objects. Erasure codes also grant a uniform storage balance between machines and provide fault tolerance.

We plan to implement the proposed method and evaluate its performance against other distributed LSH solutions. Our implementation will also allow the use of various LSH and storage backend solutions, so that tests can be performed with different settings in different environments. We also plan to distribute our implementation as a public, free, open-source software, enabling it to be freely altered and redistributed.

1.4 Contributions

The main contributions from this work are:

- A review of the state of the art in distributed LSH solutions and the identification of three key factors that improve search times.
- The design of a system that solves the ANN problem based on a distributed LSH that balances all the three key-points.
- An implementation of the proposed method, as an open-source framework that works with many different LSH implementations and storage backend solutions.
- A performance and cost evaluation of the implemented framework compared to previously published solutions using well-known datasets.

1.5 Document structure

The remainder of this document is organized as follows: In Section 2, we introduce the main concepts and technologies used in our work. Section 3 discusses the state of the art in distributed LSH implementations and groups them by their main focuses. In Section 4, we describe the proposed model and the datasets we plan to use in our evaluations. Section 5 presents our plans for the remainder of the thesis period.

2 Background

In this section, we present a brief description for the main concepts and technologies used in our work.

2.1 Distance Metrics

Distance metrics are a key component in similarity search since they define an algorithm that quantifies the similarity between data vectors. Many online articles introduce multiple distance metrics [4, 12]. In the remainder of this section, we will focus on presenting some of the most important distance metrics used in traditional LSH implementations.

Euclidean distance: is the most commonly used distance metric due to its simplicity and performance. The Euclidean distance is defined by the size of the segment that connects two vectors. The Euclidean metric is not scale-invariant therefore requiring feature normalization before being applied. It also falls in the curse of dimensionality since it expects low dimensional and high dimensional vectors to behave the same way.

Cosine distance: solves the high dimensionality problems with the Euclidean distance metric. This distance metric defines the distance between vectors as the cosine of the angle between vectors, which represents the product of both vectors if they were normalized to size one. Vectors equal to each other will have a cosine value of 1, and those in opposite directions will have a cosine value of -1. The Cosine metric does not take into account the vector's magnitude, which means that differences in vector's feature values are not being considered, only the vector's direction is evaluated. This causes an issue while using the Cosine distance to measure the distance between two vectors multiple of each other, as the angle between them is zero.

Jaccard distance: defines the distance based on the Jaccard index. The Jaccard index is a similarity metric that divides the number of equal values between vectors by the total number of different values. The Jaccard index from two vectors can be defined as the percentage of equal values shared between them.

Hamming distance: defines the similarity between vectors as the number of different values. This distance metric applies only to vectors with the same length, although some adaptations support different size vectors and do not account for distances between nonbinary values, only counting if the values are different or not.

Manhattan distance: (or City Block Distance) is a distance metric that corresponds to the sum of absolute differences between vector coordinates. Considering two points, A and B, in a bi-dimensional Cartesian plane (x, y), the Manhattan distance between point A and B is the distance between A and B in the x coordinate summed to the distance between A and B in the y coordinate.

2.2 LSH

2.2.1 LSH functions

Hash Functions are algorithms that translate data objects into a set of bytes with a fixed length to mapped into an Index. Many of these functions have been developed with specific purposes in mind.

Cryptographic hash functions are hash functions that include security properties such as non-reversibility. This property assures that byte arrays obtained via cryptographic hash functions cannot be used to deduct the original data object or any of its properties, this property is denoted as diffusion. These hash functions also have a very low collision rate, meaning that the probability of two different data objects have the same hash value is very low.

LSH functions are hash functions designed specifically for similarity search. These hash functions oppose the cryptographic hash function's diffusion property as they preserve the object's properties and features, allowing similar data objects to cause collisions in their hash values. This property allows similar objects to be mapped closer together in the index.

The collision rate for LSH Functions depends on a similarity threshold. This similarity threshold defines how similar two objects must be to cause a collision between their hash values.

2.2.2 LSH Multi-maps

Maps are associative data structures that relate a set of key objects to a set of value objects. Implementations of maps using arrays resort to leveraging hash functions to translate key objects into a numeric value inside the array range.

