Towards a High Dimensional Data Management System

Dong Deng[†] Lei Cao[‡] Jiachen Liu[⋆] Runhui Wang[†]

[†]Rutgers University [‡]MIT [⋆]University of Michigan

dong.deng@rutgers.edu lcao@csail.mit.edu
amberljc@umich.edu rw545@scarletmail.rutgers.edu

ABSTRACT

High dimensional data is ubiquitous nowadays and plays an important role in many artificial intelligence (AI) applications such as face recognition, image retrieval, and knowledge base construction. The semantics of images, video, documents, and knowledge can be captured by meaningful high dimensional feature vectors extracted from deep learning models. Though there are many research in querying and indexing high dimensional data, most of them are point solutions. The high dimensional data, such as feature vectors, is still largely managed by the application developers individually. In this paper, we propose to manage the high dimensional data in a systematical way and present the design of SABER, an end-to-end high dimensional data management system. SABER features scalability, high performance, and ease of use and configure. It consists of several modules, including data ingestion, storage management, index management, and polystore query processing. We aim at integrating SABER with the mature AI ecosystem and helping AI practitioners reduce the burden of managing high dimensional data.

1. INTRODUCTION

High-dimensional data is ubiquitous nowadays and plays an important role in many artificial intelligence applications such as image retrieval [2], face recognition [4], and knowledge base construction [9]. The semantics of images, video, documents, and knowledge graphs (KG) can be captured by meaningful high dimensional feature vectors extracted from deep learning models [13]. Semantic related objects, such as synonyms and edges of the same type/relationship in a KG tend to have similar feature vectors [9]. Similarity search (a.k.a., nearest neighbor search) is one of the most important query types on high dimensional data, which has been extensively studied in the last twenty years [1, 10, 13–15, 22]. Various techniques have been developed, such as the locality sensitive hashing (LSH) [10, 14], the product quantization [1, 11], the proximity graph [7, 15], the sketch-based methods [12], etc. However, most of them are point solutions and nowadays the high dimensional data is still largely managed by application developers individually. Without careful implementation, querying high dimensional data can be extremely inefficient and miss the high performance requirement in many real time applications. Thus it is necessary to build an end-to-end high dimensional data management system to ease the burden of application developers, which is exactly the purpose of this paper.

High dimensional data has the following characteristics.

- First, high dimensional data is big and grows fast. For example, Youtube-8M[1], one of the largest publicly available datasets today, contains 1.9 billion 1024-dimensional feature vectors extracted from 350,000 hours of video using the Inception network. In comparison, as of May 2019, 720,000 hours of new video were uploaded to YouTube per day[2]. The data is indeed in big volume. As another example, BIGANN[3], another large publicly available dataset, contains 1 billion 128-dimensional Scale-Invariant Feature Transform (SIFT) feature vectors extracted from 1 million images, while Instagram has more than 34 billion photos[4]. A coarse estimation gives more than 34 trillion SIFT feature vectors and many petabytes storage in raw. Thus it is critical for our system to have good scalability.

- Second, there are many kinds of high dimensional data with different sparsity, number of dimensionality, and value domain. For example, the high dimensional feature vectors extracted by deep learning techniques are usually dense, in real values, and ranges from tens to thousands of dimensions. In contrast, the TF-IDF weighed documents in information retrieval are sparse vectors and in hundreds of thousands of dimensions to millions of dimensions. Their value domain is positive value. In recommendation systems, such as the friend relationship in social networks and the purchase/rate/click history, data can be represented as ultra high dimensional (in hundreds of millions dimensions to billions of dimensions), extremely sparse, binary vectors. In addition, various metrics are designed to measure the similarity/distance between high dimensional data. Among them, Euclidean distance (Lp-Norm in general), cosine similarity, inner product,

[1]https://research.google.com/youtube8m/
[2]https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/
[3]http://corpus-texmex.irisa.fr/
[4]https://www.statisticbrain.com/instagram-company-statistics/

and Jaccard similarity are the most widely used ones. Note that, the cosine similarity can be inferred from the Euclidean distance by normalizing all vectors to unit vectors.

