DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search

Runhui Wang
Rutgers University
Department of Computer Science
runhui.wang@rutgers.edu

Dong Deng*
Rutgers University
Department of Computer Science
dong.deng@cs.rutgers.edu

ABSTRACT

High dimensional data is ubiquitous and plays an important role in many applications. However, the size of high dimensional data is usually excessively large. To alleviate this problem, in this paper, we develop novel techniques to compress and search high dimensional data. Specifically, we first apply vector quantization, a classical lossy data compression method. It quantizes a high dimensional vector to a sequence of small integers, namely the quantization code. Then, we propose a novel lossless compression algorithm, `DeltaPQ`, to further compress the quantization codes. `DeltaPQ` organizes the quantization codes in a tree structure and stores the differences between two quantization codes rather than the original codes. Among the exponential number of possible tree structures, we develop an efficient algorithm, whose time and space complexity are linear to the number of codes, to find the one with optimal compression ratio. The approximate nearest neighbor search query can be processed directly on the compressed data with small space overhead in a few bytes. Many similarity measures can be supported, such as inner product, cosine similarity, Euclidean distance, and Lp-norm. Experimental results on five large-scale real-world datasets show that `DeltaPQ` achieves a compression ratio of up to 5 (and often greater than 2) on the quantization codes whereas the state-of-art general-purpose lossless compression algorithms barely work.

PVLDB Reference Format:
Runhui Wang and Dong Deng. DeltaPQ: Lossless Product Quantization Code Compression for High Dimensional Similarity Search. *PVLDB*, 13(13): 3603-3616, 2020.
DOI: https://doi.org/10.14778/3424573.3424580

1. INTRODUCTION

High dimensional data is ubiquitous and plays an important role in many applications such as natural language processing [28, 72, 67], multimedia databases [19, 87, 68], and knowledge-base management [58, 40, 43]. For example, the semantics of images, videos, documents, and knowledge can be captured by meaningful high dimensional feature vectors extracted from deep learning models. However, the amount of high dimensional data is usually excessively large. For example, `YouTube-8M` [7] contains 1.4 billion feature vectors extracted from 350,000 hours of video using the `Inception-V3` deep neural network model [81]. In comparison, as of May 2019, the length of new video uploaded to YouTube per day is 720,000 hours [8]. As another example, on the one hand, `SIFT1B` [4] contains 1 billion Scale-Invariant Feature Transform (SIFT) [61] feature vectors extracted from 1 million images. On the other hand, TinyEye, a reverse image search engine, has around 40 billion images indexed to date [5].

A common practice in machine learning is to standardize feature vectors by feature scaling, mean normalization, dimension reduction, etc, which can reduce its storage space on disk or in memory. After these steps, feature vectors become dense vectors (typically from hundreds to thousands dimension). However, the size of dense vectors is still too large. For example, one 1,024-dimensional dense vector takes 4KB (assuming each floating-point is 32 bits). The 1.4 billion vectors in `YouTube-8M` could take over 5TB in storage.

Vector and Product Quantization. To further compress the dense vectors, quantization-based methods [37] can be used. For example, in the `YouTube-8M` dataset, all the 32-bit floating-points are quantized to 256 distinct values in each dimension, i.e., the floating point value is replaced by an integer in $[1, 256]$ denoting the index of the 256 distinct values, which takes only $\log 256 = 8$ bits. Thus the 1.4 billion feature vectors in `YouTube-8M` are compressed to 1.4TB.

Product quantization extends the above idea. It evenly partitions the original d dimensional space to m subspace and the dense vectors are accordingly partitioned to a few subvectors, one for each subspace. In each subspace, product quantization quantizes the subvectors in this subspace to l centroids learned by k-means clustering [60] (i.e., each subvector is quantized to its nearest centroid). In this way, each dense vector can be represented by a sequence of m integers in the range of $[1, l]$ using $m \log l$ bits, namely the product quantization codes. In a common setting where $l = 256$ and $m = 8$, the `YouTube-8M` dataset can be significantly compressed to 11.2GB only.

Lossless Code Compression. To manage large-scale high dimensional data, in this paper, we propose to further compress the quantization codes losslessly and perform the nearest neighbor search directly on the compressed data. A classical lossless data compression technique, differential compression, can be applied. It takes as input source data and

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 13, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3424573.3424580

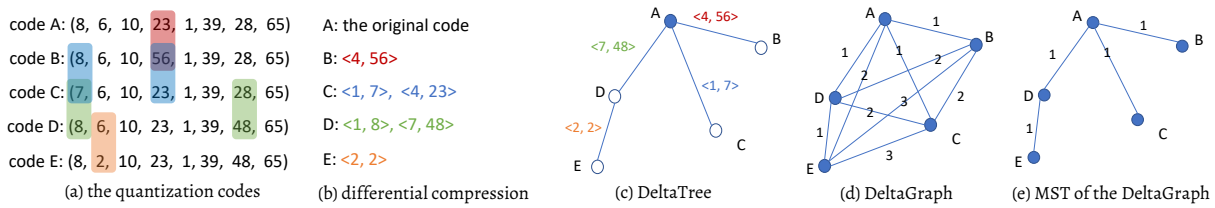


Figure 1: Examples of lossless quantization code compression.

target data and produces the difference data such that given the source data and the difference data, one can reconstruct the target data. For example, Figure 1(a) shows five codes. Among them, codes A and B only differ in the 4-th coordinate. All the other coordinates are exactly the same. We can use a tuple $\langle i, x\rangle$ to denote a difference which means that the i-th coordinate is x in the target code. As an example, as shown in Figure 1(b), the target code B has one difference $\langle 4, 23\rangle$ from the source code A, which means that the 4-th coordinate is 56 in B while all the other coordinates are the same as in A. Each difference takes only $\log m + \log l$ bits as $i \in [1, m]$ and $x \in [1, l]$. In comparison, each original code takes $m \log l$ bits. As long as the number of differences is small, storing the differences takes less space than storing the entire code.

To compress a collection of codes, we first sort them in an arbitrary order. Then for every pair of adjacent codes, their differences are calculated and stored. Finally, we can drop all the codes except for the first one. Clearly, all the codes can be losslessly reconstructed with the first code and all the differences. Figures 1(a) and (b) show such an example.

Note that the order of the codes can impact the total number of differences and compression ratio. A natural question to ask is "*how to achieve the minimum total number of differences?*". We observe that differential compression leverages only the "similarities" between two adjacent codes. However, one code can be "similar" to many codes. For example, in Figure 1(a), the code A is similar to codes B, C, and D (each of them has only 1 difference with code A) at the same time. To unleash the full potential of differential compression, we propose `DeltaPQ`.

Differential Tree Compression. `DeltaPQ` encodes all the codes in a tree, which we call the DeltaTree. Formally, given a collection of codes, there is a one-to-one correspondence between the tree nodes and the codes. For the root node, the original code is stored. For every other node, the differences between this node and its parent node are stored. For example, Figure 1(c) shows a DeltaTree for the five codes in Figure 1(a). In total, only 4 differences are stored in this DeltaTree while in differential compression 6 differences are stored. Obviously, all the codes can be losslessly reconstructed by traversing a DeltaTree from its root.

Clearly, there are an exponential number of all possible DeltaTrees for a collection of n codes. It is rather challenging to find the optimal one with the minimum number of differences (thus the highest compression ratio). We will show the optimal DeltaTree can be derived from the minimum spanning trees (MST) of a complete graph with all the codes as the vertexes, as illustrated in Figures 1(d) and 1(e). Note that there are n^2 edges in the complete graph and classical MST generation algorithms take $O(n^2)$ time, which is prohibitively expensive for a large n. To alleviate this problem, we develop an algorithm with time and space complexities linear to n to generate the MST and the optimal DeltaTree. Specifically, the time and space complexity are $O(2^m n)$ and $O(n)$ respectively. Note that m is typically a small integer (in most of existing work [12, 66, 59, 65, 51] it is set to 8).

Similarity Search Directly on Compressed Data. One of the most important operations on dense vectors is similarity search (a.k.a., nearest neighbor search): given a query vector, find the data vectors that are most similar to the query vector. Due to the "curse of dimensionality" [48], exact algorithms that guarantee to return completely accurate results are not competitive with the brute-force method for high dimensional dense vectors. For example, the R-tree [41] and K-D [21] tree indexes become even slower than the simple linear scan when data dimensionality is larger than 10 [78]. Thus, people resort to approximate algorithms, which is usually good enough in most applications.

Existing approximate algorithms can be broadly classified into three categories, the Locality Sensitive Hashing (LSH) based [48, 10, 47, 35, 82, 83, 80, 11, 26, 38, 62], the proximity graph based [64, 33], and the product quantization based [53, 59, 66, 51, 12, 36]. Methods in the first two categories often incur huge space costs. Specifically, LSH employs specific hash functions that are more likely to place similar vectors to the same bucket than dissimilar vectors. However, to achieve high accuracy, LSH based methods usually involve an excessively large number of hash tables and incur huge index size and suboptimal efficiency [82, 83]. In proximity graphs, each vertex corresponds to a data vector. Nearby vectors are connected such that for any vertex, one of its neighbors must be closer to the query than itself. Then a best-first search (such as A* search [75]) on the proximity graph can find the nearest neighbor to a query [64]. However, building and storing such a graph is very expensive. Furthermore, LSH and proximity graph based methods all need to consult the original dense vectors during query processing. Thus, they are not ideal for searching large-scale dense vectors. In contrast, a nice property of product quantization is that similarity search can be performed directly on the quantization codes without accessing the original dense vectors. Our proposed approximate algorithm in this paper searches the quantization codes in the DeltaTree and takes only $O(m)$ space overhead for each query.