Multi-maps are data structures similar to maps, but instead of associating a key with one value, they relate a key to a list of values. Multi-maps are used in LSH as a comparison tool, used in the search phase as a fast way to retrieve all candidates answers to a query.

The LSH index is built by a set of multi-maps where each multi-map will associate a specific predetermined partition of the LSH value with a list of all objects that contain that same partition in their LSH value. This allows for objects with similar partitions of the LSH value to be considered similar, effectively binding the number of multi-maps to the similarity threshold of the algorithm.

2.3 Erasure codes

Erasure codes are a set of symbols mainly used in error correction. These codes are obtained by encoding an object with an erasure encoding function and a subset of codes can be used to restore the original object through an erasure decoder function. Note that in the decoding process, only a set of erasure codes are required to restore the original data object, which allows the correction of data omission faults. These codes are used to disperse objects in distributed storage systems where some machines may fail and not deliver requests. Erasure codes also allow datasets to be uniformly dispersed in distributed storage solutions.

3 Related Works

Similarity search is a well-studied subject in the centralized setting due to its importance in database research and machine learning applications. Since our work focuses on developing LSH in a distributed setting, most of the research we analyse mainly focuses on existing distributed LSH solutions and how they tackle the obstacles mentioned in Section 1.2.

By studying existing distributed LSH solutions, we identified three pillars required to achieve search efficiency and distributed the related work in accordance with them in Table 1. These pillars consist of storage limitations, network overflows and computational loads. These pillars directly affect each other, which means that improving one pillar may negatively affect all others. This issue is one of the most commonly observed challenges during our research and most solutions opt to develop only one or two pillars while sacrificing performance in the remaining ones.

Table 1. Table associating distributed LSH tools with search efficiency pillars.

Related Work	Storage	Processing	Network
LSH at Large [6]	✗		✗
SES-LSH [8]	✗	✗	
C2Net [9]	✗		✗
Haghani *et al.* [5]	✗		✗
Bahmani *et al.* [1]	✗		✗
Durmaz *et al.* [3]	✗	✗	
Patil *et al.* [11]	✗	✗	✗

Previous works have focused on the network and storage pillars by presenting a distributed solution where LSH Buckets are dispersed across peers [1, 5, 6]. This approach enables similar objects to be stored in the same nodes while keeping some level of data distribution across the system. Network efficiency is also improved since queries only need to access one node to retrieve all answers. Storing all similar objects in the same peers causes a storage balancing issue in applications that require large similarity thresholds or homogeneous datasets. To solve this problem, some solutions include a mechanism that considers load balance during the indexing phase, by allowing similar data to be stored in the neighboring peers [5]. This is achieved by representing the LSH index as a Distributed Hash Table (DHT) using for instance Chord [14]. Solutions such as C2Net [9] also focus on the storage and network pillars by constructing a similarity cluster graph that utilizes a minimum spanning tree to distribute the clusters through the data nodes. This approach promises good network load balance since similar data is stored in the same node while dealing with load balance by distributing buckets according to their weight. These solutions may present a good load balance through the system and good network usage, but they do not guarantee a uniform load balance throughout the system, nor deal well with continuously growing datasets.

Some solutions propose the randomized distribution of data through data nodes (e.g., [3]). During the indexing phase, the dataset is divided into n equal subsets, where n corresponds to the number of data nodes available. Each data subset is assigned to a random data node so data objects can be added to the node's LSH index. During the query phase, a centralized node is used to broadcast query objects directly to each data node. Each node uses the query object to collect all data objects that are similar to the given query in their local index. These data objects are then sent to the central node that gathers all data objects from all data nodes and processes the final answer to the query object. Random distribution of data allows for a better representation of data in a distributed system, but it does not guarantee a uniform load balance between data nodes, which may overload some nodes with data.

SLSH [8] is an LSH implementation that provides good storage load balance on top of Spark [19, 20]. SLSH uses Spark's Resilient Distributed Databases (RDD) [19] implementation to partition both LSH index and data objects into different Querier nodes. LSH in RDD presents some issues related to the distribution of data contained in hash tables that led to heavy data shuffles when constructing the data RDD [8]. These issues led the authors to develop a Shuffle-Efficient Similarity Search scheme (SES-LSH) [8], which improves SLSH by adding efficient shuffle mechanisms. These solutions improve network and storage performance by adding computational steps that increase the overall systems computational requirements.