- Third, the high dimensional data is usually coupled with other metadata information. For example, in online e-commercial websites, feature vectors are extracted from product images, while these products also have other structured information like names, price, ratings, and inventory, as well as unstructured information like descriptions and reviews.

Based on the characteristic of high dimensional data, we design SABER, an end-to-end high dimensional data management system, which features scalability, high performance, and ease of use and configure. It consists of data ingestion, storage management, index management, and polystore query processing components. The data ingestion module extracts high dimensional feature vectors from the raw input data. The high dimensional data, along with their metadata, are stored in multiple different data stores. The polystore query processing component coordinates these data stores to process multi-modal queries [17]. SABER uses scalable and ease to configure indexes. An index is designed for the dense vectors. SABER also supports data and index updates. SABER is designed to support the following three query types for the high dimensional data.

1. Similarity Search. This is the same as the nearest neighbor (NN) search query. Specifically, given a collection of high dimensional vectors, a distance (or similarity) metric, a query vector, and an integer k, similarity search returns k vectors whose distance to the query is smallest (or whose similarity to the query is largest) under the given metric. A commonly used variant is the range similarity search query where a threshold is given instead of the integer k and all the vectors whose distance to the query is smaller than the threshold (or whose similarity to the query is larger than the threshold) are returned.

2. Similarity Join. This is the analogy to the join query in SQL. Given two collections of vectors (or one collection in the self-join case), similarity join query returns all the similar vector pairs under a given metric. The same criteria as the similarity search query is used to judge whether two vectors are similar. This can be used to join two collections of objects (e.g., images and documents).

3. Similarity GroupBy. This is the analogy to the groupby query in SQL. Intuitively, this query deduplicates the high dimensional vectors and returns clusters of similar/duplicate vectors. This is useful when the users want to deduplicate or aggregate the high dimensional data (e.g., removing the duplicate products in product search). Formally, given a collection of vectors, similarity groupby query returns clusters of vectors such that in any two clusters, none of their vectors are similar. This definition guarantees the transitivity closure: if X and Y are duplicate and Y and Z are duplicate, X are Z must also be duplicates. For ease of understand, we can build a graph where each vertex corresponds to a vector and there is an edge if and only if two vectors on the two ends are similar. Then each cluster corresponds to a connected component in this graph.

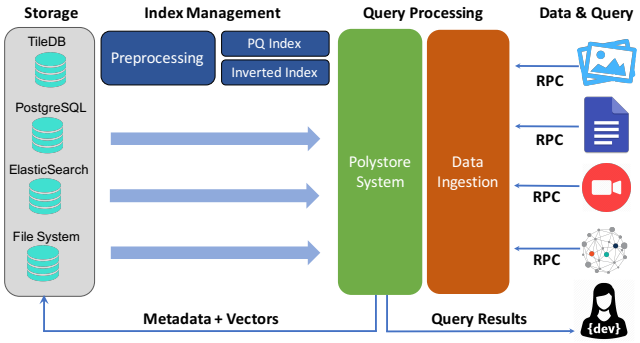

Figure 1: System architecture

SABER deals with sparse dense, real-value high dimensional vectors and sparse, positive-value ultra high dimensional vectors. These are two most interesting types of high dimensional data. Many feature vectors (especially those extracted from deep learning models) are dense, real-value vectors while TF-IDF weighted documents (and ratings in recommendation systems) are sparse, positive-value vectors. For dense vectors, Euclidian distance and cosine similarity are supported, while Jaccard similarity and cosine similarity are supported for sparse data. We expect to support more metrics and data types in the future.

