ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph

Chaoji Zuo
Rutgers University
chaoji.zuo@rutgers.edu

Dong Deng
Rutgers University
dong.deng@rutgers.edu

ABSTRACT

Given a collection of vectors, the approximate K-nearest-neighbor graph (KGraph for short) connects every vector to its approximate K-nearest-neighbors (KNN for short). KGraph plays an important role in high dimensional data visualization, semantic search, manifold learning, and machine learning. The vectors are typically vector representations of real-world objects (e.g., images and documents), which often come with a few structured attributes, such as timestamps and locations. In this paper, we study the all-range approximate K-nearest-neighbor graph (ARKGraph) problem. Specifically, given a collection of vectors, each associated with a numerical search key (e.g., a timestamp), we aim to build an index that takes a search key range as the query and returns the KGraph of vectors whose search keys are within the query range. ARKGraph can facilitate interactive high dimensional data visualization, data mining, etc. A key challenge of this problem is the huge index size. This is because, given n vectors, a brute-force index stores a KGraph for every search key range, which results in $O(\mathrm{K}n^3)$ index size as there are $O(n^2)$ search key ranges and each KGraph takes $O(\mathrm{K}n)$ space. We observe that the KNN of a vector in nearby ranges are often the same, which can be grouped together to save space. Based on this observation, we propose a series of novel techniques that reduce the index size significantly to just $O(\mathrm{K}n \log n)$ in the average case. Furthermore, we develop an efficient indexing algorithm that constructs the optimized ARKGraph index directly without exhaustively calculating the distance between every pair of vectors. To process a query, for each vector in the query range, we only need $O(\log\log n + \mathrm{K}\log\mathrm{K})$ to restore its KNN in the query range from the optimized ARKGraph index. We conducted extensive experiments on real-world datasets. Experimental results show that our optimized ARKGraph index achieved a small index size, low query latency, and good scalability. Specifically, our approach was 1000x faster than the baseline method that builds a KGraph for all the vectors in the query range on-the-fly.

PVLDB Reference Format:
Chaoji Zuo and Dong Deng. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. PVLDB, 16(10): 2645 - 2658, 2023.
doi:10.14778/3603581.3603601

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rutgers-db/range-knn-code.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603601

1 INTRODUCTION

Large-scale high dimensional dense vectors are ubiquitous nowadays due to the rapid development of deep learning and representation learning. For example, many machine learning models, such as word2vec [43, 46], doc2vec [34], node2vec [24], and graph2vec [26, 45] are developed to effectively represent real-world objects (e.g., images, documents, and graphs) as high dimensional dense vectors. A common operation over the vectors is approximate K-nearest-neighbor graph (KGraph for short) construction [16], which, given a set of vectors, connects each vector with its approximate K-nearest neighbors (KNN for short). It finds applications in high dimensional data visualization [51], semantic search (a.k.a., neural search and approximate nearest neighbor search) [19], data mining [13], machine learning [12], and manifold learning [58].

We observe that real-world objects are often associated with structured attributes, such as prices, timestamps, and locations. Thus, in this paper, we propose to study the all-range approximate K-nearest-neighbor graph (ARKGraph for short) problem. Specifically, given a set of vectors, each associated with a search key value (e.g., a timestamp), we aim to build an index that takes a search key range as the query and produces the KGraph over and only over those vectors whose search keys are within the query range. It can facilitate data mining and data visualization, as illustrated below.

Motivation Example 1. Consider the surveillance cameras deployed on the roads. Each camera continuously detects vehicles passing by it and assigns a timestamp to the vehicle. At the same time, a feature vector is extracted from each detected vehicle. Figure 1 shows all the vehicles captured by the cameras, along with the detection time. Then, we can build an ARKGraph over the vehicle feature vectors to infer the trajectories of all vehicles during any specific time period. For example, we can query the ARKGraph using the query range 12:00:00-18:00:00, which generates a KGraph as shown in the figure. Vehicles with similar feature vectors are connected to each other and they may refer to the same vehicle. Thus each clique in the KGraph (in red color) probably corresponds to the trajectory of a vehicle between 12:00:00 and 18:00:00, since the locations of cameras and the detection time are both available.

Motivation Example 2. t-SNE [55] and its variants are the *de facto* high dimensional data visualization methods. The first step of t-SNE is constructing a KGraph over the vectors to be visualized [51, 54]. Thus we can use ARKGraph to help explore the visualizations of feature vectors in user-specific ranges (e.g., visualizing the feature vectors of news articles published during the outrage of pandemic).

A key challenge here is the huge size of the ARKGraph. To see this, consider a set of n feature vectors and search keys with a total order. A raw ARKGraph contains a KGraph for each of the $O(n^2)$ search key ranges. In addition, each KGraph contains $O(n)$ adjacent lists of fixed-length K (each adjacent list is a KNN). Thus the total size of the raw ARKGraph is $O(\mathrm{K}n^3)$, which is prohibitively large. To address this issue, we observe that the KNN of a vector in nearby

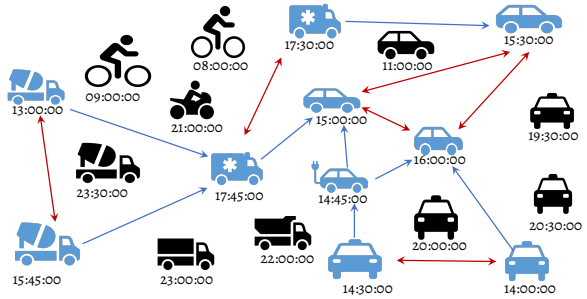


Figure 1: A motivation example.

search key ranges (e.g., $[i, j]$ and $[i, j+1]$) are likely to be the same. Thus, for each vector, we propose to group the search key ranges in which its KNN remain the same and store the KNN (i.e., adjacent list) only once. It can reduce the index size to $O(\mathrm{K}^2 n^2)$.

The index size can be further reduced. Specifically, consider a vector v with search key value c and a search key range $[i, j]$, where $i \le c \le j$. We observe that the KNN of v in the range $[i, j]$ can be derived from the KNN of v in $[i, c-1]$ and the KNN of v in $[c+1, j]$. This is because the K vectors nearest to v in the two KNN lists must be the KNN of v in $[i, j]$ (note the KNN of v does not include itself). Moreover, there are $(c-1)(n-c)$ possible ranges $[i, j]$ and KNN lists for v in these ranges. In comparison, there are only $n-1$ possible "partial ranges" $[i, c-1]$ and $[c+1, j]$ and KNN lists for v in these partial ranges. Based on these observations, we propose to replace the KNN of the vector v in the range $[i, j]$ with the two KNN of v in the two partial ranges $[i, c-1]$ and $[c+1, j]$. After applying the grouping technique discussed earlier, the index size can be significantly reduced. Formally, we prove that storing the (grouped) KNN in partial ranges can reduce the index size to $O(\mathrm{K}n^2)$ in the worst case and $O(\mathrm{K}^2 n \log n)$ in the average case.

Furthermore, we find that, even if the KNN lists of a vector are not entirely the same in two consecutive partial ranges, their differences are small (differ by one neighbor at most). Thus we propose to store the delta of KNN in consecutive partial ranges (a.k.a., delta compression [44]). We formally prove that after applying delta compression our optimized ARKGraph index takes only $O(\mathrm{K}n \log n)$ space on average. In comparison, a single KGraph over all the vectors takes $O(\mathrm{K}n)$ space, in both the average and the worst case.