4 Our Proposal

4.1 Overview

In this section, we present a distributed model for LSH similar to a Client-Server model where most computationally intensive tasks are executed at the client-side and storage tasks are performed at the server-side. Furthermore, we present a data storage model for LSH based on erasure codes that allows data objects to be divided and stored in the distributed multi-maps, removing the need for additional storage solutions and data structures while guaranteeing a uniform storage load balancing. Figure 1 depicts the overview of the proposed model where both sides of the system and the relation between them can be seen.

In our model, data objects are required to have a unique identifier. For this requirement, we take advantage of the cryptographic hash functions properties. Cryptographic hash functions have a very low collision rate, meaning that the probability of two objects generating the same hash value is very low. Therefore, for the remainder of this section, we will refer to the object's cryptographic hash value as its name.

We propose the use of erasure codes to disperse data objects through the set of distributed LSH multi-maps. Storing data objects inside the distributed LSH index using erasure codes guarantees uniform data distribution between storage nodes and removes the necessity for additional storage solutions and data structures.

Clients are responsible for all pre-processing and post-processing tasks that are required in LSH, such as the computation of the LSH-hash, the encoding and decoding of erasure codes, and the computation of cryptographic hash functions. They can interact with the servers in different ways, which means that specific clients can be created to perform specific tasks, such as the Inserter and Querier nodes described later in this section. Clients can also be connected to each other to better distribute the system's computational load.

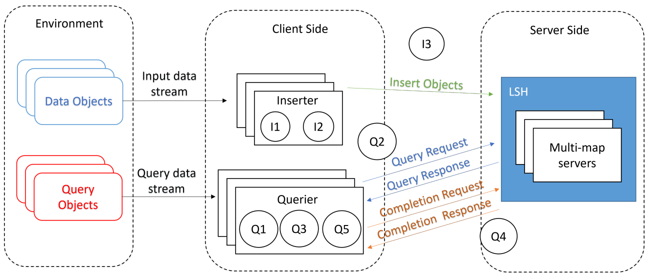

Figure 1. An overview of the proposed model.

The server side of our system contains the functioning core of the LSH, as the complete set of servers containing multi-maps represents the LSH index. Since data objects are divided between servers in the form of erasure codes, additional mechanisms that request erasure codes from multi-map servers are required to reconstruct data objects. Erasure codes also offer fault tolerance against node omissions, meaning that only a subset of servers is required to deliver the erasure code request.

4.1.1 Inserter

Inserter nodes are specialized client nodes designed to insert data objects in the LSH index. Each Inserter node contains a routine that is triggered once per data object. This routine can be divided into two phases, Pre-Process and Distribute.

The first phase is designated Pre-process and consists of a set of instructions that transform the data object so it can be inserted into the LSH index. This phase, as depicted in Figure 2, uses the three functions: cryptohash that returns the object's name, Erasure Encode that returns the object's set of erasure codes, and LSH function that returns the object's LSH-hash.

The second phase is designated Distribute and consists of the distribution of data through the LSH index, as presented in Figure 3. During this phase, the information collected in the previous phase is grouped in data packages to be sent to the multi-map Servers. Each package is created specifically for the multi-map it is destined to. Therefore, it only contains information relevant for that multi-map, such as the LSH-hash subset destined to the multi-map, one erasure code, and the data object's name. These packages are sent directly to the multi-map servers, where the erasure code is grouped with the object's name and stored in the list corresponding to the LSH-hash subset.

4.1.2 Querier

Querier nodes are Client nodes designed to respond to query requests. Just like the Inserter nodes, Querier nodes also have a routine that is triggered once per queried object. This