The rest of the paper is organized as follows. In Section 2, we present the architecture of SABER. Section 3 introduces the data ingestion module and Section 4 discuss our novel index structure for dense vectors. We discuss the ploystore query processing in Section 5, review related work in Section 6, and conclude the paper in Section 7.

2. SYSTEM ARCHITECTURE

The system architecture is shown in Figure 1. It consists of the following components.

Data Ingestion. The application developers use Remote Procedure Call (RPC) services to interact with the system, such as the `Google gRPC` and the `Apache Thrift`. The data and query (e.g., images, documents, video, knowledge graphs) are first processed by a data ingestion module, which we discuss in details in Section 3. This data ingestion module is responsible for extracting high dimensional feature vectors as well as the corresponding metadata (e.g., the location and timestamp in a photo and the object class of an image determined by a classifier) from the input. We also accept plain high dimensional vectors as the input data and query.

Storage Management. The metadata and vectors generated from the data ingestion module are forwarded to the storage module. As discussed before, the high dimensional data is usually coupled with other structured and unstructured metadata. SABER utilizes multiple systems to store the high dimensional vectors and the metadata. For example, SABER stores the structured data in the relational database management system (RDBMS) like `PostgreSQL` while the semi-structured and unstructured data are stored in the full-text search engine `ElasticSearch`. For the raw multimedia data, a file system is employed to store them. SABER uses

Figure 2: Illustration of the multi-modal query [17].

TileDB to store the dense and sparse vectors. These systems provide fast, reliable, and scalable data read and write services.

Index Management for High Dimensional Data. For the structured, semi-structured, and unstructured data, their corresponding data stores provide powerful indexing mechanism and SABER use them as it is. For the high dimensional vectors, SABER builds and manages its own index. Many feature vectors (especially those extracted from deep learning models) are dense, real-value vectors. We leverage product quantization to compress the original dense vectors and build novel (minimum spanning) tree index for the compressed data. We also design strategies to deal with data and index updates. The details will be given in Section 4.

For the sparse, positive-value vectors, we leverage the inverted index and the filter-and-refine framework [5], such as the prefix filtering [3], to process queries. A huge advantage of these techniques is the ease of use and configure. Usually there is few to no hyper-parameters to tune. These are mature techniques and we leave out the details.

Polystore Query Processing. Our system has a polystore query processing module to coordinate the multi-modal query against the data in multiple stores. Multi-modal query involves data of different types, which is very common in the real world. Figure 2 shows product search examples in the work [17] from WalmartLab. We can issue the similarity search query over the high dimensional feature vectors extracted from the product photos in conjunction with the keyword search query over the textual data. As another example, surveillance applications usually only interests in objects (such as people and cars) that appear in a specific region or within a specific period of time. It can be expressed as a multi-modal query across the structured data and the high dimensional feature vectors. We discuss the details of polystore query processing in Section 5.

3. FAST DATA INGESTION

The system provides a rich set of tools to facilitate the users to ingest the input data in various types of formats into high dimensional vectors that effectively represent the key features of the raw input data. Based on the types of the input that users can provide, the tools can be categorized into the following types:

- Direct data loading. If the input data is already in the format of high dimensional feature vectors, then our system simply efficiently loads the data into data table leveraging the existing tools ingesting structured data.

- Extract features from supervised model. If the input data is in complex format, such as image or video, simply treating each raw image file/video frame as a high dimensional feature vector is not effective in supporting similarity search. This is because even a small rotation of image will dramatically change its representation. However, when the users are able to provide a Convolutional Neural Network (CNN) that effectively classifies the given image datasets/videos, our system fully leverages the representation learning capability of CNN and provides tools to extract features from the neural network. Instead of directly extracting features from a single layer close to the output, we design a mechanism that aggregates the states of different layers to better represent the image.