Another key challenge is how to construct the above optimized ARKGraph index efficiently without exhaustively calculating the distance between every pair of vectors. We find that certain distance calculations can be avoided. For example, consider a vector v with search key c. Suppose the KNN of v in $[c+1, n]$ is obtained. Let u be the K-th nearest neighbor of v in $[c+1, n]$ and j be its search key. Then, the KNN of v in $[c+1, j]$, $[c+1, j+1]$, $\cdots$, $[c+1, n-1]$ must be the same as that in $[c+1, n]$. Thus it is unnecessary to calculate the distance between v and any vector with search key in $[j+1, n]$. Based on this observation, we propose to visit the vectors in the ascending order of their distances to v such that many vectors can be skipped. Finally, we design an efficient query processing method. It takes a query range as input and restores the KNN of every vector in the query range using merely $O(\log\log n + \mathrm{K}\log \mathrm{K})$ time.

In summary, we make the following contributions in this paper.

(1) We formalize the all-range approximate K-nearest-neighbor graph ARKGraph problem.

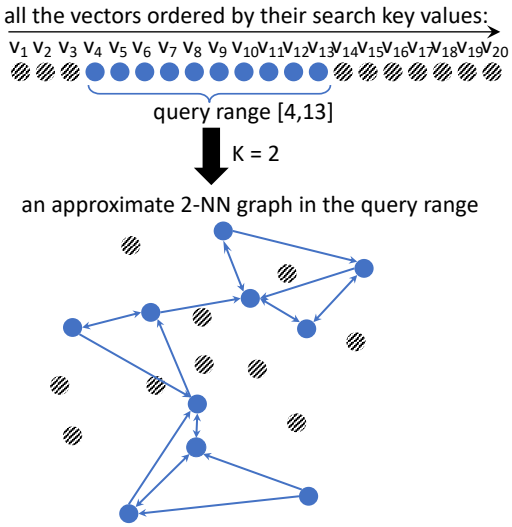


Figure 2: An example of the range KGraph query $[4, 13]$.

(2) We propose a series of novel techniques that reduce the index size from $O(\mathrm{K}n^3)$ (the raw ARKGraph) to $O(\mathrm{K}n \log n)$ (the optimized ARKGraph) in the average case.
(3) We develop an efficient indexing algorithm that construct the optimized ARKGraph index directly, as well as query processing algorithms that restore the KGraph in any query range instantly.
(4) We conduct experiments on real-world datasets. Experimental results show our optimized ARKGraph index achieved small index size, low query latency, and high scalability.

The rest of the paper is organized in the following way. Section 2 defines the problem and introduces the brute-force index. Section 3 and Section 4 present our grouping techniques. We discuss query processing in Section 5. Section 6 shows the experimental results, Section 7 surveys related works, and Section 8 concludes the paper.

2 PRELIMINARIES

2.1 Problem Definition

We first formally define the approximate K-nearest-neighbor graph.

Definition 1. *An approximate* K*-nearest-neighbor graph (*KGraph*) of a set $\mathcal{D}$ of vectors consists of an adjacent list $L(v)$ for each vector $v \in \mathcal{D}$. The adjacent list $L(v)$ contains* K *vectors in $\mathcal{D} \setminus \{v\}$. Let $N(v)$ be the* K *vectors in $\mathcal{D} \setminus \{v\}$ nearest to v. The accuracy of the graph is*

$$\frac{1}{|\mathcal{D}|} \sum_{v \in \mathcal{D}} \frac{|L(v) \cap N(v)|}{\mathrm{K}}.$$

The distance of two vectors are measured by the Euclidean distance. Next, we formally define the range approximate K-nearest-neighbor graph query (range KGraph query) as below.

Definition 2 (Range K-Nearest-Neighbor Graph Query). *Given a set $\mathcal{D}$ of vectors, where each vector $v_i \in \mathcal{D}$ is associated with a search key value i, a range KGraph query is a range $[x, y]$. It returns a KGraph over the subset $\{v_i \in \mathcal{D} \mid x \le i \le y\}$ of vectors of $\mathcal{D}$.*

Example 1. Consider the set of vectors $\mathcal{D} = \{v_1, v_2, \cdots, v_{20}\}$ as shown in Figure 2 and let K = 2. The range approximate 2-nearest-neighbor graph query $[4, 13]$ returns the (highlighted) approximate 2-nearest-neighbor graph with 10 nodes and 20 directed edges.

Algorithm 1: BruteForceIndex($\mathcal{D}$, K)

Input: $\mathcal{D} = \{v_1, \cdots, v_n\}$: a set of data vectors; K: an integer.
Output: G: a raw ARKGraph index that takes $O(\mathrm{K}n^3)$ space.

```
1 begin
2     foreach data vector v_i ∈ D do
3         foreach 1 ≤ x < i and i < y ≤ n do
4             Calculate the KNN of v_i in the range [x, y] \ {i}
              and store them in the adjacent list G_[x,y][v_i];
5     return G;
6 end
```

2.2 Raw ARKGraph Index

In this section, we present a brute-force method that constructs a raw ARKGraph index. It enumerates every search key range and builds a KGraph over the vectors in the enumerated range. Thus the raw ARKGraph index consists of a KGraph for each search key range. Note, for simplicity, we refer to the vectors whose search keys are in a range as vectors in that range.

Algorithm 1 shows the pseudo-code of the brute-force method. It takes a set of vectors $\mathcal{D} = \{v_1, v_2, \cdots, v_n\}$ (each v_i is associated with a search key value i) and an integer K as input. It first enumerates every vector v_i in $\mathcal{D}$ (Line 2). Then it enumerates every search key range $[x, y]$ containing v_i where $1 \leq x < i$ and $i < y \leq n$ (Line 3). Next, it calculates the KNN of v_i in $[x, y] \setminus \{i\}$ and stores them in the adjacent list $\mathbf{G}_{[x,y]}[v_i]$ (Line 4). Note that in the corner case where there are less than K vectors in the range $[x, y] \setminus \{i\}$, we simply include all the vectors in it in the adjacent list. This corner case is handled in the same way hereinafter. Furthermore, we assume the distance to v_i from every other vector is distinct (order by their search keys if two vectors have the same distance to v_i). In addition, when the context is clear, we refer to the range $[x, y] \setminus \{i\}$ simply as $[x, y]$. Finally, the raw ARKGraph index $\mathbf{G}$ is returned (Line 5).

Example 2. Consider the set of 20 vectors at the top of Figure 3. Let the integers under the vectors be their distance to the vector v_9. As shown in Figure 3 in the middle left, for the vector v_9 and K = 2, the brute-force method would enumerate every range $[x, y]$ where $1 \leq x < 9$ and $9 < y \leq 20$. In total, 88 ranges and the corresponding 2NN of v_9 in them are indexed as adjacent lists. This process is repeated for each of the rest of 19 vectors. During the online query phase, we can derive the 2NN graph over any query range s on demand because for every vector v in s, we have its 2NN in s in the raw ARKGraph index represented by the adjacent list $\mathbf{G}_s[v]$.

Complexity Analysis. For each vector, there are $O(n^2)$ ranges containing it where n is the total number of vectors. In total, $O(n^3)$ adjacent lists are generated, each contains $O(\mathrm{K})$ neighbors. Thus the size of the raw ARKGraph index is $O(\mathrm{K}n^3)$. The time complexity of the brute-force method is $O(n^2 d + n^4 \log \mathrm{K} + n^3 \mathrm{K})$, where d is the dimensionality of the vectors. This is because it takes $O(n^2 d)$ to calculate the distance of all pairs of vectors beforehand and for each vector v_i, it enumerates $O(n^2)$ ranges. For each range, it takes $O(n \log \mathrm{K})$ to get the KNN of v_i and $O(\mathrm{K})$ to produce the adjacent list. Clearly, *two key challenges in our framework are the huge index size and excessively long indexing time.* Next, we discuss how to significantly reduce both of them.