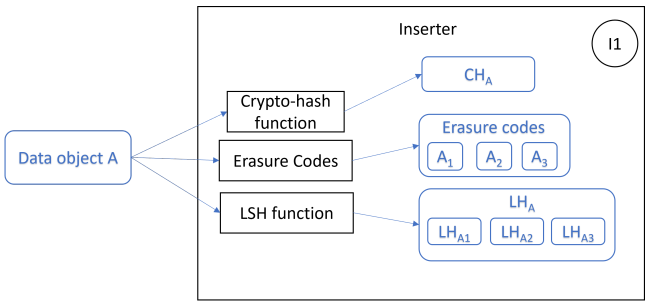

Figure 2. Inserter Phase 1: Pre-process

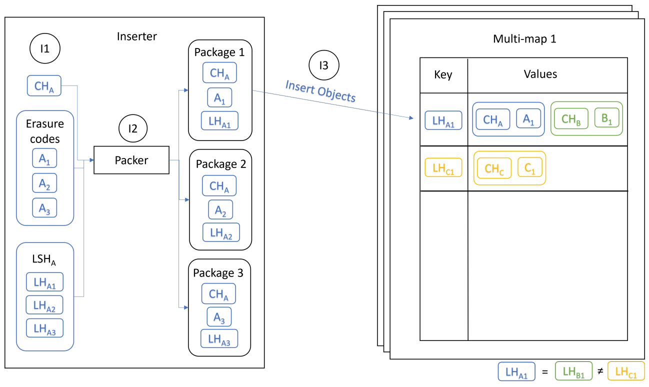

Figure 3. Inserter Phase 2: Distribute

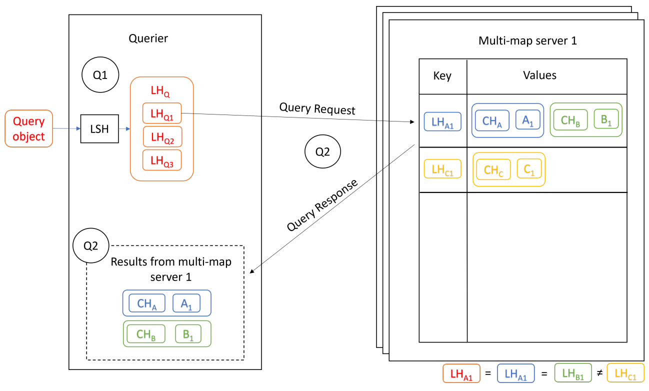
Querier
Q1
Query object
LSH
LH_Q
LH_{Q1} LH_{Q2} LH_{Q3}
Query Request
Q2
Query Response
Results from multi-map server 1
Multi-map server 1
Key
Values
LH_{A1} = LH_{A1} = LH_{B1} ≠ LH_{C1}

Figure 4. Querier Phase 1: Query

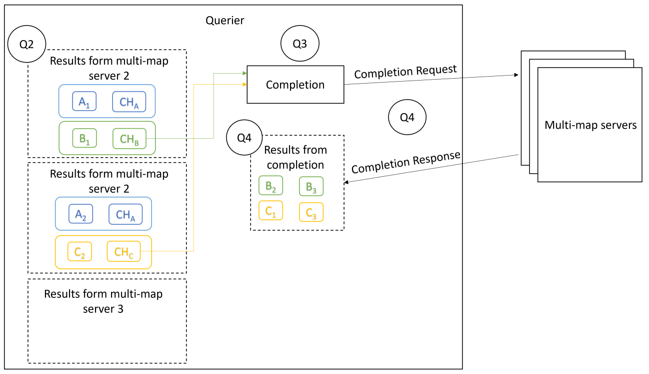

Figure 5. Querier Phase 2: Completion

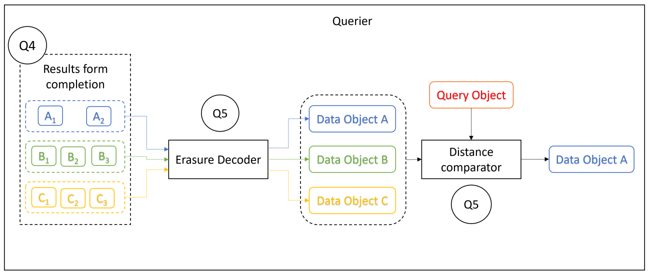

Figure 6. Querier Phase 3: Decode and Compare

routine contains a three-phase process that searches the LSH-index for IDs similar objects, retrieves and assembles the said objects, and searches these objects for the best candidate to return in the query reply. Querier routines can be altered depending on the system requirements. In the following section, we will present the Querier node's routine that returns the most similar data object to a given query object.

The first phase, designated Query, is depicted in Figure 4 and consists of the extraction of similar data objects from the LSH index. This phase starts by generating the LSH-hash from the query object through the same LSH function used in the Inserter nodes. The resulting LSH-hash is then divided into subsets so that each subset can be sent to the respective multi-map server in a Query Request message. Each multi-map then responds with a Query Response message, which contains the list of all data pairs mapped by the given LSH-hash subset. Each received pair contains the data object's name and one of the data object's erasure codes.