- Semi-supervised feature extraction. However, users are not always able to provide a well trained classification model due to the lack of abundant training data, etc. Therefore, we design a semi-supervised feature extraction method that uses a small set of examples to learn the high dimensional representations of the raw images/videos. The key idea is to leverage the observation that Generative Adversarial Networks (GAN) can effectively capture the data manifold during the battle of data generator and data discriminator [8]. Besides requiring the GAN to separate the fake examples and real examples, our semi-supervised method also uses the small set of examples to guide the GAN to learn a representation that can separate different classes of real data, therefore more effective in representing the raw input.

- Unsupervised feature extraction. In the case that the users cannot provide any labeled example, we provide multiple unsupervised tools to extract features from the raw data. These tools either leverage the manifold learning capacity of GAN similar to the semi-supervised tool, or use Autoencoder that is shown to be effective in learning representation without relying on any label.

- Feature extraction for text data. Besides the image/video data, our system also provides tools to effective extract features from text data including TF-IDF and word2vector [16], etc.

Leveraging the progress of AutoML, these tools save users the efforts of tuning parameters. In most of cases, the users only need to specify the dimension of the feature vector they prefer. For the structured and unstructured metadata, we use existing tools and the corresponding data store in the system to digest them.

4. INDEX FOR DENSE VECTORS

In this section, we present our novel index for the dense, real-value high dimensional vectors. We first introduce the limitations of existing approaches in Section 4.1. Section 4.2 presents our novel tree-based index and Section 4.3 discusses how to support distributed query processing. We design strategies to deal with data and index updates in Section 4.4.

	Part 1	···	Part M
Centroid 1	0.45	···	1.24
···	···	···	···
Centroid k	0.88	···	0.82

Table 1: An example distance lookup table

PQ code A: (8, 6, 10, **23**, 1, 39, 28, 65)

PQ code B: (8, 6, 10, **56**, 1, 39, 28, 65)

Distance(Q, A) = Table[8][1] + Table[6][2] + ... + Table[64][8]

Distance(Q, B) = Distance(Q, A) + Table[**23**][4] - Table[**56**][4]

Figure 3: Sharing computation between similar PQ codes.

4.1 Limitation of Existing Approaches

Similarity queries, including similarity search (a.k.a. nearest neighbor search), similarity join, and similarity groupby, play an important role in managing high dimensional data and have enormous applications. However, it is rather challenge to process similarity queries at large scale and existing approaches would all fail. Specifically, it is well known that exact methods that guarantee to return completely accurate results do not competitive with the brute-force linear scan of the entire dataset due to the "curse of dimensionality" [10]. For example, the R-tree and K-D tree index become even slower than the simple linear scan when data dimensionality is larger than 10 [18]. Existing approximate algorithms (including Locality Sensitive Hashing (LSH) based, proximity-graph based, and vector quantization based methods) can scale to higher dimension. However, they have problems when the data cardinality is at large scale.

LSH based methods. In general, LSH based methods employ many hash functions in a specific hash family to partition high dimensional data such that nearby/similar data points are more likely to be hashed to the same bucket than faraway/dissimilar data points [10, 22]. However, to achieve high accuracy, LSH based methods usually involve excessively large number of hash tables and incur huge index size and suboptimal efficiency.

Proximity-graph based methods. In proximity graphs, each vertex corresponds to a data point. Nearby data points are connected such that for any data point, one of its neighbors must be closer to the query than the data point itself [15]. Then the best first search (such as the A* search) on the proximity graph can find the nearest neighbor to a query. However, building such a graph is very expensive and storing the graph needs large space.

Another noticeable disadvantage of proximity-graph based methods and LSH based methods is that they both need to access the original data vectors to calculate their real distance to the query. This incurs many expensive random IOs and/or network communication (in distributed query processing).