3 COMPACT GRAPH INDEX

3.1 Compact Adjacent List

Our key observation is that, for each vector v_i, in many nearby ranges (i.e., those with similar starting and ending positions), the KNN of v_i are exactly the same. For example, consider the vector v_9 in Figure 3. The 2NN of v_9 in the four ranges $[1, 11]$, $[2, 11]$, $[2, 12]$, and $[3, 12]$ are exactly the same, which are $\{v_{11}, v_6\}$. Actually, as one can verify, in every range $[x, y]$, where $1 \leq x \leq 6$ and $11 \leq y \leq 12$, the 2NN of v_9 are the same, i.e., $\mathbf{G}_{[x,y]}[v_9] = \{v_{11}, v_6\}$. Based on this observation, for each vector v_i, we propose to group all the search key ranges across i by the KNN of v_i in them. That is, we propose to aggregate all the ranges $[x, y]$ where $1 \leq x < i < y \leq n$ with the same adjacent list $\mathbf{G}_{[x,y]}[v_i]$. For this purpose, we formally define the *compact range* and *compact adjacent list* of a vector.

Definition 3 (Compact Range). *The compact range $\langle b, e \rangle$ of a vector v_i consists of two intervals $b = [b_l, b_r]$ and $e = [e_l, e_r]$ where $b_r < i < e_l$. It represents all the ranges $[x, y]$ where $b_l \leq x \leq b_r$ and $e_l \leq y \leq e_r$, i.e., starting from b and ending within e.*

Definition 4 (Compact Adjacent List). *The compact adjacent list $\mathbf{G}_{\langle b,e \rangle}[v_i] = C$ of a vector v_i in its compact range $\langle b, e \rangle$, if exists, is the list C of* K *search keys in $[b_r, e_l] \setminus \{i\}$ such that*

(1) C is the KNN of v_i in $[b_l, e_r] \setminus \{i\}$.
(2) $b_r = C_{min}$: the smallest search key value in $C \cup \{i - 1\}$;
(3) $e_l = C_{max}$: the largest search key value in $C \cup \{i + 1\}$;
(4) $\exists\, p \in C$ s.t. $d(v_p, v_i) > d(v_{b_l - 1}, v_i)$, if $b_l \neq 1$;
(5) $\exists\, p \in C$ s.t. $d(v_p, v_i) > d(v_{e_r + 1}, v_i)$, if $e_r \neq n$.

For ease of presentation, we refer to a vector v_i and its search key value i interchangeably when the context is clear. Thus C denotes both a list of K search key values and the corresponding list of K vectors. The first condition in Definition 4 implies that the KNN of v_i in every range $[x, y]$ in the compact range $\langle b, e \rangle$ where $x \in b$ and $y \in e$ is C. This is because C consists of the K vectors nearest to v_i among all the vectors in $[b_l, e_r]$, while $C \subseteq [b_r, e_l] \subseteq [x, y] \subseteq [b_l, e_r]$. The last four conditions ensure the compact range is "maximal", i.e., expanding either interval b or e makes the first condition no longer hold.

Example 3. Consider the set of vectors $\{v_1, v_2, \cdots, v_{20}\}$ in Figure 3. The two intervals $b = [b_l = 1, b_r = 6]$ and $e = [e_l = 11, e_r = 12]$ compose a compact range $\langle b, e \rangle$ of v_9 as $b_r < i = 9 < e_l$. Let K = 2. There is a compact adjacent list of v_9 in the compact range $\langle b, e \rangle$, which is $\mathbf{G}_{\langle b,e \rangle}[v_9] = C = \{v_{11}, v_6\}$. This is because (1) the 2NN of v_9 in the range $[b_l, e_r] = [1, 12]$ is $C = \{v_{11}, v_6\}$; (2) the smallest search key value C_{min} in $C \cup \{i - 1\}$ is 6 and $b_r = 6$; (3) the largest search key value C_{max} in $C \cup \{i + 1\}$ is 11 and $e_l = 11$; (4) $b_l = 1$; and (5) v_{13} is closer to v_9 than $v_p \in C$ for $p = 6$ (and $p = 11$).

Note that not every compact range of a vector has a compact adjacent list. The total number of compact adjacent lists in a vector is much less than the total number of its (ordinary) adjacent lists. Actually, the compact adjacent lists can be seen as a *lossless compression* of the ordinary adjacent lists generated by the brute-force index method. This is because every ordinary adjacent list is in one and only one compact adjacent list as formalized below.

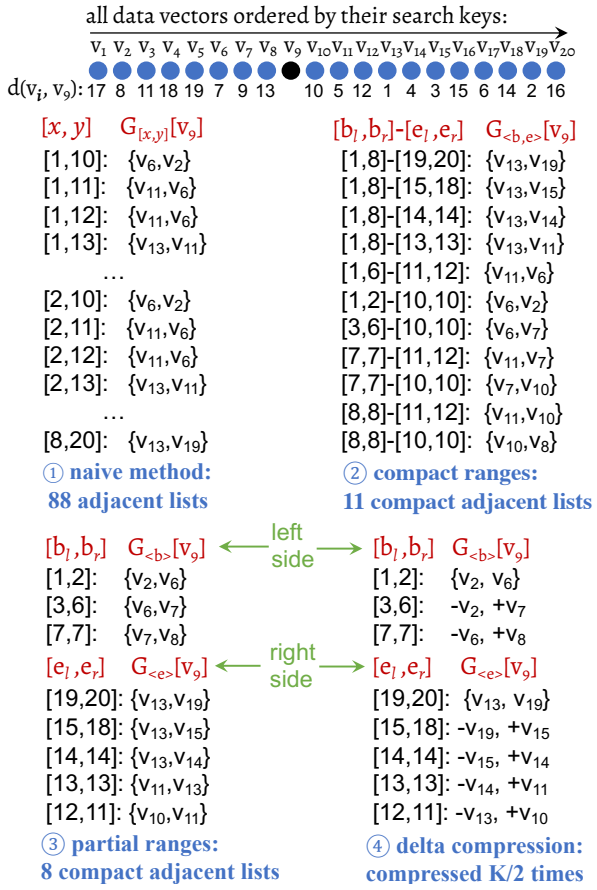


Figure 3: An example of four index methods.

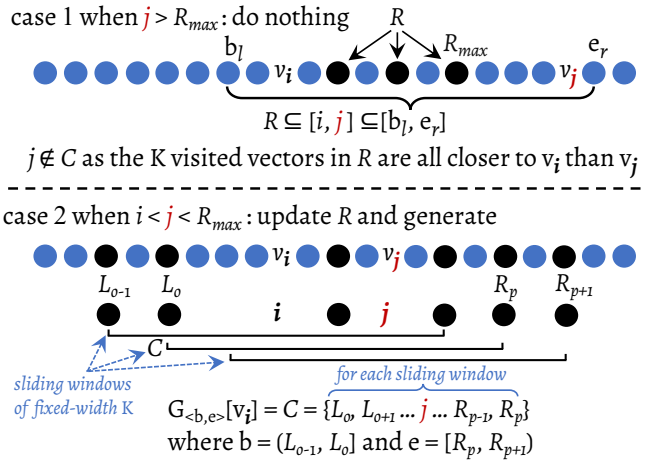


Figure 4: Two cases when visiting next vector v_j in sequence.