In summary, we make the following contributions.

- We propose to compress the quantization codes by storing their difference in a DeltaTree and formalize the optimal DeltaTree construction problem.
- We develop an algorithm to construct the optimal DeltaTree in $O(2^m n)$ time and $O(n)$ space.
- Approximate nearest neighbor search can be performed directly on compressed codes with tiny space overhead.
- Experimental results show that our approach can often achieve a compression ratio greater than 2 and up to 5 on five large-scale real-world datasets.

2. PRELIMINARIES

2.1 Background: Product Quantization

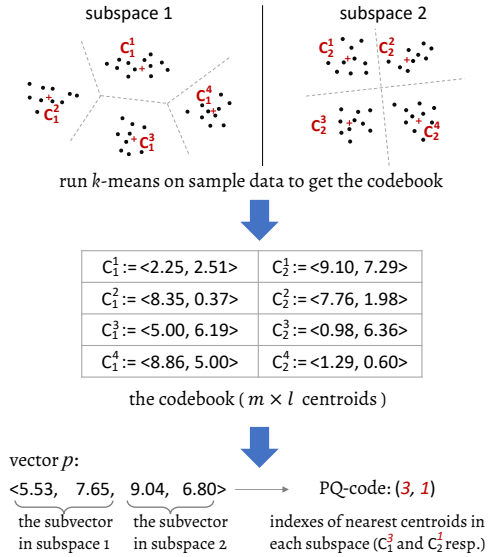


Figure 2: Quantization, codebook, and code.

We first introduce some terms in product quantization (PQ). PQ evenly partitions the original space into m *subspace*. Then, all (data and query) vectors are accordingly divided into m *subvectors*. In each individual subspace, all the subvectors are quantized to l distinct centroids, which are learned from a sample of data using clustering algorithms (e.g., k-means). That is to say, the subvector is represented by the index of its nearest centroid in the same subspace, which is an integer in $[1, l]$. Then, each original vector is represented by a tuple of m integers, which is called its quantization code (*code* for short). All the $m \times l$ centroids collectively compose the *codebook* of the product quantizer.

Formally, PQ evenly partitions the d dimensional original space into m sub-spaces. Thus each is $\frac{d}{m}$ dimensional[1]. Each vector p is accordingly partitioned into m subvectors $p^1, p^2, \ldots, p^m$. Let the l centroids in the j-th sub-space be $c_1^j, c_2^j, \ldots, c_l^j$. The subvector in the j-th subspace is quantized to (and approximated by) its nearest centroid among the l centroids. For ease of representation, we denote the centroid nearest to p^j as $c(p^j)$ and its index $a(p^j)$ is determined by the following assignment function:

$$a(p^j) = \arg\min_{x \in \{1,2,\cdots,l\}} dis(p^j, c_x^j)$$

where *dis* is the Euclidean distance. The code of a vector p is the tuple $\langle a(p^1), a(p^2), \cdots, a(p^m)\rangle$ and is denoted as $a(p)$.

EXAMPLE 1. *As shown in Figure 2, the original space is $d = 4$ dimensional, which is partitioned to $m = 2$ subspace by PQ. $l = 4$ centroids are learned for each subspace, which are numbered from 1 to 4. Thus, there are in total 8 centroids in the codebook as shown in the middle. The subvectors and centroids are all $\frac{d}{m} = 2$ dimensional. In the bottom, the vector p is accordingly divided into 2 subvectors. In the first subspace, the subvector is $[5.53, 7.65]$ and its nearest centroid is c_1^3 (among c_1^1, c_1^2, c_1^3, c_1^4). In the second subspace, the subvector is $[9.04, 6.80]$ and its nearest centroid is c_2^1. Thus, this product quantizer quantizes the vector p to a code $a(p) = \langle 3, 1\rangle$ as the indexes of c_1^3 and c_2^1 are 3 and 1 respectively in their subspace. Note that with the code $\langle 3, 1\rangle$ and the codebook, we can approximately reconstruct the original vector $p = \langle 5.53, 7.65, 9.04, 6.80\rangle$ as $\langle 5.00, 6.19, 9.10, 7.29\rangle$.*

[1]For ease of presentation, we assume d can be divided by m and both m and l are a power of two. All the techniques in this paper remain applicable if d, m, or l is not the case.

Distance/Similarity Calculation. A vector can be approximately reconstructed from its code using the codebook. Moreover, the distance/similarity between two vectors can also be estimated from their codes. For example, the inner product (*inn*) of two vectors is the summation of the inner products of all their m subvectors. On the other hand, the inner product of two subvectors can be approximated by the inner product of their nearest centroids of one subvector and the nearest centroid of another subvector. Formally, the inner product of p to q can be estimated by:

$$inn(p,q) = \sum_{j=1}^{m} inn(p^j, q^j) \approx \sum_{j=1}^{m} inn(c(p^j), q^j) \approx \sum_{j=1}^{m} inn(c(p^j), c(q^j)).$$

The latter two are called the asymmetric distance (AD, the distance calculated by a vector and a code) and the symmetric distance (SD, the distance calculated by two codes). SD and AD are accurate estimations of $inn(q, p)$ in the original space with bounded distortion [51]. The error of SD is twice of AD's, so in practice, AD is usually used. In addition to inner product, product quantization supports many other measures, such as Euclidean distance, cosine similarity, and Lp-norm. Our techniques work seamlessly with all these measures. For ease of presentation, in this paper, we use inner product as an example.

Similarity Search using PQScan [51]. In a nutshell, given a query vector, PQScan scans all the codes, calculates the AD from each code to the query vector, and returns the one with the largest (or smallest, based on the measure used) AD as the nearest neighbor. As the inner product between the centroids and the query subvectors are heavily accessed, to achieve high performance, a $l \times m$ lookup table T is constructed beforehand where $T[i][j] = inn(c_i^j, q^j)$. Then the AD between any data subvector p^j and query subvector q^j can be found in the lookup table and we have

$$inn(p,q) \approx \mathrm{AD}(p,q) = \sum_{j=1}^{m} inn(c(p^j), q^j) = \sum_{j=1}^{m} \mathrm{T}[a(p^j)][j]. \quad (1)$$

Note that the lookup table T is usually tiny. In the common setting where $m = 8$ and $l = 256$, the lookup table has only 2048 floating points, which takes merely 8 KB. One codebook is maintained for a product quantizer, which contains $l \times m$ centroids of $\frac{d}{m}$ dimensional. It is 1 MB when $d = 1024$.

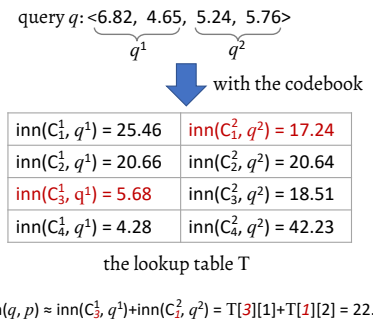


Figure 3: An example of query processing.

EXAMPLE 2. *Following Example 1, Figure 3 shows how to use the lookup table* T *to process a query. Given a query vector q, we first split it into 2 subvectors* q^1 *and* q^2*. Then we calculate the inner product of* q^1 *and each centroid in the first subspace in the codebook (the first column of the codebook) as well as the inner product of* q^2 *and each centroid in the second subspace in the codebook (the second column of the codebook) and store the results in the lookup table* T*. Then, to get the* AD *of q and a vector p (the p in Figure 2), we simply use its code* $a(p) = \langle 3, 1\rangle$ *to fetch the two precomputed inner products in* T*, and add them together to get the* AD *of 22.92, which is a close approximation of the real inner product inn (q, p) = 26.19.*

2.2 Problem Definition

In this paper, we study lossless code compression to further reduce space cost. In addition, we also study how to search the code with the smallest (or largest) AD to a given query vector. The two problems are defined as below.

DEFINITION 1 (LOSSLESS CODE COMPRESSION). *Given a collection of product quantization codes, the lossless code compression encodes the data such that the original codes can be perfectly reconstructed.*

DEFINITION 2 (NEAREST NEIGHBOR SEARCH). *Given a collection of product quantization codes, a lookup table, and a query vector, the nearest neighbor search finds the code with the smallest (or largest)* AD *to the query vector.*

We focus on generic measures that are supported by product quantization and do not cover any measure-specific optimization in this paper.

3. DIFFERENTIAL CODE COMPRESSION

3.1 Lossless Differential Code Compression

In this section, we discuss how to use differential compression to encode a collection of codes and search the one with the smallest (or largest) AD directly on the compressed data. Formally, we define a *difference* $\langle i, x, y\rangle$ as a tuple of three integers, which means that the i-th coordinate in the source code is x while it is y in the target code. For example, the target code B in Figure 1(a) has only one difference $\langle 4, 23, 56\rangle$ with the source code A, which means that the 4-th coordinate is 23 in A while it is 56 in B, and all the other coordinates are exactly the same. Clearly, given the source code and the difference, the target code can be restored losslessly by replacing the i-th coordinate with y in the source code for each difference $\langle i, x, y\rangle$.

For a collection of codes, we can calculate the differences between every pair of adjacent codes and store their differences in an array. Then we keep the first code and drop all the rest. For example, Figures 1(b) shows the differences calculated from the collection of codes in Figure 1(a) (note that as we will show shortly the second integer x in the difference are unnecessary and thus it is left out in the figures). With the first code and the array of differences, all the codes can be losslessly reconstructed sequentially.