Quantization based methods. Vector quantization clusters the data vectors into k clusters (using k-means for example) and approximate each vector by its nearest centroid. Product quantization evenly divides the D dimensional space into M disjoint D/M dimensional subspaces and applies vector quantization to every subspace independently [11]. Then each vector is compressed to a M$\log k$ bit code to indicate the M corresponding nearest centroids in the M subspaces. For D=128, M=16, and k=64, the compression rate is 64. More importantly, the distance to the query can be estimated using only the code; i.e., the original vectors are never accessed in query processing. However, existing product-quantization based methods need to exhaustively scan all the codes or search in an exponentially large space. This is too expensive for large scale dataset where even the codes cannot fit in memory.

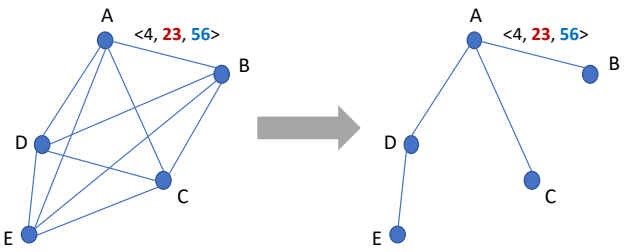

Figure 4: A minimum spanning tree generated from the full graph

4.2 Tree-based Index and Query Processing

The query processing engine for large scale high dimensional data in Saber is built based on a novel tree index structure. In a high level, we build a (wide-and-flat) tree index for all the product quantization codes. By traversing the tree, we can find the nearest neighbors to the query. The computation will be shared between all parent and child nodes. Moreover, for some tree nodes that satisfy certain conditions, we can skip traversing the entire subtrees rooted at these nodes. Same as other tree indexes like B-tree, we can store the lower level nodes on disk or in different machines for large scale dataset. The pruned branches will never be visited. Thus, random IO accesses and/or network communication in distributed computing setting will be reduced.

Sharing computation in similar PQ codes. More specifically, as shown in Figure 1(a), PQ-based methods precompute a lookup table containing the distances from all centroids to the query in the M subspaces. The code of a vector contains the indexes of the nearest centroids in the M subspace. Then the distance from a vector to the query can be estimated using M lookups. For example, in Figure 1(b), the distance from a data vector A to the query Q can be estimated by adding up the M=8 table lookups. We observe that the distance computation between "similar" codes can be shared. For example, as shown in Figure 1(b), the codes A and B only differ in the 4-th part. Thus Distance(Q, B) can be calculated by removing the differences in the 4-th part from the previously computed Distance(Q, A). In this way, only two lookups are needed instead of M=8 to calculate Distance(Q, B).

The minimum spanning tree index. To share more computation, we can (virtually) build a full graph for all the codes, where the edge weight is the number of differences between two codes. For example, as shown in Figure 2(a), the edge between codes A and B has a weight of one as they

only differ in the 4-th part. Then the minimum spanning tree generated from the graph, as shown in Figure 2(b), has the minimum total number of differences. By traversing this tree and using the differences to calculate distances, we can get the nearest neighbor to the query.

Pruning computation by filtering subtrees. We observe that some subtrees can be skipped during tree traversing. Specifically, for any node X in the subtree rooted at node A, based on the triangle inequality

$$Distance(Q, X) \geq Distance(Q, A) - sdDistance(A, X)$$

holds. Note Distance(A, X) is independent to the query Q and can be pre-computed in the offline indexing phase. Moreover, when A is visited during traversing, Distance(Q, A) is calculated. Thus we have a lower bound of the distance from any node in the subtree rooted at A to the query. If this lower bound is no smaller than the threshold (i.e., the distance from the query to the nearest neighbor visited so far), the entire subtree rooted at A can be safely pruned as it cannot yield any node closer to the query than the current nearest one.

Similarity join and similarity group query. At this point, SABER simply decomposes the similarity join and similarity groupby queries into a bunch of similarity search queries. In the future, we will design more sophisticated algorithms to process these queries more efficiently.

Same as the inverted index, product quantization has few to no hyper parameters to tune and is ease to use and configure. In contrast, the LSH-based methods and proximity-graph based methods involves multiple hyper parameters and their performance is sensitive to those parameters.