Lemma 1. *For every adjacent list* $\mathbf{G}_{[x,y]}[v_i]$ *of a vector* v_i*, where* $x < i < y$*, there is one and only one compact range* $\langle b, e\rangle$ *such that* $\mathbf{G}_{\langle b,e\rangle}[v_i]$ *is a compact adjacent list and* $\mathbf{G}_{\langle b,e\rangle}[v_i] = \mathbf{G}_{[x,y]}[v_i]$.

3.2 Compact Adjacent List Generation

In this section, we discuss how to generate all the compact adjacent lists of a data vector. Given a data vector v_i, to generate all its compact adjacent lists, we sort all the data vectors in $\mathcal{D} \setminus \{v_i\}$ by their distance to v_i in ascending order and visit them in sequence. *Each time we visit a data vector* v_j*, we aim to generate all the compact adjacent lists* C *(if there is any) where* $v_j \in C$ *is the farthest to* v_i *among the* K *data vectors in* C. We observe that all the visited vectors are closer to v_i than all the unvisited vectors. Thus the target C must consist of K visited vectors including v_j. Based on this observation, we propose to maintain two lists $\mathcal{L}$ and $\mathcal{R}$ of visited vectors. Specifically, among the visited vectors whose search keys are smaller than i (i.e., on the left side of v_i), we keep the K largest ones in $\mathcal{L}$. Similarly, among the visited vectors whose search keys are larger than i (i.e., on the right side of v_i), we keep the K smallest ones in $\mathcal{R}$. Moreover, the search keys in $\mathcal{L}$ and $\mathcal{R}$ are sorted in ascending order. For ease of presentation, we denote the p-th search key in $\mathcal{L}$ and $\mathcal{R}$ as $\mathcal{L}_p$ and $\mathcal{R}_p$. In addition, we denote the minimum search key in $\mathcal{L}$ as $\mathcal{L}_{min}$ and the maximum search key in $\mathcal{R}$ as $\mathcal{R}_{max}$. Let v_j be the next vector to visit in sequence (note $j \neq i$ and $j \notin \mathcal{L} \cup \mathcal{R}$). We consider the two cases when $j > \mathcal{R}_{max}$ and when $i < j < \mathcal{R}_{max}$ as shown in Figure 4 (note at the beginning, when $\mathcal{L}$ and $\mathcal{R}$ are empty, we define $\mathcal{L}_{min} = 0$ and $\mathcal{R}_{max} = n + 1$).

Case 1: $j > \mathcal{R}_{max}$**.** Our goal is to generate all the compact adjacent lists $\mathbf{G}_{\langle b,e\rangle}[v_i] = C$ (if there is any) where the current visiting vector $v_j \in C$ is farthest to v_i among the K data vectors in C. We prove no such compact adjacent list exists when $j > \mathcal{R}_{max}$ by contradiction. Suppose there exists one such compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v_i] = C$. Based on Definition 4, the search keys in C, including j, are all in $[b_r, e_l]$. Thus $j \leq e_l$. Based on the definitions of $\mathcal{R}$ and compact range, we have $b_l \leq b_r < i < \mathcal{R}_{max} < j \leq e_l \leq e_r$. Thus the range $[b_l, e_r]$ must contain all the K visited vectors in $\mathcal{R}$, which are all closer to v_i than v_j, as shown in Figure 4 at the top. Thus C cannot be the KNN of v_i in $[b_l, e_r]$ as $v_j \in C$, which contradicts with the first condition in Definition 4. Thus no target compact adjacent list exists and we do nothing when $j > \mathcal{R}_{max}$.

Case 2: $i < j < \mathcal{R}_{max}$**.** In this case, we first update $\mathcal{R}$, which consists of the K smallest search keys of visited vectors that are greater than i. Since the search key $j > i$ of the current visiting vector v_j is smaller than $\mathcal{R}_{max}$, we insert j into $\mathcal{R}$ and remove $\mathcal{R}_{max}$ from $\mathcal{R}$ if there are more than K search keys in $\mathcal{R}$.

Next, we generate all the compact adjacent lists $\mathbf{G}_{\langle b,e\rangle}[v_i] = C$ where the current visiting vector $v_j \in C$ is furthest to v_i among the K data vectors in C. As all the visited vectors are closer to v_i than v_j, while all the unvisited vectors are farther to v_i than v_j, the K data vectors in C must be K visited vectors including v_j. However, we can prove C cannot contain any visited vector other than those in $\mathcal{L}$ or $\mathcal{R}$ by contradiction. Without loss of generality, suppose C contains a visited vector v_p where $p > i$ and $v_p \notin \mathcal{L} \cup \mathcal{R}$. As $\mathcal{R}$ are the K smallest search keys of visited vectors greater than i, we have $p > \mathcal{R}_{max}$. Then there are at least K vectors in $\mathcal{R} \cup \{v_p\}$ visited before v_j and thus closer to v_i than v_j. Thus C cannot be the KNN of $[b_l, e_r]$ as $v_j \in C$ and $\mathcal{R} \cup \{v_p\} \subseteq [b_r, e_l] \subseteq [b_l, e_r]$ (this is because $e_l = C_{max} \geq p \in C$ and $b_r < i < \mathcal{R}_{min}$ based on Definition 4), which contradicts with the first condition in Definition 4. Thus the K data vectors in C must be from $\mathcal{L} \cup \mathcal{R}$ and must include v_j. Similarly,

Algorithm 2: CompactGraphIndex($\mathcal{D}$, K)

Input: $\mathcal{D} = \{v_1, \cdots, v_n\}$: a set of data vectors; K: the degree.
Output: G: the compact graph index of $\mathcal{D}$.

```
1  foreach data vector v_i ∈ D do
2      S = SortByDistance(D, v_i);
3      foreach v_j ∈ S in sequence do
4          if j < i then
5              if j > L_min then
6                  Insert j into L;
7                  if |L| == K + 1 then
8                      Remove L_min from L;
9                  Slide a window of fixed-width K containing j
                   over L ∪ R;
10                 For each sliding window C = {L_o, ..., R_p},
                   build a compact adjacent list G_⟨b,e⟩[v_i] = C
                   where b = (L_{o-1}, L_o] and e = [R_p, R_{p+1});
11                 if |L| == K then
12                     Build compact adjacent list G_⟨b,e⟩[v_i] = L,
                       where b=(L_{min-1}, L_min], e=[i-1, R_min]
                       and L_{min-1} is the previous L_min in L.
13         else if j > i then
14             if j < R_max then
                   // symmetric process to Lines 6-12
15 return G;
```

we can prove the K data vectors in C must be consecutive. Thus, as shown in Figure 4 in the bottom, we propose to slide a window of fixed-length K over the list $\mathcal{L} \cup \mathcal{R}$. For each sliding window $C = \{\mathcal{L}_o, \cdots, j, \cdots, \mathcal{R}_p\}$ containing v_j, we generate a compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v_i] = C$ where $b = (\mathcal{L}_{o-1}, \mathcal{L}_o]$ and $e = [\mathcal{R}_p, \mathcal{R}_{p+1}]$. In addition, as a corner case, if $|\mathcal{R}| = \mathrm{K}$, we generate an additional compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v_i] = \mathcal{R}$ where $b = (\mathcal{L}_{max}, i-1]$ and $e = [\mathcal{R}_{max}, \mathcal{R}_{max+1}]$, where $\mathcal{R}_{max+1}$ is the previous $\mathcal{R}_{max}$ in $\mathcal{R}$.