3.2 Searching Differential Compressed Codes

Next, we show how to search the code with the smallest (or largest) AD to a query vector. A naive way is to reconstruct each code and use Equation 1 to calculate its AD to the query. However, we observe some common computation between the source code and target code can be shared. We illustrate our idea with an example. Consider two data vectors p and o and their codes $a(p) = \langle 22, 1, 17, 20\rangle$ and $a(o) = \langle 22, 1, 19, 20\rangle$. As discussed before, there is only one difference $\langle 3, 17, 19\rangle$ between the source code and the target code and their AD to a query vector q are respectively

$$\mathsf{AD}(p, q) = \mathrm{T}[22][1] + \mathrm{T}[1][2] + \mathrm{T}[17][3] + \mathrm{T}[20][4]$$

and

$$\mathsf{AD}(o, q) = \mathrm{T}[22][1] + \mathrm{T}[1][2] + \mathrm{T}[19][3] + \mathrm{T}[20][4],$$

which only differ in the third term. Clearly, we have

$$\mathsf{AD}(o, q) = \mathsf{AD}(p, q) - \mathrm{T}[17][3] + \mathrm{T}[19][3].$$

That is, with $\mathsf{AD}(p, q)$ and the difference $\langle 3, 17, 19\rangle$, we can obtain $\mathsf{AD}(o, q)$. In general, for any source code $a(p)$, target code $a(o)$, and every difference $\langle i, x, y\rangle$ of theirs, we have

$$\mathsf{AD}(o, q) = \mathsf{AD}(p, q) + \sum_{\langle i,x,y\rangle} (\mathrm{T}[y][i] - \mathrm{T}[x][i]) \qquad (2)$$

Clearly, based on Equation 2, with the differential compressed codes, we can calculate the AD from every code to the query and find the one with the smallest (or largest) AD.

Algorithm 1: DIFFERENTIALCODECOMPRESSION

Input: $\{a_1, a_2, \cdots, a_n\}$: a collection of n codes.
Output: differential compressed codes on disk.

```
write a_1 to the buffer;
foreach k in [2, n] do
    write a bitmap with m bits of 0 to the buffer;
    foreach difference ⟨i, y⟩ of a_{k-1} and a_k do
        set the i-th bit in the bitmap;
        write y to the buffer;
    flush the buffer to disk;
```

Algorithm 2: SEARCHDIFFCOMPRESSEDCODE

Input: differential compressed codes on disk;
q: a query vector; T: the lookup table.
Output: a: the code with largest/smallest AD to q.

```
read m integers of log l bits long each and use them
  as the source code a;
calculate AD = AD(a, q) using Equation 1;
while not reaching EOF do
    read a bitmap of m bits;
    for the i-th bit that is set in the bitmap do
        read an integer y of log l bits long;
        AD = AD − T[a[i]][i] + T[y][i];
        set a[i] as y;
    return a if AD is smallest/largest;
```

3.3 Storing Differential Compressed Codes

Next, we discuss how to store the compressed data physically. Our goal is to compress the code as small as possible while guarantee that they can be losslessly restored.

First of all, the second integer x, which is a part of the source data, in a difference $\langle i, x, y\rangle$ is unnecessary to store. We can maintain a source code (which initially is the first code) and keep it up to date by replacing the i-th coordinate with y for every difference $\langle i, y\rangle$. The second integer x can be

found in the i-th coordinate in the up-to-date source code. In this way, each difference takes only $\log m + \log l$ bits as $i \in [1, m]$ and $y \in [1, l]$.

To be self-contain, for each pair of adjacent codes, we need to store the number s of their differences and the list of differences. Note that $s \in [0, m]$. Thus, we can use $\log(m+1)$ bits to represent s and $\log m + \log l$ bits for each difference. Alternatively, we can use a bitmap with m bits to record which coordinates are different. In this way, we can drop the first integers i in all the differences. Moreover, the number s of differences can be inferred from the bitmap (which is the number bits that are set) and thus can also be dropped. In this way, we only need $m + s \log l$ bits rather than $\log(m+1) + s(\log m + \log l)$ bits.

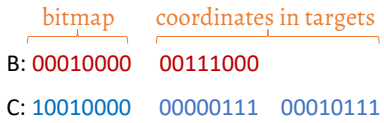


Figure 4: Storing differential compressed codes.

EXAMPLE 3. *Figure 4 shows an example of how to store the two differential compressed codes B and C on disk. In the example, $m = 8$, $l = 256$, and original codes are in Figure 1(b). For code B, since it is only different from A by the 4th coordinate, we set the 4th bit, unset the rest bits, and store its 4th coordinate value 56 in its binary form 00111000. For code C, the 1st and 4th coordinates are different from code B, thus we set the 1st and 4th bit in its bitmap, unset the rest bits, and store the two coordinate values 7 and 23 in C in their binary forms 00000111 and 00010111 respectively. Since each bitmap takes up $m = 8$ bits, and each code takes $\log l = 8$ bits, the compressed codes B and C takes only 5 bytes in total, rather than 16 bytes for the original codes.*

Algorithms 1 and 2 show the pseudo-code of compressing codes by differential compression and searching directly on the differential compressed codes respectively. Both of them takes only $O(n)$ time where n is the number of codes.

4. DIFFERENTIAL TREE COMPRESSION

4.1 The DeltaTree

In differential compression, we store the differences between adjacent codes. The number of differences is proportional to the storage cost and the computation cost. A natural question to ask is "*how to achieve the minimum number of differences?*". We observe that "similar" codes tend to share many coordinates and have small number of differences. However, one code can be "similar" to many codes. For example, as shown in Figure 1(a), the code A is similar to B, C, and D simultaneously (each of them has only one difference with code A). In differential compression, A cannot be adjacent to all the three codes. To address this issue, we propose to store the differences in a *tree-structured index*, namely the DeltaTree, which we define as below.

DEFINITION 3 (DELTATREE). *Given one collection of codes, a DeltaTree has one tree node for each code. For the root node, the original code is stored. For every other tree node, the differences between this node and its parent are stored.*

For example, Figure 1(c) shows a DeltaTree for the five codes in Figure 1(a). In total, only 4 differences are stored in the DeltaTree instead of 6 differences in differential compression as shown in Figure 1(b). Clearly, we can restore all the codes by traversing the DeltaTree from the root node (we will discuss the details later).

For a collection of codes, there are enormous DeltaTrees. The DeltaTree with a smaller total number of differences has less storage and computation cost and a higher compression ratio. Next, we discuss which DeltaTree has the minimum number of differences and how to efficiently generate it.

Remark. Another possible solution is to encode the differences in a connected graph instead of in a tree, where each vertex corresponds to one code and each edge is labeled with the differences between the two codes on the two ends. With one original code on a vertex and all the labels on the edges, we can restore all the codes. However, if the graph has a cycle, we can always remove one of the edges in the cycle and still restore all the codes with the induced graph. On the other hand, if the graph has no cycle, by definition, an acyclic connected graph is a tree and the graph is a DeltaTree. Thus, the DeltaTree dominates this solution.

4.2 The Optimal DeltaTree

In this section, we discuss how to find the DeltaTree with the minimum total number of differences, which we call the *optimal DeltaTree* as defined below.

DEFINITION 4 (OPTIMAL DELTATREE). *Given a collection of codes, its optimal DeltaTree is the one with the minimum total number of differences in all its nodes.*

To find the optimal DeltaTree, we first identify the scope of all possible DeltaTree for a collection of codes. For this purpose, we give the concept of DeltaGraph as defined below.

DEFINITION 5 (DELTAGRAPH). *Given one collection of codes, their DeltaGraph is a complete graph with each code as a vertex. Each edge is weighted by the number of differences between the pair of codes on the two ends.*

For example, Figures 1(c) and (d) show the optimal DeltaTree and the DeltaGraph of the five codes in Figure 1(a).

Given a collection of n codes, we observe that there is a 1-to-n correspondence between the spanning trees of their DeltaGraph and their DeltaTrees. This is because, by definition, a DeltaTree is a tree where each node corresponds to one code, while a spanning tree of the DeltaGraph is also a tree where each vertex corresponds to a code. As there are n vertexes in a spanning tree and each one of them can be the root of a DeltaTree, there is a 1-to-n correspondence between the spanning trees of the DeltaGraph and the DeltaTrees.

Based on the discussion above, we can conclude that all possible DeltaTrees of a collection of codes can be generated from the spanning trees of their DeltaGraph. Moreover, by Definitions 3 and 5, the total weight of a spanning tree is exactly the same as the total number of differences in each of its n correspondent DeltaTrees. Clearly, the minimum spanning tree (MST) of the DeltaGraph has the minimum total weight and its n correspondent DeltaTrees are all optimal, as stated in Lemma 1.

LEMMA 1. *Given a collection of codes, the corresponding DeltaTrees of the minimum spanning tree of their DeltaGraph are all optimal DeltaTrees.*

We omit the formal proof due to space limits.

The DeltaGraph of n codes is a complete graph and has $O(n^2)$ edges. The classical algorithms for MST generation for the DeltaGraph take at least $O(n^2)$ time and space [73, 56, 74], which is prohibitively expensive for very large n. Next, we discuss efficient MST generation in a DeltaGraph and give an algorithm in $O(2^m n)$ time and $O(n)$ space.

4.3 MST Generation

In this section, we discuss how to generate the MST of the DeltaGraph efficiently. Our approach built upon the classical Kruskal algorithm [56], which we recap below.

The Kruskal Algorithm. Given a connected graph with n vertices, to generate its minimum spanning tree, the Kruskal algorithm first sorts all the edges by their weights. Then it accesses each edge in the ascending order of their weights and tries to add the edge to a graph E (which is an empty graph at the beginning). An edge is added to E iff. its two vertices are not in the same connected component in E. Once $n-1$ edges are added to E, the algorithm terminates, and E is guaranteed to be the minimum spanning tree. To efficiently test if two vertices are within the same connected component, a *disjoint-set* [34] data structure is employed. The disjoint-set is an array of n integers. It takes constant time[2] to test an edge [84].