4.3 Distributed Similarity Query Processing

To further improve the scalability of similarity query processing, we propose to support distributed query processing. A simple solution evenly distributes the data to all the machines. A coordinate machine is assigned to each similarity search query. The query is processed in each individual machine and the results are returned to and aggregated by the coordinate machine.

SABER supports another alternative solution which partitions the tree index. Specifically, we first determine a few inner nodes as the boundary nodes. All the ancestor nodes of these boundary nodes consist a tree, which we name as the filter tree. The filter tree is copied to every machine. The subtrees rooted at these inner nodes are partitioned and evenly distributed to all the machines. Each query is randomly assigned a coordinate machine. The query is first processed by the filter tree to prune unqualified subtrees as discussed before. Then the query is sent to all those machines with at least one qualified subtree. Finally, the results are returned to and aggregated by the coordinate machine.

4.4 Data and Index Update

Dealing with continuously growing large scale datasets has many applications. For example, the news searching system that updates on hourly/daily basis, recommendation systems based on users' most recent behavior, and object detection in video surveillance. However, existing methods for similarity search are designed for static datasets, which lead to high computational cost for adjusting indexes to accommodate newly arrived data.

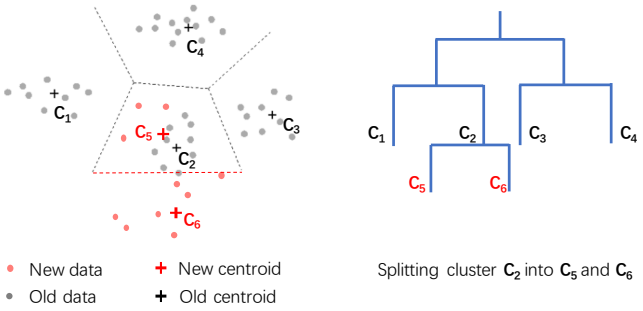

Figure 5: Hierarchical clustering for data and index updates.

Hierarchical clustering. To support efficient and accurate index updates in SABER, we leverage the hierarchical clustering (hierarchical k-means more specifically) for vector quantization. Hierarchical k-means decomposition represents an input dataset as a tree. The root of the tree points to the centroid of the whole dataset. The decomposition process iteratively divides the dataset into k sub-clusters using k-means. Each sub-cluster is then associated to a child node of the current tree node. The decomposition process continues to the sub-clusters until the size of each sub-cluster is small enough.

Index updating. Index updating is achieved by merging and splitting specific sub-clusters whenever necessary. In a high level, we merge two clusters when deleting data and split clusters when inserting data if certain condition is met. When merging clusters, we directly merge two clusters that share the same parent tree node and update the centroid. When splitting a cluster, we create two child nodes under the specific tree node, split the specific cluster into two clusters, and calculate the centroids.

If the number of points in a cluster is below certain threshold, we merge it with its neighbor. In contrast, if the variance of one cluster exceeds some threshold and the number of points exceeds certain number, we perform split operation since the quality of the cluster is low.

For example, as shown in Figure 5, the space is originally split into four clusters. After inserting some new points (in red color), since the variance inside cluster C_3 barely changes, the corresponding index remains untouched. In contrast, the variance inside cluster C_2 hugely changed. Thus we split this cluster and update the index for both old and new data.

5. POLYSTORE QUERY PROCESSING

To support multi-modal query, our system uses consistent IDs across different data stores, i.e., the same object has an unique ID for information stored in different places. The query is first decomposed into several sub-queries where each sub-query involves data only in a single store. Then these sub-queries are processed by their corresponding data stores. Next, the results of the sub-queries are aggregated. Each result corresponds to a set of object IDs. The results are intersected for conjunction queries and unioned for disjunction queries. Finally, the raw data corresponds to the object IDs in the final result set is returned from the file system with those projected attributions.