The other two cases symmetric to the two we discussed above can be handled similarly. Specifically, when $j < \mathcal{L}_{min}$, we do nothing. In case $\mathcal{L}_{min} < j < i$, we update $\mathcal{L}$ with j and generate a compact adjacent list for each sliding window of fixed-width K over the list $\mathcal{L} \cup \mathcal{R}$ given that the sliding window contains j.

Algorithm 2 shows the pseudo-code of our compact adjacent list generation algorithm. It takes a set $\mathcal{D}$ of data vectors and an integer K as input and outputs a compact graph $\mathbf{G}$ of $\mathcal{D}$ index for ARKGraph query processing. For each data vector $v_i \in \mathcal{D}$, it first sorts all the other data vectors in $\mathcal{D}$ by their distance to v_i (Line 2) and then visits them in sequence (Line 3). For each visit, if the visiting vector v_j is to the left of v_i, it checks if its search key j is among the K largest search keys smaller than i of visited vectors (Lines 4 to 5). If so, it first updates $\mathcal{L}$ with j (Lines 6 to 8) and then slides a window of fixed-width K containing j over $\mathcal{L} \cup \mathcal{R}$ (Line 9). Next, for each sliding window, it generates a compact adjacent list (Line 10). Finally, it generate a compact adjacent list for the corner case if there are K visited vectors in $\mathcal{L}$ (Line 12). When the visiting vector v_j is to the right of v_i and its search key j is among the K smallest ones greater than i of visited vectors, the algorithm performs symmetrically to the above process (Lines 13 to 14). Lastly, the compact graph index $\mathbf{G}$ in the format of compact adjacent lists is returned (Line 15).

Example 4. As shown in Figure 3, consider the data vector v_9 and $\mathrm{K} = 2$. We will sort and visit $v_{13}, v_{19}, v_{15}, v_{14}, v_{11}, v_{17}, v_6, v_2, v_7, v_{10}, v_3, v_{12}, v_8, v_{18}, v_{16}, v_{20}, v_1, v_4, v_5$ in sequence. In the first visit v_{13}, as $j = 13 > i = 9$ and $j < \mathcal{R}_{max} = n{+}1 = 21$, we insert j to $\mathcal{R}$ and have $|\mathcal{R}| = 1 < \mathrm{K}+1$. There is no sliding window over $\mathcal{L} \cup \mathcal{R} = \{13\}$ of width $\mathrm{K} = 2$ and we do not generate any compact adjacent list. Next, it visits v_{19}, v_{15}, and v_{14} in the next three times. $\mathcal{R}$ becomes $\{13, 19\}$, $\{13, 15\}$, and $\{13, 14\}$ after each of the three visits, while $\mathcal{L}$ remains empty. Then, it visits v_{11}. Since $j = 11 > i$ and $j < \mathcal{R}_{max} = 14$, we update $\mathcal{R}$ as $\{11, 13\}$. Next we slide a window of width 2 over $\mathcal{L} \cup \mathcal{R} = \{11, 13\}$. There is no sliding window starting from a search key in $\mathcal{L} = \emptyset$ and no compact adjacent list is generated. However, as $|\mathcal{R}| = 2 = \mathrm{K}$, for the corner case, we generate a compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v_9] = \mathcal{R} = \{11, 13\}$ where $b = (\mathcal{L}_{min} = 0, i-1 = 9]$ and $e = [\mathcal{R}_{max} = 13, \mathcal{R}_{max+1} = 14)$ (note $\mathcal{R}_{max+1}$ is the previous $\mathcal{R}_{max} = 14$ before updating). In total, it only indexes 11 compact adjacent lists instead of 88 adjacent lists in the naive method, which significantly reduces the index size.

Theorem 5. *Algorithm 2 is sound and correct, i.e., it generates and only generates all the compact adjacent lists of every data vector.*

Complexity Analysis. We first analyze the index size. For each data vector, we visit all the other $n-1$ data vectors in sequence. Each visit produces $O(\mathrm{K})$ compact adjacent lists. In total, $O(n^2\mathrm{K})$ compact adjacent lists are generated. As each compact adjacent list takes $O(\mathrm{K})$ space, the size of the compact graph index in the format of compact adjacent lists is $O(\mathrm{K}^2n^2)$. Let the time complexity of sorting data vectors be $O(\mathcal{S})$. The total index time complexity is $O(\mathcal{S} + \mathrm{K}^2n^2)$. This is because, for each data vector, it visits the rest $n-1$ data vectors. In each visit, it takes $O(\mathrm{K})$ to update the list $\mathcal{L}$ or $\mathcal{R}$ and $O(\mathrm{K}^2)$ to generate the compact adjacent lists.

The above analysis considers the worst case. However, when visiting a data vector v_j, it is skipped and results in no compact adjacent list whenever $j < \mathcal{L}_{min}$ or $j > \mathcal{R}_{max}$. A natural question to ask is how many data vectors are skipped in expectation? Without loss of generality, consider $j < i$. We observe that, when visiting v_j, if there are at least K visited vectors whose search keys are larger than j and smaller than i, $j < \mathcal{L}_{min}$ and v_j is skipped. Similarly, when $j > i$, if there are at least K visited vectors whose search keys are larger than i and smaller than j, $j > \mathcal{R}_{max}$ and v_j is skipped.

Based on the discussion above, we calculate the expected number of skipped data vectors for the data vector v_i. Let $x_1, x_2, \cdots, x_{i-1}$ be the search keys of $v_1, v_2, \cdots, v_{i-1}$ ordered by their corresponding data vectors' distance to v_i. Then in the h-th visit, the data vector v_{x_h} is visited. This data vector is skipped iff. there are at least K of $x_1, x_2, \cdots, x_{h-1}$ that are greater than x_h. Suppose the distance between two data vectors is independent to their search keys. The probability this data vector is not skipped is $\frac{\mathrm{K}}{h}$. Thus the expected number of data vectors that are not skipped is

$$\mathbb{E}[\#\ visits] = \sum_{h=\mathrm{K}}^{i-1} \frac{\mathrm{K}}{h} + \sum_{h=\mathrm{K}}^{n-i} \frac{\mathrm{K}}{h} = O(\mathrm{K}\log n). \tag{1}$$

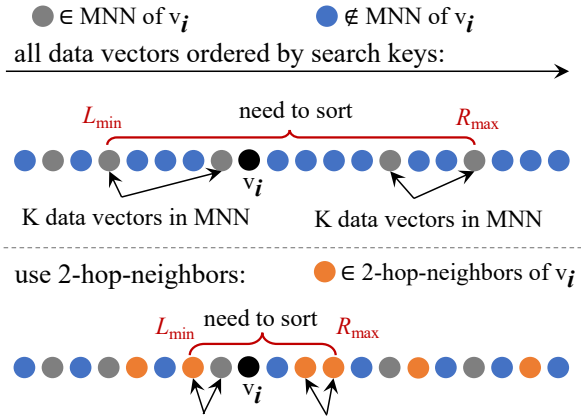


Figure 5: Skip a few data vectors using the MNN of v_i.

Based on the discussion above, we have the followings.