The time complexity of Kruskal algorithm for our DeltaGraph is $O(n^2 \log n)$. It is prohibitively expensive to apply the Kruskal algorithm directly to generate the MST for our purpose. To avoid generating and sorting all the $O(n^2)$ edges in the DeltaGraph upfront, we propose to generate the edges on-the-fly and in the order of their weights. The generated edges will be checked by the disjoint-set. Only if approved, the edge will be stored (in adjacent lists). Our algorithm terminates once $n-1$ edges are approved and stored. In this way, we can avoid generating, storing, and sorting all the edges. Actually, at any time, we store at most $n-1$ edges. Next, we discuss how to generate the edges on-the-fly.

Incremental Edge Generation. Our key observation for the DeltaGraph is that the number of possible edge weights is limited. The weight is an integer in the domain of $[0, m]$, where m is a very small number (e.g, 8). Thus, we can enumerate every weight w in ascending order and in each round generate all the edges in the DeltaGraph with weight w. Since the batch of edges have the same weight w, they can be checked *in arbitrary order* by the disjoint-set. Next, we discuss how to generate all the edges with weight w.

Based on Definition 5, finding edges with weight w is equivalent to finding code pairs with w differences. For this purpose, we enumerate every combination of w coordinates and "mask" them in all the codes. Then the codes become exactly the same after masking must have at most w differences (which can only be in the masked coordinates). Moreover, two codes with w differences must be the same after masking their w differences. Thus, this approach can find all the code pairs with w differences. To mask a code, we can simply set the w coordinates in the code as zero. For this purpose, we create a mask bit array of the same length as the code, i.e., $m \log l$ bits. Then for each i-th coordinate in the combination, we unset the $(i-1)\log l + 1$ to $i \log l$-th bit in the mask. The rest of the bits are set. Then we can simply AND the mask bit array with the code, which results in the corresponding masked code.

[2]It takes $O(\alpha(n))$ time where $\alpha(n)$ is the inverse Ackermann function. This function has a value $\alpha(n) < 5$ for any value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time.

Algorithm 3: MSTGeneration

```
Input: {a_1, a_2, ..., a_n}: a collection of codes.
Output: G: the MST of the DeltaGraph of the codes.
1  foreach w ∈ [0, m] do
2      foreach combination of w numbers in [1, m] do
           // generating all edges of weight w
3          Create a mask for the w coordinates;
4          foreach code a_i do
5              Mask the w coordinates in a_i with mask;
6              HashMap[masked a_i].append(i);
           // check edges with disjoint-set
7          foreach list of IDs in HashMap do
8              foreach pair of IDs ⟨x, y⟩ in the list do
9                  check the pair ⟨x, y⟩ with disjoint-set;
10                 if approved then
11                     G[x].append(y);
12                     if n − 1 pairs are approved then
13                         return G;
```

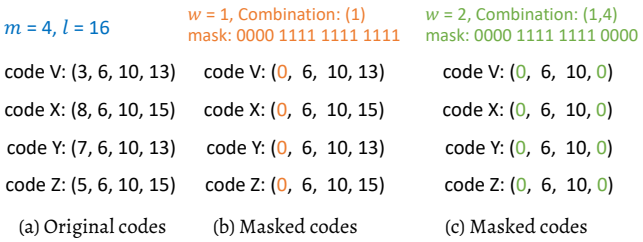


Figure 5: Differential compressed codes example.

Example 4. *Figure 5 shows an example of masking. In this example, $m = 4$, $l = 16$. In the first mask with $w = 1$, we mask the first coordinate and find two pairs of masked codes that are the same: (V, Y) and (X, Z). Thus, we generate two edges (V, Y) and (X, Z) whose weight are 1. We also show an example where $w = 2$. The combination is $(1, 4)$. We found all the masked codes become the same. Thus, we generate an edge for every pair of them with weight 2.*

The pseudo-code of the MST generation (including edge generation) is given in Algorithm 3. It takes a collection of codes as input and outputs the MST of their DeltaGraph. For each $w \in [0, m]$ in ascending order, it enumerates all the combinations of w numbers in $[1, m]$ (Lines 1 to 2). For each combination, the corresponding w coordinates of all codes are masked. The code is appended to a list of codes that share the same masked code. For this purpose, a hash map is employed where the masked code is the key field and the list of codes is the value field (Lines 2 to 5). Each pair of codes in the same list/value in the hash map have at most w differences. The edge between the pair of codes is generated and checked by the disjoint-set. If the edge is approved by the disjoint-set, it is added to the result G (Line 6 to 9). Finally, if there are $n-1$ edges in G, G must be the MST of the DeltaGraph of the input codes and will be returned (Lines 10 to 11).

Algorithm 4: MSTGeneration-SkipEdge

```
// Replace with Line 6 in MSTGeneration
1 For each adjacent pair of IDs ⟨x, y⟩ in the list;
```

Lemma 2. *Given a collection of codes,* MSTGeneration *produce an MST of the DeltaGraph of the codes.*

Complexity Analysis. For each combination of w coordinates, MSTGeneration spends $O(n)$ time to generate the masked codes. Each pair in a list is an edge and the list can be as long as $O(n)$. Thus $O(n^2)$ edges are generated and checked in each combination. The total number of combinations is 2^m at most for all $w \in [0, m]$. Thus, the time complexity is $O(2^m n^2)$. The space complexity is $O(n)$ for storing the n masked codes for each combination.

As discussed above, the number of edges generated and checked in each combination is quadratic to n, the number of codes. We observe that not all the edges in a list are required to be checked. Actually, it is sufficient to check only a very small portion of them to generate the MST.

4.4 Skipping Unnecessary Edges

Consider a list of codes that are exactly the same after masking a combination of w coordinates. In MSTGeneration, each pair of them are enumerated and checked if they are within the same connected component, which is unnecessary. We observe that, *for any pair of codes in this list, they must be within the same connected component after checking with disjoint-set.* This is because, on the one hand, if they are not in the same connected component, the disjoint-set will approve this pair and add the edge between them to E. On the other hand, if they are already in the same connected component, the disjoint-set will do nothing. In either case, they are eventually in the same connected component.

Based on the observation above, we can enumerate and check every pair of adjacent code in the list only. After this, all the codes in this list must be within the same connected component. Any other pair of codes in the list must be omitted by the disjoint-set as they are already in the same connected component. Thus, we can safely skip them.

The number of adjacent code pairs in a list is less than the length of the list. For each combination, the total length of all lists is exactly n. Thus, the above method generates and checks at most n edges (i.e., pairs of codes) for each combination. The total time complexity of MSTGeneration is thus reduced to $O(2^m n)$ while the space complexity remains the same. The pseudo-code is shown in Algorithm 4.

Theorem 1. *Given a set of codes,* MSTGeneration-SkipEdge *produces an MST of the DeltaGraph of the codes.*

4.5 The Fixed-Height Spanning Tree

As we will see later, the space overhead to process a query directly on a DeltaTree is proportional to the height of the DeltaTree. The optimal DeltaTree we constructed from the minimum spanning tree can be as high as $O(n)$. Given that n is a large number, the space overhead per query is unacceptable. To address this problem, in this section, we discuss how to generate a fixed-height spanning tree. Our empirical results show that the corresponding DeltaTree of the fixed-height spanning tree, whose height is bounded by $O(m)$, has the total number of differences only marginally larger than that of the optimal DeltaTree, while its space overhead per query is merely $O(m)$ in a few bytes.

Algorithm 5: FixedHeightSpanningTree

```
   Input: {a_1, a_2, ··· , a_n}: a collection of codes.
   Output: G: a fixed-height spanning tree.
1  Initialize an n-bit bitmap and a height array of n 1s;
2  foreach w ∈ [0, m] do
3      foreach combination of w numbers in [1, m] do
4          Create the HashMap as in MSTGeneration
             using a_i only if the i-th bit is set in the
             bitmap and a_i's height is no larger than
             w+1;
5          foreach ⟨k, v⟩ ∈ HashMap where v.len > 1
             do
6              Let a_j be the code in v with the highest
                 height;
7              foreach code a_i in v except a_j do
8                  G[j].append(i);
9                  Unset the i-th bit of the bitmap;
10             Increase a_j's height by 1 if there is
                 another code in v with the same height
                 as a_j;
11 return G;
```

To generate a fixed-height spanning tree, we treat each connected component in the result graph G as a tree. Initially, each code is a connected component itself and a root node. Then, we enumerate edges in DeltaGraph in the increasing order of their weights. However, we only generate edges between the root nodes of the connected components. For this purpose, we use an n-bit bitmap to record if a code is a root node in a connected component. When an edge is added to G, only one end of it remains a root node. The other end becomes a child of the root node. The bitmap is updated accordingly. In this way, each connected component is always a tree. Moreover, we add an edge with weight w to G only if it did not result in a tree with a height of more than $w + 2$. In this way, the spanning tree G we get has a maximum height of $w + 2$. To this end, we use an array to maintain the height of the subtree rooted at each code in G.