Since the previous phase does not guarantee that all gathered data objects have enough erasure codes to restore the original data objects, a Completion phase may be required to retrieve the remaining erasure codes from incomplete data objects. During this phase, the Querier node groups all erasure codes received in the previous phase by the object's name and checks if the number of received erasure codes is enough to assemble the data objects. If not, the Querier node sends a Completion Request message containing the data objects name and the object's LSH value, to all multi-map servers, that respond with a Completion Response message, containing the erasure code mapped to the data object's name.

The third phase, denoted Decode and Compare, consists of the process of assembling data objects and the process of retrieving the closest data object to the query object, as shown in Figure 6. Two main functions, Erasure Decoder and Distance Comparison, are required in this phase. The Erasure Decoder function receives all erasure codes from the same data object and assembles them into the original data object. The Distance Comparison function measures and compares the distance between a set of data objects to a query object to determine which data object is the closest (i.e., the nearest) to the query object according to a distance metric. Decode and comparison are placed in the same phase since both functions can be executed in the same routine to better manage the Querier node's processing and storage resources.

4.1.3 Multi-map Server

The server-side is composed of a set of server nodes that store data objects in multi-maps. Each Multi-map server only contains one multi-map, meaning that there is a minimum server requirement to guarantee the system's accuracy. Multi-maps, in our scheme, are similar to multi-maps present in centralized LSH implementations, where each key is an LSH-hash subset. However, instead of storing full data objects, our multi-maps store a data pair containing one erasure code and the data object's name. Figures 3 and 4 present one multi-map server represented by the Multi-map 1.

4.2 Adaptability and Alternative Implementations

In this section, we delve into the possible modifications that allow our system model to be adapted to the environment in which it will be deployed.

Our proposed model presents a distributed LSH index that assigns multi-maps to system nodes, allowing any LSH functions that support the multi-map scheme to be integrated into our system. Our system also defines object's names as the object's cryptographic hash value. This naming scheme can be performed by any cryptographic hash function or UID (Unique IDentifier) generator, as long as it presents a low collision rate and the resulting name size is considerably smaller than the original data object.

As mentioned before, the complete specification of our client implementations remains open, which means that the overall function of a client can be changed to better adapt the system to its environment. Therefore, clients can be set to communicate with each other forming a peer-to-peer network. This option allows the implementation of several mechanisms for distributed computational load balancing, fault tolerance, and Byzantine fault tolerance.

Our approach to the server-side also leaves some room for adaptability. Each server only requires the standard multi-map operations and the additional operation that retrieves any erasure code based on their name. Therefore, any multi-map database implementation can be integrated into our proposed method without significant changes. Our server model also tolerates adding more servers than the required minimum. This means that servers can be replicated, adding fault tolerance against crash faults.

The use of erasure codes is completely optional in our presented method. Our approach allows clients to integrate external data storage backends to store objects separately from the multi-maps. Adding an external storage solution may create a network bottleneck since all client requests converge in a single network point. External storage solutions may also increase the system's cost.

Our model allows erasure codes to be configured to tolerate omission faults. Erasure encoders and decoders can be configured to tolerate certain numbers of missing erasure codes. The number of erasure codes is related to the number of faults tolerated, which means that increasing the number of supported faults increases the system's storage overhead since servers are required to store more codes.

4.3 Data

Our approach focuses on processing big datasets containing high dimensional data objects. High dimensional data objects are objects that contain a large number of features.

For instance, related studies have already used public datasets containing SIFT keypoints [3, 10, 13, 17, 18]. SIFT is a computer image algorithm that extracts keypoints from a set of images and stores them into a database. SIFT keypoints are circular image regions with orientation, where each keypoint contains a set of coordinates, a scale, and an orientation [16].

Other compatible datasets are vector representations for words. These datasets have also been used in previous works (e.g., [10]). Additionally, Glove [7] is an unsupervised learning algorithm that can be used for the generation of vector representations of words.