Lemma 2. *The KNNG-based compact graph index has an average size of $O(K^3 n \log n)$ assuming the distance of two vectors and their search keys are independent.*

Proof. In expectation, $O(K \log n)$ visits are not skipped for each data vector. Each visit not skips results in $O(K)$ compact adjacent lists. Thus the index size is $O(K^3 n \log n)$ on average. □

3.3 Efficient Vector Sorting with Cost Model

To efficiently implement the procedure SortByDistance, we propose to partially sort all the data vectors in advance using NNDescent [16]. It generates an approximate M-nearest neighbor graph (MNNG for short) over all the data vectors efficiently. That is, every data vector $v_i \in \mathcal{D}$ is connected to its approximate M-nearest neighbors in $\mathcal{D} \setminus \{i\}$.

For each data vector v_i, we visit the other vectors in the ascending order of their distance to v_i. Thus the MNN of v_i are visited first. We observe that, when visiting a data vector v_j, we do nothing if $j < \mathcal{L}_{min}$ or $j > \mathcal{R}_{max}$. Thus, after visiting the MNN of v_i, as shown in Figure 5 at the top, we can skip all the data vectors whose search keys are smaller than $\mathcal{L}_{min}$ or larger than $\mathcal{R}_{max}$ without calculating their distance to v_i as they do not result in any compact adjacent list. Moreover, based on the definitions, $\mathcal{L}_{min}$ must be the K-th largest search key smaller than i in the MNN of v_i, while $\mathcal{R}_{max}$ must be the K-th smallest search key greater than i in the MNN of v_i. Thus, for each data vector v_i, we first visit the MNN of v_i in sequence. Then we sort all the data vectors by their distance to v_i in the rage $(\mathcal{L}_{min}, \mathcal{R}_{max}) \setminus \{i\}$ and visit them in sequence.

For the partial sort to be effective, we need to set M ≥ K. Furthermore, in the MNNG produced by NNDescent, the neighbors of v_i's neighbors (i.e., 2-hop neighbors) are also likely to be close to v_i. Thus we can make an assumption that the x-hop neighbors of v_i are closer to v_i than the $(x+1)$-hop neighbors. Based on this assumption, we can sort all the neighbors of v_i within x-hop to get an approximate X-nearest-neighbors of v_i where X is the number of neighbors within x-hop. Figure 5 on the bottom shows how the 2-hop-neighbor helps to reduce the number of distances calculations.

Clearly, the more hops we go, the more neighbors we need to visit, while the smaller the range $(\mathcal{L}_{min}, \mathcal{R}_{max}) \setminus \{i\}$ we need to scan. To strike a good balance, we propose a simple cost model. Specifically, we increase the hop x one by one and stop whenever $\mathcal{R}_{max} - \mathcal{L}_{min}$ (the number of data vectors need to be scanned) is smaller than the total number of $(x+1)$-hop neighbors.

4 ARKGRAPH INDEX

Though the index size in the format of compact adjacent lists is reduced in compare with the raw format, it is still prohibitively large. Next, we discuss how to further reduce the index size significantly.

4.1 Compact Partial Ranges

Given a data vector v_i, we observe that the KNN of v_i in any range $[x, y] \setminus \{i\}$ can be derived from the KNN of v_i in the "partial range" $[x, i)$ and the KNN of v_i in the "partial range" $(i, y]$ as $[x, y] \setminus \{i\} = [x, i) \cup (i, y]$. Specifically, after merging the two KNNs, the first K data vectors ranked by their distance to v_i are obviously the KNN of v_i in the range $[x, y] \setminus \{i\}$. Based on this observation, we propose to index the KNN of v_i in every range $[x, i)$ and $(i, y]$ where $1 \le x < i$ and $i < y \le n$ in the adjacent list $\mathbf{G}_x[v_i]$ and $\mathbf{G}_y[v_i]$. During the online querying phase, given a query range $s = [x, y]$, to find the KNN of a data vector v_i in the query range, we merge the two adjacent lists $\mathbf{G}_x[v_i]$ and $\mathbf{G}_y[v_i]$ and retrieve the top-K of them.

For this purpose, given a data vector v_i, we define the range ending with $i-1$ or starting from $i+1$ as a *partial range* of v_i. Clearly, in total there are only $O(n)$ partial ranges for every data vector v_i. In comparison, the naive index method generates $O(n^2)$ adjacent lists for v_i, one for each range $[x, y]$ (where $x < i < y$). By indexing the adjacent lists of the partial ranges instead, the KNNG-based compact graph index size is reduced to $O(Kn^2)$ as summarized in Table 1. Moreover, we can also aggregate the partial ranges with common adjacent lists into "*compact partial ranges*" to further reduce the index size and time. Next, we discuss the details.

Aggregating partial ranges with the same adjacent lists. Given a data vector v_i, without loss of generality, we consider all the search keys smaller than i. All of our discussions and conclusions naturally apply to the search keys larger than i in a symmetric way. To aggregate partial ranges with the same adjacent lists, we formally define the compact partial range as below.

Definition 5 (Compact Partial Range). *Given a data vector v_i, its compact partial range is an interval $b = [b_l, b_r]$ where $1 \le b_l \le b_r < i$. It represents all the partial ranges $[x, i)$ where $b_l \le x \le b_r$. Its compact adjacent list $\mathbf{G}_{\langle b \rangle}[v_i] = C$, if exists, is the list C of K search keys in $[b_r, i)$ such that*

(1) C is the KNN of v_i in $[b_l, i)$;
(2) $b_r = C_{min}$: the smallest search key in C;
(3) $\exists\, p \in C$ s.t. $d(v_p, v_i) > d(v_{b_l - 1}, v_i)$, if $b_l \neq 1$.

Example 6. Consider the data vectors in Figure 3 and K = 2. The interval $b = [b_l = 3, b_r = 6]$ is a compact partial range of v_9 as $b_r < i = 9$. It has a compact adjacent list $\mathbf{G}_{\langle b \rangle}[v_9] = C = \{v_6, v_7\}$. This is because (1) the 2NN of v_9 in the range $[b_l, i) = [3, 8]$ is $C = \{v_6, v_7\}$; (2) the smallest search key C_{min} in C is 6 and $b_r = 6$; and (3) v_2 is closer to v_9 than $v_p \in C$ for $p = 7$.

Algorithm 3: ARKGraphIndex($\mathcal{D}$, K)

```
// Replace Lines 8-12 by the below in Algorithm 2.
1 if |L| == K + 1 then
2     L_{min-1} = L_min;
3     Remove L_min from L; // note L_min is updated;
4     Build a compact adjacent list G_{⟨b⟩}[v_i] = L where
      b = (L_{min-1}, L_min];
// Symmetric process after Line 14 of Algorithm 2.
```

Similarly, we can prove that all the compact adjacent lists in a data vector's compact partial ranges are a lossless compression of all the adjacent lists in its partial ranges as formalized below.

Lemma 3. *For every adjacent list $\mathbf{G}_x[v_i]$ of a data vector v_i, there is one and only one compact partial range b such that it has a compact adjacent list $\mathbf{G}_{\langle b\rangle}[v_i]$ and $\mathbf{G}_{\langle b\rangle}[v_i] = \mathbf{G}_x[v_i]$.*

The proof is similar to the proof of Lemma 1. Due to space limit, we omit it here. Next we discuss how to generate all the compact adjacent lists of the compact partial ranges in a given data vector.