Algorithm 5 shows the pseudo-code of generating a fixed-height spanning tree. It takes a collection of codes as input and generates a spanning tree with maximum height $m + 2$. It first initializes a n-bit bitmap to record if a code is the root node of a connected component. In addition, an array of size n to record the height of each code is created. At the beginning, as every code itself is a connected component and root node, all bits are set in the bitmap and all heights are 1 (Line 1). Then the algorithm enumerates all edges with weights w between the root nodes whose height no larger than $w + 1$ by masking a combination of w coordinates (Lines 2 to 4). Then for each list v of codes which are identical after masking, we choose the one a_j with the highest height as the new root node and add to G an edge between a_j and every other node in v (Lines 5 to 8). The bitmap and the height array are updated correspondingly (Lines 9 to 10). Finally, G must be a spanning tree of the DeltaGraph with a maximum height of $m + 2$ and will be returned (Line 11). The complexity of this algorithm is identical to that of MSTGeneration.

Lemma 3. *Given a collection of codes,* FixedHeightSpanningTree *must generate a spanning tree of the DeltaGraph of the codes with a height of no more than* $m + 2$.

Algorithm 6: DifferentialTreeCompression

Input: G: a fixed-height spanning tree of DeltaGraph.
Output: The corresponding DeltaTree of G on disk.

1 Write the code a_i with highest height in G to disk;
2 DeepFirstTraverse(i, G);

Algorithm 7: DeepFirstTraverse(i, G)

Input: i : the parent; G: the spanning tree.

1 **foreach** $j \in G[i]$ **do**
2 write $is\_leaf = 0$ to disk if $G[j]$ is empty to indicate a_j is a leaf node (otherwise, write 1);
3 write $is\_the\_last\_child = 1$ to disk if j is the last element in $G[i]$ (otherwise, write 0);
4 write the differences of a_i and a_j to disk;
5 DeepFirstTraverse(j, G);

4.6 Storing and Searching the DeltaTree

From Fixed-Height Spanning Tree to DeltaTree. Once a fixed-height spanning tree is generated, we deduce its corresponding DeltaTree and store it on disk. Specifically, we traverse the spanning tree in pre-order (i.e., depth-first order) and store the differences in this order in the same format as in differential compression. That is storing an m-bit bitmap about the positions of the differences and the list of different coordinates. To maintain the tree information, for each node, we keep two boolean fields is_leaf and $is\_the\_last\_child$. The first field indicates if the tree node is a leaf node while the second field indicates if the node is the last child of its parent in the depth-first order.

Searching the DeltaTree. We sequentially scan the compressed codes in a DeltaTree and maintain a stack h_1 of source codes and a stack h_2 of AD between the source codes in h_1 and the query vector. The source code on top of h_1 is guaranteed to be the parent of the code that is currently visiting. Then same as in searching the differential compressed code, with the AD of the source code on top of h_2 and the differences, we can efficiently calculate the AD of the currently visiting code using Equation 2.

Next we discuss how to keep two stacks up to date using the two boolean fields is_leaf and is_last_child. Basically, if a node is a leaf node, it will not be the parent of any other code and we do not push its information to the two stacks. Right after visiting the last child of an internal node, we pop the two stacks. This is because the information of this internal node will not be referred anymore and the elements on top of the two stacks are information about the internal node since the currently visiting code is a child of the internal node. In this way, we can always keep the two stacks up to date.

Algorithms 6 and 7 show the pseudo-codes of the proposed algorithm `DeltaPQ`. It takes the fixed-height spanning tree generated by Algorithm 5 as input and generates the corresponding DeltaTree on disk. It first writes the root node in G to disk. Then the algorithm traverse G in depth-first order. For each node in G, it writes one bit to indicate if the node is a leaf node and another bit to indicate if the node is the last child of its parent (Lines 2 to 3). Then the differences of this node and its parent are also written to disk in the same format as in differential compression (Line 4).

Algorithm 8 gives the pseudo-code of searching a DeltaTree on disk. It first reads the root node of the DeltaTree and calculates its AD to the query. The two stacks are initialized with the root node and its AD (Lines 1 to 2). Then it reads 2 bits as the two boolean fields, m bits as the bitmap, and a sequence of $\log l$ bits as indicated by the bitmap. With the source code and its AD on top of the two stacks, the AD of currently visiting node is calculated (Lines 4 to 5). Finally, the two stacks are maintained in accordance with the two boolean fields (Lines 6 to 8) and the largest (or smallest) AD is returned after visiting all the nodes (Line 8).

Algorithm 8: SearchDeltaTree

Input: differential tree compressed codes DeltaTree on disk; q: the query; T: the lookup table.
Output: a: the code with largest/smallest AD to q.

1 read m integers of $\log l$ bits each as the source code a and push a to stack h_1;
2 calculate AD = AD(a, q) and push AD to stack h_2;
3 **while** *not reaching EOF* **do**
4 read 2 bits as is_leaf and $is\_the\_last\_child$;
5 read a bitmap of m bits and the differences from disk, let a be the one on top of h_1 and AD be the one on top of h_2, update a and AD same as in Algorithm 2;
6 **if** is_last_child *is* 1 **then** pop h_1 and h_2;
7 **if** is_leaf *is* 0 **then** push a to h_1 and AD to h_2;
8 **return** a if AD is smallest (or largest);

Table 1: The dataset details.

dataset	n	d	raw data size	quantized size
`BigAnn`	10^9	128	123GB	7.5GB
`Deep1B`	10^9	96	361GB	7.5GB
`Audio`	1.4×10^9	128	172GB	11GB
`Video`	1.4×10^9	1,024	1,341GB	11GB
`SALD`	0.89×10^9	128	429GB	6.7GB

Lemma 4. *Given a DeltaTree, a lookup table* T*, and a query vector* q*,* SearchDeltaTree *must return the code with the smallest (or largest) AD to the query.*

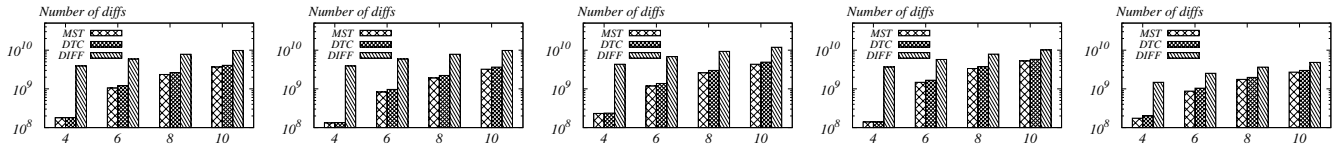
4.7 Update the DeltaTree

When a new vector arrives, we need to update the DeltaTree. The vector is firstly quantized to a code. Then we append the code to the existing DeltaTree as the last child of the root node. Though it does not lead to the minimum number of additional differences, the insertion operation can be performed efficiently. To process the deletion query, we employ a boolean column to mark the existence of each vector. A deletion query simply marks the existence of the corresponding code as false. Periodically, we rebuild the entire DeltaTree to optimize the space cost.

5. EXPERIMENTS

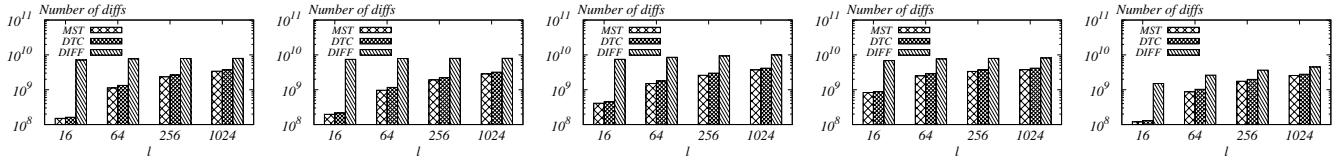
Datasets. All experiments were conducted on the following five real-world datasets.

- `BigAnn` [4] consists of 1 billion 128-dimensional SIFT vectors (a widely adopted image descriptor [61]) extracted from $\approx$ 1 million images. This dataset has been extensively used in similarity search studies [12, 66, 52, 53].
- `Deep1B` [16] is a billion-scale dataset that is obtained from the outputs of the last fully-connected layer of a deep neural network for a billion images on the Web. Each vector is compressed by PCA to 96 dimensions.
- `Video` and `Audio` [9, 7] are the frame-level video and audio features extracted from 350,000 hours of YouTube video



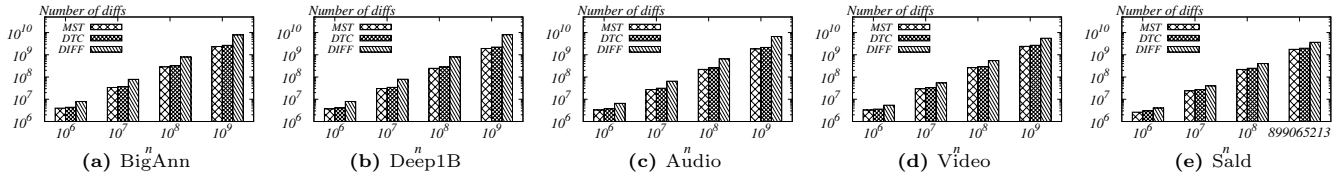
(a) BigAnn **(b)** Deep1B **(c)** Audio **(d)** Video **(e)** Sald

Figure 6: Number of differences by varying m



(a) BigAnn **(b)** Deep1B **(c)** Audio **(d)** Video **(e)** Sald

Figure 7: Number of differences by varying l



(a) BigAnn **(b)** Deep1B **(c)** Audio **(d)** Video **(e)** Sald

Figure 8: Number of differences by varying n

using Inception-V3 and VGG-like models. Each of them has around 1.4 billion feature vectors. These features are preprocessed by feature standardization, dimension reduction, and vector quantization. Values in each coordinate are quantized to 256 distinct values.

- `SALD` [3, 30, 31] contains neuroscience MRI data and consists of 899 million 128-dimension vectors.

Table 1 shows the detailed statistics of all datasets. Notably, the largest dataset contains 1.4 billion vectors and takes 1342 GB space on the disk (after vector quantization).