Joint Query Optimization. For disjunction multi-modal queries like "find similar objects within a specific period of

time", it may not be optimal to process all the sub-queries simultaneously and aggregate the results. This is because the similarity search sub-query on high dimensional data is time-consuming and usually becomes the bottleneck. Thus it may be much more efficient if the other sub-queries first efficiently identify a small set of objects as candidates and then the similarity search sub-query processes the small set of candidates right after. Based on this observation, SABER designs a joint query optimization module. It uses a cost model to estimate the cost of different sub-queries and determines the optimal execution order of the sub-queries.

Development Plan. We are still in the initial stage of system implementation. The plan is to get a prototype ready by next January. If this paper is accepted, we will demo SABER during the conference presentation.

6. RELATED WORK

`BigDAWG` [6] is a polystore system. Silva et al. [19] propose `SimDB` to support similarity query processing in relational DBMS. Mu et al. [17] propose to index high dimensional data in `ElasticSearch` so as to support multi-modal query. However, all these systems have poor performance and scalability in processing high dimensional data. Silva et al. [20] and Tang et al. [21] give several similarity groupby definitions on numerical and multi-dimensional data, which is different from our similarity groupby definition on high dimensional data. Xu et al. [23] proposed an index structure that can handle index updates. However, it only updates the index of the newly arrived data and has constraints on the distribution of new data to guarantee the accuracy of the updated index, which is unstable with large change of data distribution.

7. CONCLUSION

In this paper, we present the design of our end-to-end high dimensional data management system SABER. It consists of data ingestion, storage management, index management, and polystore query processing components. The data ingestion module extracts high dimensional feature vectors from the raw input data. The high dimensional data, along with their metadata, are stored in multiple different data stores. The polystore query processing component coordinates these data stores to process the multi-modal query. We quantize the dense high dimensional vectors into codes using product quantization and index the codes using a tree index structure. For the sparse high dimensional vectors, we employ the inverted index structure and prefix filtering techniques to process queries. We also design strategies to deal with data and index updates.

8. REFERENCES

[1] F. André, A. Kermarrec, and N. L. Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. *PVLDB*, 9(4):288–299, 2015.

[2] A. Babenko, A. Slesarev, A. Chigorin, and V. S. Lempitsky. Neural codes for image retrieval. In *ECCV*, pages 584–599, 2014.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[4] Q. Cao, Y. Ying, and P. Li. Similarity metric learning for face recognition. In *ICCV*, pages 2408–2415, 2013.

[5] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[6] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The bigdawg polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[7] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB*, 12(5):461–474, 2019.

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

[9] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864, 2016.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[11] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[12] P. Li, A. B. Owen, and C. Zhang. One permutation hashing. In *NIPS*, pages 3122–3130, 2012.

[13] Y. Liu, H. Cheng, and J. Cui. PQBF: i/o-efficient approximate nearest neighbor search by product quantization. In *CIKM*, pages 667–676, 2017.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[15] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[17] C. Mu, J. Zhao, G. Yang, J. Zhang, and Z. Yan. Towards practical visual search engine within elasticsearch. In *SIGIR 2018 Workshop On eCommerce*, 2018.

[18] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann series in data management systems. Academic Press, 2006.

[19] Y. N. Silva, A. M. Aly, W. G. Aref, and P. Larson. Simdb: a similarity-aware database system. In *SIGMOD*, pages 1243–1246, 2010.

[20] Y. N. Silva, W. G. Aref, and M. H. Ali. Similarity group-by. In *ICDE*, pages 904–915, 2009.

[21] M. Tang, R. Y. Tahboub, W. G. Aref, M. J. Atallah, Q. M. Malluhi, M. Ouzzani, and Y. N. Silva. Similarity group-by operators for multi-dimensional relational data. In *ICDE*, pages 1448–1449, 2016.

[22] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[23] D. Xu, I. W. Tsang, and Y. Zhang. Online product quantization. *TKDE*, 30(11):2185–2198, 2018.