4.2 ARKGraph Index Construction

Given a data vector v_i, we visit all the data vectors whose search keys are smaller than i (i.e., $v_1, v_2, \cdots, v_{i-1}$) in the ascending order of their distance to v_i. Sam as before, *when visiting v_j, we aim to generate all the compact adjacent lists C where $v_j \in C$ is farthest to v_i among the* K *data vectors in C*. For this purpose, we keep the K largest search keys of visited vectors smaller than i in $\mathcal{L}$ and denote $\mathcal{L}_{min}$ as the smallest search key in $\mathcal{L}$. At the beginning, $\mathcal{L}_{min} = 0$. We consider the following two cases.

Case 1: $j < \mathcal{L}_{min}$**.** In this case, the K data vectors in $\mathcal{L}$ are all closer to v_i than v_j and their search keys are all larger than j. Thus for any partial range $[x, i)$, where $x \le j$, the KNN of v_i in it cannot contain v_j. Thus there does not exist any compact adjacent lists C where $v_j \in C$ in any compact partial range b as C is the KNN of v_i in $[b_l, i)$ contradicts with $b_l \le b_r = C_{min} \le j$. Thus we do nothing.

Case 2: $\mathcal{L}_{min} < j < i$**.** We first update $\mathcal{L}$ as the search key $j < i$ of the current visiting vector v_j is greater than $\mathcal{L}_{min}$, the K-th largest search key of visited vectors smaller than i. To this end, we insert j to $\mathcal{L}$ and remove $\mathcal{L}_{min}$ from $\mathcal{L}$ if there are K + 1 search keys in $\mathcal{L}$.

Next, we generate the compact adjacent lists. Specifically, if there are K search keys in $\mathcal{L}$, we generate a compact adjacent list $\mathbf{G}_{\langle b\rangle}[v_i] = \mathcal{L}$ where $b = (\mathcal{L}_{min-1}, \mathcal{L}_{min}]$ and $\mathcal{L}_{min-1}$ is the previous $\mathcal{L}_{min}$ just removed from $\mathcal{L}$.

Algorithm 3 shows the pseudo-code of the index construction algorithm. For every visit, it is almost the same as Algorithm 2 except that it does not need to slide a fixed-width window and generates a compact adjacent list for each sliding window. It only needs to generate one compact adjacent list when there are K+1 data vectors in $\mathcal{L}$ after inserting j (Line 1). Specifically, it first removes the additional data vector $\mathcal{L}_{min}$ from $\mathcal{L}$ and keep it as $\mathcal{L}_{min-1}$ (a dummy variable for ease of presentation). Then it generates $\mathbf{G}_{\langle b\rangle}[v_i] = \mathcal{L}$ where $b = (\mathcal{L}_{min-1}, \mathcal{L}_{min}]$ (Lines 2 to 4). The process for $j > i$ is symmetric. The rests are the same as in Algorithm 2.

Table 1: Space complexities of different indexes.

index methods	worst	average
AL (adjacent list, brute-force index)	$O(\mathrm{K}n^3)$	$O(\mathrm{K}n^3)$
AL + PR (partial ranges)	$O(\mathrm{K}n^2)$	$O(\mathrm{K}n^2)$
CAL (compact graph index)	$O(\mathrm{K}^2n^2)$	$O(\mathrm{K}^3n\log n)$
CAL + PR (ARKGraph)	$O(\mathrm{K}n^2)$	$O(\mathrm{K}^2n\log n)$
CAL + PR + DC (delta ARKGraph)	$O(n^2)$	$O(\mathrm{K}n\log n)$

Example 7. As shown in Figure 3, consider the left side of the data vector v_9 and K = 2. We will visit $v_6, v_2, v_7, v_3, v_8, v_1, v_4, v_5$ in sequence. Initially, we have $\mathcal{L} = \{0\}$ and $\mathcal{L}_{min} = 0$. During the second visit, we have $\mathcal{L}_{min-1} = 0$ and $\mathcal{L}_{min} = 2$. A compact adjacent list $\mathbf{G}_{\langle b\rangle}[v_9] = \{v_2, v_6\}$ where $b = [1, 2]$ is generated. During the third visit, we have $\mathcal{L}_{min-1} = 2$ and $\mathcal{L}_{min} = 6$. A compact adjacent list $\mathbf{G}_{\langle b\rangle}[v_9] = \{v_6, v_7\}$ where $b = [3, 6]$ is generated. The fourth visit results in no compact adjacent list as $j = 3 < \mathcal{L}_{min} = 6$. During the fifth visit, we have $\mathcal{L}_{min-1} = 6$ and $\mathcal{L}_{min} = 7$. A compact adjacent list $\mathbf{G}_{\langle b\rangle}[v_9] = \{v_7, v_8\}$ where $b = [7, 7]$ is generated. Finally, the sixth, seventh, and eighth visits all lead no compact adjacent list. In total, 3 compact adjacent lists are generated on the left and 5 on the right. In comparison, 11 compact adjacent lists are generated without using the compact partial range.

Theorem 8. *Algorithm 3 is correct and sound. It generates and only generates all the compact adjacent lists in every data vector.*

Complexity Analysis. For each visit, the algorithm generates at most one compact adjacent list, which takes $O(\mathrm{K})$ time and space. Thus the index size complexity is $O(\mathrm{K}n^2)$, as summerized in Table 1. Moreover, based on Equation 1, for each data vector, only $O(\mathrm{K}\log n)$ data vectors are not skipped in expectation. Thus the expected index size is $O(\mathrm{K}^2n\log n)$.

4.3 Delta Compression

For each data vector v_i, consider two consecutive visits v_h and v_j that result in two consecutive compact adjacent lists. As we can see from Algorithm 3, the two compact adjacent lists are almost identical. The latter compact adjacent list can be derived from the former one by adding j and removing $\mathcal{L}_{min}$. Based on this observation, we propose to use delta compression to reduce the size of the compact adjacent list. Specifically, instead of generating the complete compact adjacent list for each visit, we keep the delta only, which are two numbers, j for insertion and $\mathcal{L}_{min}$ for deletion. Note for this purpose, we need to keep track of and maintain the variable $\mathcal{L}_{min}$ in each visit. In this way, our compact graph index is further reduced by K/2 times. The index size and time (w/o sorting) complexities are both $O(n^2)$ after using delta compression. Furthermore, based on Equation 1, after delta compression, the expected index size is $O(\mathrm{K}n\log n)$.

Example 9. Figure 3 on the bottom-right shows the 8 compact adjacent lists generated by Algorithm 3 after delta compression.

5 QUERY PROCESSING

To process an ARKGraph query s, we only need to go over each vector v_i where $i \in s$ and restore its KNN in s. Next we discuss

how to restore the KNN of a vector v_i in a range $s = [x, y]$ where $x \le i \le y$ from the compact graph index, the ARKGraph index, and the delta-compressed ARKGraph index.

Compact Graph Index. To restore the KNN of v_i in $[x, y]$ from the compact graph index, we only need to fetch the compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v_i]$ where $x \in b$ and $y \in e$. Based on Lemma 1, there is one of only one such compact range and compact adjacent list. To facilitate this, we propose to build a two-dimensional segment tree for every vector v. Specifically, for each compact adjacent list $\mathbf{G}_{\langle b,e\rangle}[v]$, we index the interval b in the first dimension and the interval e in the second dimension of the segment tree of v, along with $\mathbf{G}_{\langle b,e\rangle}[v]$. To fetch $\mathbf{G}_{[x,y]}[v_i]$, we query v_i's segment tree using x and y in the first and second dimensions, respectively. Based on Lemma 1, it will hit one and only one compact range and the associated compact adjacent list is returned. As discussed in Section 3, in expectation, each vector generates $O(\mathrm{K}^2 \log n)$ compact adjacent lists. Thus the size of each segment tree is $O\big(\mathrm{K}^2 \log n \log^2(\mathrm{K}^2 \log n)\big)$ and each fetch takes $O\big(\log^2(\mathrm{K}^2 \log n) + |s|\big)$ in expectation.