Settings. All of our proposed methods are implemented in C++ and compiled using g++ -std=c++11 -fopenmp with -O3 flag. All compressing experiments are conducted on a shared cluster of machines with Centos 7, Intel(R) Xeon(R) Gold 6230 CPUs @ 2.10GHz, and 192 GB memory. The query experiments are performed on a dedicated machine with Ubuntu 18.04, Intel(R) Xeon(R) Gold 6212U CPU @ 2.40GHz, 64 GB memory, and an 8TB hard disk.

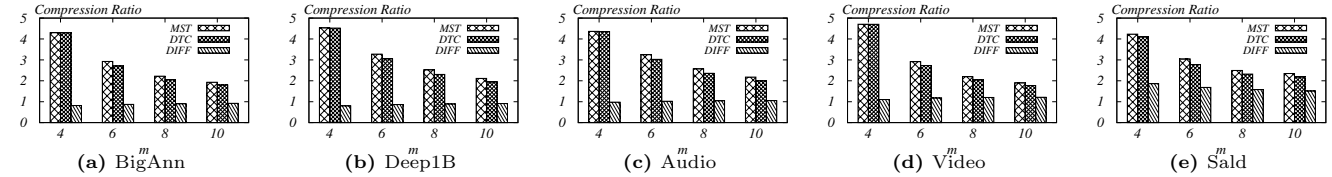
5.1 Evaluating Our Compression Methods

We first evaluate our proposed lossless compression methods. Three methods are evaluated: (1) DIFF compresses the codes by differential compression as discussed in Section 3; (2) MST encodes the codes in an optimal DeltaTree as discussed in Section 4.2; (3) DTC encodes the codes in the DeltaTree derived from the fixed-height spanning tree of the DeltaGraph as discussed in Section 4.5. We vary the three parameters, m, l, and n, and report the number of differences, the compression time, and the compression ratio of the three methods. Each time we vary one of the three parameters and fix the other two as their default values. Specifically, m varies in $[4, 6, 8, 10]$. l varies in $[16, 64, 256, 1024]$. n varies in $[10^6, 10^7, 10^8, 10^9]$. The default value is 8 for m, 256 for l, and the maximum n of each dataset (as shown in Table 1). Note that the default setting of m and l is a common trade-off between query accuracy and efficiency in many existing studies [12, 65, 66, 52, 53]. The experimental results are shown in Figures 6, 7, 8, 9, 10, 11, 12, 13, 14.

Number of Differences. As shown in Figures 6, 7, and 8, DIFF had the maximum number of differences among the three methods, MST achieved the minimum total number of differences, while the number of differences of DTC was only marginally larger than that of MST. This means our compression method `DeltaPQ` almost achieves the optimal compression ratio among all DeltaTrees. For example, in the default setting, the number of differences for MST, DTC, and DIFF were respectively 1.9×10^9, 2.2×10^9, and 7.9×10^9 on `Deep1B` dataset. In addition, we observe that with the increase of m and l, all the methods had a greater number of differences, and the gap between DIFF and MST (DTC) decreased. This is because the code space m^l (the number of possible codes) grows when m and l increase and the chance that two codes sharing common coordinates decreases. Thus, the number of differences increases. In addition, DIFF almost had the maximum possible number of differences even when m and l were small and could not become worse with the increase of m and l. The number of differences grows with the input size n.

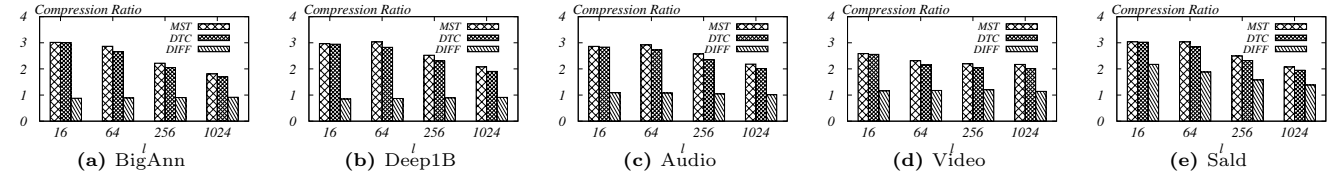
Compression Ratio. Figures 9, 10, and 11 report the compression ratio of the three methods under different parameters. The compression ratio is calculated as the size of the original codes divided by the size of the compressed data and the larger the better. We can see the compression ratio had the same trend as the total number of differences, which is consistent with our analysis. Note that the compression ratio of DTC is only marginally smaller than MST and is greater than 2 in almost all cases. The compression ratio of DIFF was even smaller than 1, meaning the compressed data is larger than the original codes. This is because DIFF has too many differences and needs to store a bitmap to record the position information of the differences. For example, in the default setting, the compression ratio of DTC, MST, and DIFF were respectively 2.57, 2.36, and 1.04 on the `Audio` dataset. In addition, as the number n of codes grows, the compression ratio tends to increase. For example, with $n = 10^6$ on `Deep1B`, the compression ratio of DTC was 1.4, while when n increased to 10^9, it improved to 2.1. This is because, with the increase of input size, the codes are more likely to be connected with lighter weight edges.

Compression Time. The compression time is shown in Figures 12, 13, and 14. DIFF had very small compression time as it only needs to scan all the codes once and compare



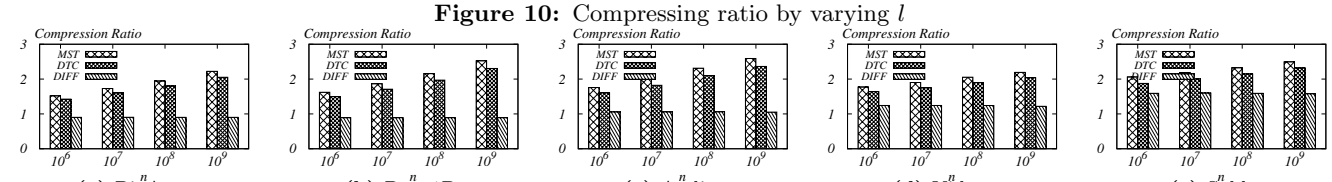
(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 9: Compression ratio by varying m



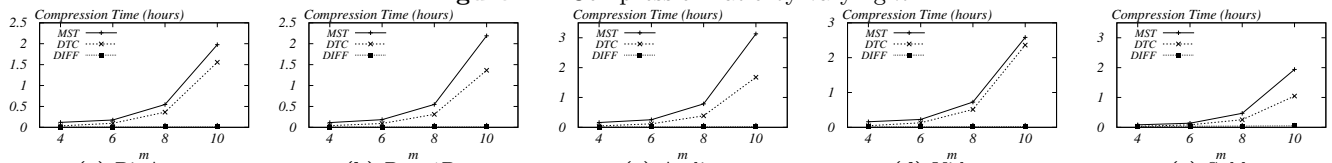
(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 10: Compressing ratio by varying l



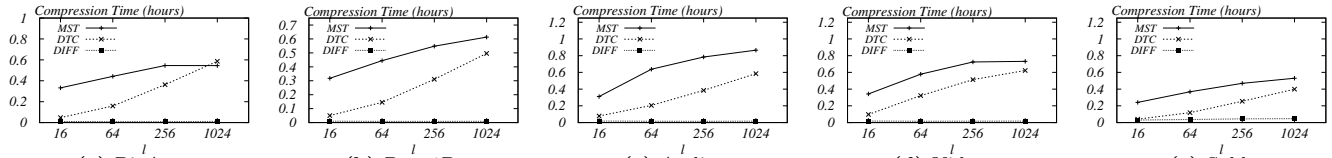
(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 11: Compression ratio by varying n



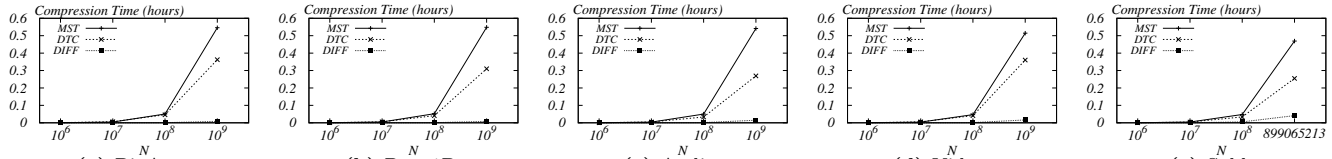
(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 12: Compressing time by varying m



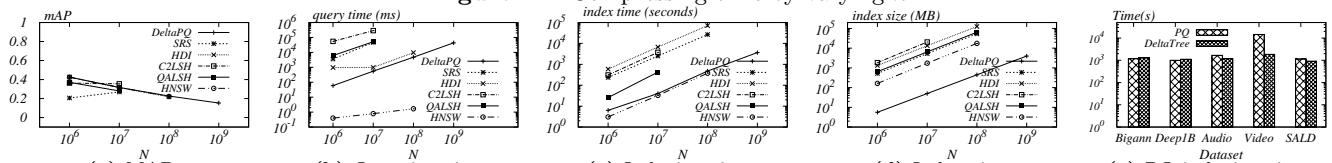
(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 13: Compressing time by varying l



(a) BigAnn (b) Deep1B (c) Audio (d) Video (e) Sald

Figure 14: Compressing time by varying n



(a) MAP (b) Querying time (c) Indexing time (d) Index size (e) PQ indexing time

Figure 15: Comparison with the state-of-the-art similarity search algorithms

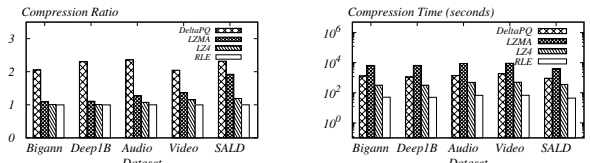
the adjacent codes. MST and DTC had longer compression time. We can see from the figures that the compression time of MST and DTC grew exponentially with the increase of m and grew linearly with the increase of n. This is consistent with our complexity analysis. Note that DTC was around 2× faster than MST in most cases. This is because DTC generates less edges than MST. It only generates edges between codes that are root nodes in connected components and whose height is small. However, DTC took more time than MST on `BigAnn` for $l = 1024$. This is because, for a large l, the common coordinates become rare and fewer edges can be generated. Thus, most of the codes remain root nodes with small height and DTC needs to generate edges between all of them. Besides, DTC needs to maintain the root node bitmap and height array. This is also why the compression time grew marginally with an increase of l.