Another dataset type already explored in LSH solutions encompasses sequencing data from human genomes (e.g., [2]). Sequencing data is composed of large strings of nucleotides (i.e., the nulceic acids) that can be later aligned to compose a complete human genome sequence, together with the respective quality scores that attest the certainty of the sequencing machine in the sequenced nucleotides.

5 Forthcoming Work

With the presented model, we plan to implement an open framework for LSH-based applications that allow one to experiment with datasets presented in Section 4.3. Since the presented datasets were used in many state of the art tools, we plan to use the testing results to compare the efficiency of our approach with the one from other state of the art approaches. We will also write a final report containing our framework's description, as well as the results from our experiments and comparisons.

References

[1] Bahman Bahmani, Ashish Goel, and Rajendra Shinde. 2012. Efficient distributed locality sensitive hashing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2174–2178.

[2] Vinicius Cogo, Joao Paulo, and Alysson Bessani. 2020. Genodedup: Similarity-based deduplication and delta-encoding for genome sequencing data. *IEEE Trans. Comput.* 70, 5 (2020), 669–681.

[3] Osman Durmaz and Hasan Sakir Bilge. 2019. Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters* 128 (2019), 361–369.

[4] Maarten Grootendorst. 2021. 9 Distance Measures in Data Science. https://towardsdatascience.com/9-distance-measures-in-data-science-918109d069fa [Online; posted 1-February-2021].

[5] Parisa Haghani, Sebastian Michel, and Karl Aberer. 2009. Distributed similarity search in high dimensions using locality sensitive hashing. In *Proceedings of the 12nd International Conference on Extending Database Technology: Advances in Database Technology*. 744–755.

[6] Parisa Haghani, Sebastian Michel, Karl Aberer, et al. 2008. LSH at large-distributed knn search in high dimensions. In *Proceedings of the 11th International Workshop on the Web and Databases (WebDB)*.

[7] Christopher D. Manning Jeffrey Pennington, Richard Socher. 2014. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/ [Online; posted August-2014].

[8] Dongsheng Li, Wanxin Zhang, Siqi Shen, and Yiming Zhang. 2017. SES-LSH: Shuffle-efficient locality sensitive hashing for distributed similarity search. In *Proceedings of the 2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 822–827.

[9] Hangyu Li, Sarana Nutanong, Hong Xu, Foryu Ha, et al. 2018. C2Net: A network-efficient approach to collision counting LSH similarity join. *IEEE Transactions on Knowledge and Data Engineering* 31, 3 (2018), 423–436.

[10] Jinfeng Li, James Cheng, Fan Yang, Yuzhen Huang, Yunjian Zhao, Xiao Yan, and Ruihao Zhao. 2017. Losha: A general framework for scalable locality sensitive hashing. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 635–644.

[11] Akshay N Patil. 2007. Distributed Multi-Probe LSH: Tackling Real World Data. (2007).

[12] Pulkit Sharma. 2020. 4 Types of Distance Metrics in Machine Learning. https://www.analyticsvidhya.com/blog/2020/02/4-types-of-distance-metrics-in-machine-learning/ [Online; posted 25-February-2020].

[13] Lu Shen, Jiagao Wu, Yongrong Wang, and Linfeng Liu. 2018. Towards load balancing for LSH-based distributed similarity indexing in high-dimensional space. In *Proceedings of the 20th IEEE International Conference on High Performance Computing and Communications (HPCC)*. 384–391.

[14] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking* 11, 1 (2003), 17–32.

[15] Eyal Trabelsi. 2020. Comprehensive Guide To Approximate Nearest Neighbors Algorithms. https://towardsdatascience.com/comprehensive-guide-to-approximate-nearest-neighbors-algorithms-8b94f057d6b6 [Online; posted 14-February-2020].

[16] Andrea Vedaldi. 2007. Scale Invariant Feature Transform (SIFT). https://www.vlfeat.org/api/sift.html [Online; posted 2007].

[17] Jiagao Wu, Lu Shen, and Linfeng Liu. 2020. LSH-based distributed similarity indexing with load balancing in high-dimensional space. *The Journal of Supercomputing* 76, 1 (2020), 636–665.

[18] Xiangyang Xu, Tongwei Ren, and Gangshan Wu. 2014. Clsh: Cluster-based locality-sensitive hashing. In *Proceedings of International Conference on Internet Multimedia Computing and Service*. 144–147.

[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.