Summary. DTC achieved a similar compression ratio as MST, up to 5 when m and l were small and took only half of the compression time. DIFF barely compressed the codes.

5.2 Comparing with Compression Algorithms

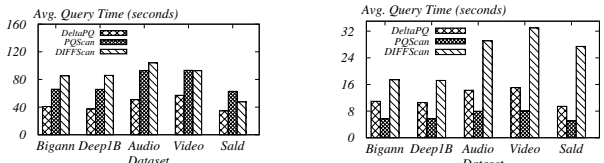
In this subsection, we compare DeltaPQ with three state-of-the-art general-purpose lossless compression algorithms.

- RLE Run-Length-Encoding [46] is a lossless compression method widely used in column-oriented database systems.
- LZMA Lempel-Ziv-Markov chain algorithm [71] is a lossless compression method optimized for compression ratio.



(a) Compression ratio **(b)** Compression time

Figure 16: Compression performance



(a) On-disk **(b)** In-memory

Figure 17: Search Performance

- LZ4 [25] is another lossless data compression algorithm that is optimized for compression and decompression time.

We downloaded standard implementations of the above algorithms from the official websites or GitHub repositories [2, 71, 6]. All of them are implemented in C++. The compression ratio on the five datasets with default parameters is reported in Figure 16a. As we can see from the figures, DeltaPQ outperformed LZMA, LZ4, and RLE on all of the datasets. The three general-purpose compression algorithms barely work. For example, on the `Deep1B` dataset, the compression ratio of DeltaPQ, LZMA, LZ4, and RLE were respectively 2.35, 1.11, 1.01, and 0.998. This is the state-of-the-art algorithms search for repetitive patterns in the data and compress them by keeping only one copy of each repetitive pattern. However, the quantization codes rarely have repetitive patterns. On the contrary, DeltaPQ leverages the common coordinates to compress codes, which there are a lot because the domain of each coordinate is very limited.

Figure 16b shows the compression time on the five datasets with the default setting. LZ4 and RLE compressed data at a very high speed, as fast as 50 seconds per billion codes, yet they did not compress the data well and their compression ratios were close to 1, which means they barely reduce the file size after compression. Although LZMA achieved a slightly higher compression ratio than LZ4 and RLE, its compression time was orders of magnitude longer. Compared to LZMA, DeltaPQ achieved a much better compression ratio with an order of magnitude less compression time.

Update Query. We now evaluate the performance of update queries. We reported our experimental results on the `BigAnn` dataset and omit others due to space limit, though similar results are observed. The update query vectors are randomly chosen from the original dataset. Since we only need to append the new quantization code to the end of the DeltaTree with at most 2 IOs, the average update time is only 88ms. The increased compression ratio grows linearly with the number of insertion queries. For example, with 1 million, 10 million, and 100 million insertion queries, the compression ratio increases by 0.00237, 0.0237, and 0.237.

5.3 Similarity Search

Searching by Scanning PQ codes. We compared three methods: (1) DIFFScan searches directly on the differential compressed codes as discussed in Section 3; (2) DeltaPQ searches directly on the DeltaTree derived from the fixed-height spanning tree as discussed in Section 4.5; (3) PQScan searches on the uncompressed codes. All the compressed (or uncompressed) codes are on disk. We did not include the method that searches on the optimal DeltaTree as it has no bound on the space overhead per query. In contrast, the three methods above all have very small space overhead in a few bytes per query. We reported the average query time on the five datasets with the default setting in Figure 17a. DeltaPQ achieved the best query performance while PQScan outperformed DIFFScan. For example, the average query time for DeltaPQ, PQScan, and DIFFScan on the `Audio` dataset were respectively 51, 93, and 104 seconds. The performance boost is mainly due to the reduced size of the compressed data and thus the reduced I/O cost. DIFFScan not only had a higher I/O cost but also had a higher CPU cost compared to PQScan. We also compared the average query time when all codes are in memory. As shown in Figure 17b, DeltaPQ takes less than twice time than PQScan, mainly because of the high branch misprediction penalty at the CPU architecture level caused by the unpredictable node depth during the scan. DIFFScan is slower than DeltaPQ because it executes up to 4 × more arithmetic operations. In comparison, scanning the raw data takes 740s, 2190s, 1040s, 8070s, and 2420s for `BigAnn`, `Deep1B`, `Audio`,`Video`, and `SALD` respectively, which are two orders of magnitude slower.

Table 2 shows the search accuracy of PQ-based methods on the entire datasets and their one million sample. Recall and ε-Recall, which are also used in the benchmark [1], are reported, along with the mean average precision (MAP) [13]. ε-Recall is the percentage of results whose distances are within $1+\varepsilon$ times of that of the k-th real nearest-neighbor. The accuracy is lower in large datasets than in small datasets as vectors in large datasets are more dense and more indistinguishable. The accuraccy is higher in datasets of lower dimensionalities as the quantization error is smaller.

Comparing with State-of-the-art Similarity Search Methods. Next, we compared our method with five state-of-the-art approximate nearest neighbors search algorithms HDIndex [13], C2LSH [35], QALSH [47], SRS [80], and HNSW [64]. HDIndex is an on-disk tree-based method. C2LSH, QALSH and SRS are on-disk LSH-based methods. HNSW is an in-memory graph-based method. We run these algorithms on `BigAnn` and scale the input size from 10^6 to 10^9. We tune parameters to make their search accuracy roughly the same (as evaluated by Mean Average Precision MAP [30, 31, 13, 36]) and report their query time and index cost

As shown in Figure 15, DeltaPQ answers top-100 queries with MAP of 0.42, 0.32, 0.23 and 0.15 for Sift1M, Sift10M, Sift100M and Sift1B respectively. In comparison, none of the state-of-the-art methods scales to billion scale and gets similar accuracy within a reasonable indexing time and size. HDIndex requires more than 10 days and 1TB disk space to build the index, while its query latency is at least an order of magnitude longer than DeltaPQ. This is because HDIndex incurs randomized disk access for building multiple huge B-Trees on disk. In comparison, DeltaPQ only needs less than 1 hour for preprocessing and less than 4GB for storage. Compared to DeltaPQ, the index sizes of SRS, C2LSH, and QALSH are more than two orders of magnitude larger, and the indexing time, as well as query latency are at least one order of magnitude longer. This is because LSH-based methods require an excessively large number of hash tables in order to match the accuracy of DeltaPQ. HNSW answers query the fastest as it is an in-memory method while all the others are disk-based. HNSW requires slightly less time for indexing in the experiment because the number of neighbors of each vertex only needs to be 2 to achieve the same ac-

Table 2: Search Accuracy of PQ-based Methods (ε=0.1)

dataset	ε-Recall 1M	ε-Recall 1B	Recall 1M	Recall 1B	MAP 1M	MAP 1B
BigAnn	0.88	0.52	0.48	0.19	0.43	0.15
Deep1B	0.70	0.25	0.29	0.08	0.24	0.08
Audio	0.67	0.27	0.41	0.12	0.35	0.11
Video	0.31	0.15	0.21	0.09	0.24	0.13
SALD	0.90	0.60	0.56	0.15	0.46	0.13

curacy as our method, which largely reduces the indexing time. However, even for a very small number of neighbors of 2, its index size is still at least 30× larger than DeltaPQ.

Indexing Time. As shown in Figure 15e, the quantization time of PQScan is similar to the preprocessing time (including quantization and compression) of DeltaPQ on all datasets except for Video, which takes about 12× longer for DeltaTree. This is because Video has a much higher dimensionality than all other datasets, so it requires a longer time for distance calculation when encoding. In Figure 15c, we compare the whole index time of DeltaPQ with all the baselines. HDIndex, SRS, and C2LSH are orders of magnitude slower than our method while QALSH is about one order of magnitude slower. HNSW is slightly faster than DeltaTree, but its index size is at least 30× larger.

To sum up, DeltaPQ has smaller index cost than graph-based methods and LSH-based methods while providing reasonable query performance and scales linearly to the dataset.

Remark. The trade-off between search accuracy, query time, and index cost of the graph-based methods, LSH-based methods, and PQ-based methods have been well studied. For example, ann-benchmark [1] is a popular benchmark for approximate nearest neighbor search. In summary, the PQ-based methods have the smallest index size and index time among them, while the graph-based methods achieves the best accuracy-search time trade-off. PQ-based methods can be very useful when handling extremely large datasets. It can be too expensive to apply the graph-based method just building the index. In addition, the search accuracy of PQ-based method is usually good enough for many applications. For example, [65] shows that clustering based on product quantized vectors can produce competitive results compared with k-means. In addition, [9] confirms that quantized vectors on the Youtube-8M dataset do not significantly hurt the evaluation results on several video classification models.

6. RELATED WORK

Tree Indexes. Similarity search on high dimensional vectors has been extensively studied. Many tree structured indexes based on space partition have been proposed, such as the KD-tree [21], the R-tree [41], the TV-tree [57], the Quadtree [77, 32], the X-tree [22], the adaptive B^+-tree [49], A-Tree [76], M-Tree [24], Multicurves [85], DSTree [86], iSAX2+ [23], and etc.. A-Tree [76] introduces Virtual Bound Rectangles to reduce page access. M-Tree [24] is a balanced tree for generic metric space and supports I/O and computation efficient similarity search for dimensions under 50. Multicurves [85] uses space-filling curves to reduce dimensionality and then store each curve in a B-Tree. DSTree [86] uses the Extended Adaptive Piecewise Approximation summarization technique and supports upper and lower bound distances for node splitting. iSAX2+ [23] is the latest variant of the iSAX family and supports bulk loading. Its index can be used for both exact and approximate NN search [70]. Due to "the curse of dimensionality" [48], these methods only work for multi-dimensional data, and their performances degenerate dramatically when the dimensionality slightly increases [78]. HDIndex [13] is state-of-the-art tree-based method that incorporates Multicurves and stores the distances to reference sets for speeding up filtering. However, its indexing time and indexing space limit its usability on large scale datasets.

Locality Sensitive Hashing. Indyk et al. propose Locality Sensitive Hashing (LSH) to conquer the curse of dimensionality, which guarantees that close by vectors are more likely to be hashed to the same bucket than far away vectors. Datar et al. [26] design the LSH scheme for Euclidean distance and L^p-norm. Gionis et al. [38] propose LSH for hamming distance, Anshumali et al. [79] design asymmetric LSH for maximum inner product search, while Andoni et al. [11] develop an optimal LSH for Angular distance. However, the index size of LSH based methods is excessively large. To alleviate this problem, Lv et al. [62] propose to probe multiple nearby hash buckets to reduce the index size. Tao et al. [82, 83] propose to use one single index for nearest neighbor search query. Gan et al. [35] propose to count the number of hash values within the same bucket using multiple B-trees [20]. Huang et al. [47] propose to use the query as the middle point to determine buckets for better accuracy.

Proximity Graph. There are many proximity graph based methods for similarity search [45, 63, 55, 64, 33, 14, 50]. Delaunay graphs [15] and monotonic search networks [27] guarantee that for any vector/vertex, one of its neighbors is closer to the query than itself. A best-first search from any vertex can find the nearest neighbors of a query. However, their time complexity is unknown. Though randomized neighborhood graphs [14] guarantee polylogarithmic search time complexity, they are expensive to construct. Hajebi et al. [42] propose to use a knn-graph [29] to approximate these graphs. Malkov et al. [64] propose to use hierarchical navigable small-world graphs [55] for similarity search, which has been shown efficient empirically. Fu et al. [33] leverage the angular between vectors in graph construction.

Product Quantization. Jégou et al. [51] propose to use product quantization for large scale similarity search. André et al. [12] propose to use SIMD for AD computation and put the lookup table to registers. Matsui et al. [66] propose to use hash tables to index codes . Johnson et al. [53] propose to use product quantization for similarity search on GPU. Liu et al. [59] proposes to index the codes using a forest of B-trees for better disk I/Os. There are studies on optimizing codebook generation for better accuracy [36, 39, 54] and coarse quantization [17]. Variant coding strategies have been proposed [18, 88]. Vector quantization has also been used for clustering [65, 69] and deep neural network compression [44].

7. CONCLUSION

In this paper, we study lossless compression techniques for high dimensional data management. We propose two differential code compression methods to compress the quantization codes and perform queries directly on the compressed data. One of them can achieve the optimal compression ratio, while another one has slightly lower compression ratio but it only incurs tiny space overhead per query. Our experimental results show that on five real-world datasets, our approach outperforms state-of-the-art general-purpose compression methods and similarity search algorithms.

8. REFERENCES

[1] Ann-benchmark. http://ann-benchmarks.com/index.html.

[2] LZ4. https://github.com/lz4/lz4.

[3] Sald dataset. http://fcon_1000.projects.nitrc.org/indi/retro/sald.html?utm_source=newsletter&utm_medium=email&utm_content=See%20Data&utm_campaign=indi-1.

[4] Sift1b dataset. http://corpus-texmex.irisa.fr/.

[5] Tinyeye. https://tineye.com/faq#count.

[6] TurboRLE. https://github.com/powturbo/Turbo-Run-Length-Encoding.

[7] Youtube-8m dataset. https://research.google.com/youtube8m/.

[8] Youtube video statistics. https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/.

[9] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *CoRR*, abs/1609.08675, 2016.

[10] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[11] A. Andoni, P. Indyk, T. Laarhoven, I. P. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. In *NeurIPS*, pages 1225–1233, 2015.

[12] 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.

[13] A. Arora, S. Sinha, P. Kumar, and A. Bhattacharya. Hd-index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. *PVLDB*, 11(8):906–919, 2018.

[14] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *ACM/SIGACT-SIAM*, pages 271–280, 1993.

[15] F. Aurenhammer. Voronoi diagrams - A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[16] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, pages 2055–2063, 2016.

[17] A. Babenko and V. S. Lempitsky. The inverted multi-index. In *CVPR*, pages 3069–3076, 2012.

[18] A. Babenko and V. S. Lempitsky. Additive quantization for extreme vector compression. In *CVPR*, pages 931–938, 2014.

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

[20] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, 1970.

[21] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[22] S. Berchtold, D. A. Keim, and H. Kriegel. The x-tree : An index structure for high-dimensional data. In *PVLDB*, pages 28–39, 1996.

[23] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *Knowledge and information systems*, 39(1):123–151, 2014.

[24] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *PVLDB*, pages 426–435. Citeseer, 1997.

[25] Y. Collet et al. Lz4: Extremely fast compression algorithm. *code. google. com*, 2013.

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

[27] D. Dearholt, N. Gonzales, and G. Kurup. Monotonic search networks for computer vision databases. In *ACSSC*, volume 2, pages 548–553, 1988.

[28] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186, 2019.

[29] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, pages 577–586, 2011.

[30] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2):112–127, 2018.

[31] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the lernaean hydra: experimental evaluation of data series approximate similarity search. *PVLDB*, 13(3):403–420, 2019.

[32] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[33] 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.

[34] B. A. Galler and M. J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[35] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[36] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, pages 2946–2953, 2013.

[37] A. Gersho and R. M. Gray. *Vector quantization and signal compression*, volume 159 of *The Kluwer international series in engineering and computer science*. Kluwer, 1991.

[38] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *PVLDB*, pages 518–529, 1999.

[39] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, pages 817–824, 2011.

[40] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *SIGKDD*, KDD '16, pages 855–864, 2016.

[41] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[42] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI*, pages 1312–1317, 2011.

[43] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[44] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[45] B. Harwood and T. Drummond. FANNG: fast approximate nearest neighbour graphs. In *CVPR*, pages 5713–5722, 2016.

[46] E. L. Hauck. Data compression using run length encoding and statistical encoding, Dec. 2 1986. US Patent 4,626,829.

[47] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *PVLDB*, 9(1):1–12, 2015.

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

[49] H. V. Jagadish, B. C. Ooi, K. Tan, C. Yu, and R. Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[50] J. Jaromczyk and G. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80:1502 – 1517, 10 1992.

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

[52] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, pages 861–864, 2011.

[53] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[54] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *CVPR*, pages 2329–2336, 2014.

[55] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798):845–845, 2000.

[56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[57] K. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.

[58] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu. Learning entity and relation embeddings for knowledge graph completion. In *AAAI*, pages 2181–2187, 2015.

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

[60] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Information Theory*, 28(2):129–136, 1982.

[61] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

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

[63] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, 2014.

[64] 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.

[65] Y. Matsui, K. Ogaki, T. Yamasaki, and K. Aizawa. Pqk-means: Billion-scale clustering for product-quantized codes. In *ACM-MM*, pages 1725–1733, 2017.

[66] Y. Matsui, T. Yamasaki, and K. Aizawa. Pqtable: Fast exact asymmetric distance neighbor search for product quantization using hash tables. In *ICCV*, pages 1940–1948, 2015.

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

[68] J. Y. Ng, F. Yang, and L. S. Davis. Exploiting local features from deep networks for image retrieval. In *CVPR*, pages 53–61, 2015.

[69] M. Norouzi and D. J. Fleet. Cartesian k-means. In *CVPR*, pages 3017–3024, 2013.

[70] T. Palpanas. Evolution of a data series index. In *ISIP*, pages 68–83. Springer, 2019.

[71] I. Pavlov. LZMA SDK. `https://www.7-zip.org/sdk.html`, 2007.

[72] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.

[73] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34.

[74] R. C. Prim. Shortest connection networks and some generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[75] S. J. Russell and P. Norvig. *Artificial intelligence - a modern approach, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, 2003.

[76] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, et al. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *PVLDB*, volume 2000, pages 5–16. Citeseer, 2000.

[77] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

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

[79] A. Shrivastava and P. Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NeurIPS*, pages 2321–2329, 2014.

[80] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB*, 8(1):1–12, 2014.

[81] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, pages 2818–2826, 2016.

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

[83] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3):20:1–20:46, 2010.

[84] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[85] E. Valle, M. Cord, and S. Philipp-Foliguet. High-dimensional descriptor indexing for large multimedia databases. In *CIKM*, pages 739–748, 2008.

[86] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10):793–804, 2013.

[87] Z. Xu, Y. Yang, and A. G. Hauptmann. A discriminative CNN video representation for event detection. In *CVPR*, pages 1798–1807, 2015.

[88] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *ICML*, pages 838–846, 2014.