ARKGraph Index. To restore the KNN of v_i in $[x, y]$ from the ARKGraph index, we only need to fetch the compact adjacent lists $\mathbf{G}_{\langle b\rangle}[v]$ and $\mathbf{G}_{\langle e\rangle}[v]$ where $x \in b$ and $y \in e$ and merge them. For this purpose, for each vector v, we store the compact adjacent lists on the left and right sides of v in two arrays ordered by their compact partial ranges. Then, we can perform binary searches in the two arrays to find the compact partial range containing x and y. Then we merge the two compact adjacent lists and use the top-K as the restored KNN. Based on the discussion in Section 4, the size of each array is $O(\mathrm{K} \log n)$ and each restoring takes $O(\log \log n + \mathrm{K} \log \mathrm{K} + |s|)$ in expectation,

Delta Compressed ARKGraph Index. It is the same as above except an additional decompression step. Once we find the slot in the left array containing the target compact partial range, we checks the precedent slots one by one. We maintain an insertion list A and a deletion list R. For each insertion number, if it is not in R, we add it to A. For each deletion number, we add it to R. In addition, if it is in A, we remove it from A. When there are K numbers in A, we stop and return A for merging. The index size remains the same. In expectation, each fetch takes $O(\log \log n + \mathrm{K} \log n + \mathrm{K} \log \mathrm{K})$.

The above procedure decompresses the compact adjacent lists backwards. We can also decompress it forwards. In this way, we actually do not need to perform the binary search at the beginning. Furthermore, the deletion list is also unnecessarily, which saves the index size by a half. However, it needs to maintain the top-K vectors when decompressing forwards and it may visit many compact partial ranges, which brings significant overhead.

Another alternative way for decompressing indexes the compact partial ranges and their delta in a segment tree. For each search key j, suppose it is added to the compact adjacent list of the compact partial range $b = [b_l, b_r]$ and removed from the compact adjacent list of the compact partial range $b' = [b'_l, b'_r]$. Then, j must reside in the compact adjacent list if the query range is within $[b_l, b'_l)$. Thus we propose to build a segment tree and index the interval $[b_l, b'_l)$ together with the search key j in the segment tree. To fetch $\mathbf{G}_{[x,y]}[v_i]$, we query the segment tree of v_i using x and it must hit and only hit exactly K intervals. The K search keys associated with the K intervals compose the KNN of v in $[x, i)$. The size of the segment tree pair is $O\big(\mathrm{K} \log n \cdot \log(\mathrm{K} \log n)\big)$ and each fetch takes $O(\log \log n + \mathrm{K} \log \mathrm{K} + |s|)$ in expectation.

6 EXPERIMENTS

6.1 Setup

Datasets. We used two large-scale datasets. (1) DEEP1B is an image descriptor dataset [9]. It consists of the projected, PCAed, and normalized activations from the last fully-connected layer of the GoogLeNet model [50], which was pre-trained on the ImageNet classification task [15]. It contains 1 billion 96 dimensional dense vectors. (2) BigGraph is a graph embedding dataset pre-trained on the full Wikidata graph (with 78 million entities) using the PBG model [2, 35]. The dimensionality of the embeddings is 200.

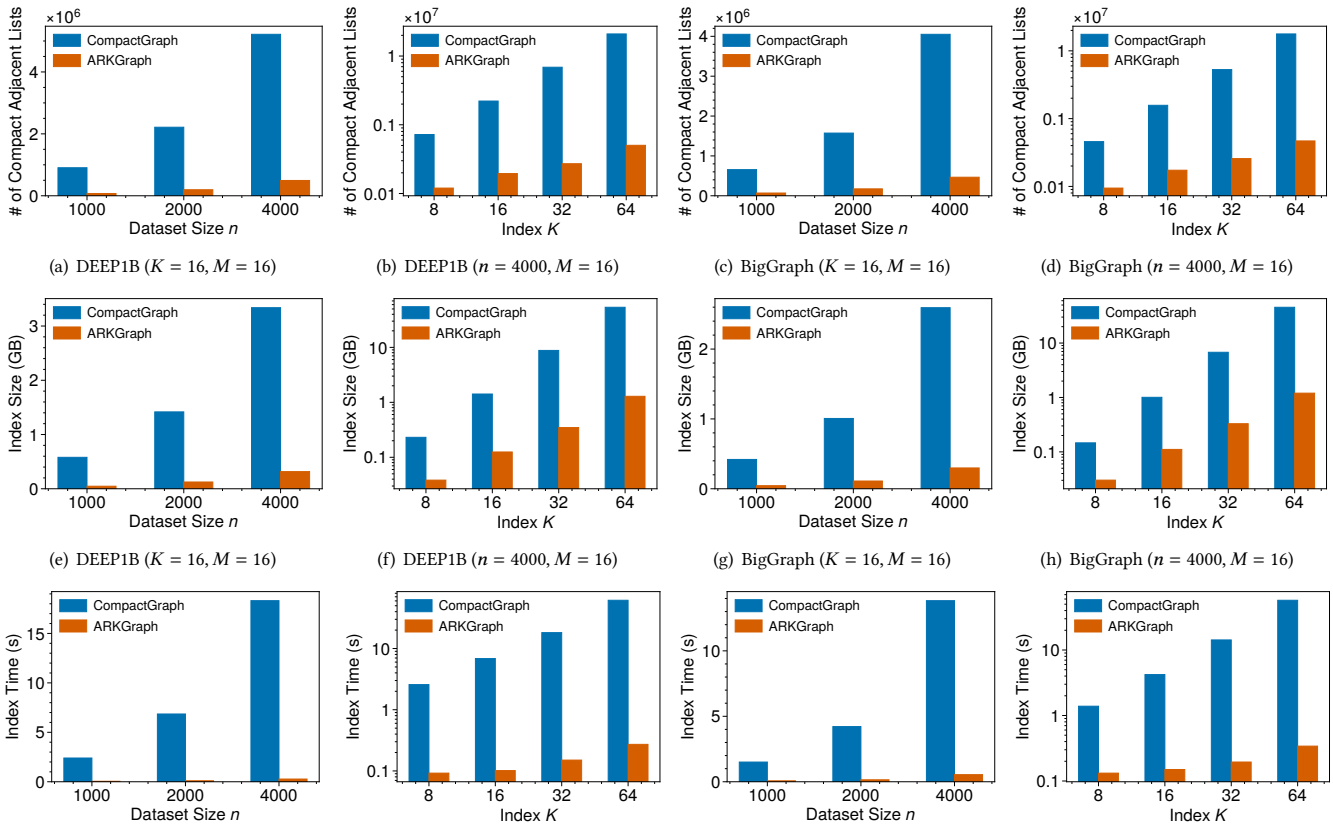
Parameters. There are two primary parameters in our index. First, K is the degree of the all-range approximate K-nearest-neighbor graph. Second, M is the degree of the graph constructed beforehand by NNDescent [16] as discussed in Section 3.3. There are also a couple of parameters during query processing, such as $|s|$, the width of the query range. In addition, the dataset size n is also a parameter.

Environment. All the experiments were conducted on a machine with Intel(R) Xeon(R) Gold 6212U CPU @ 2.40GHz and 64GB memory running Ubuntu 18.04LTS. All methods were implemented in C++ and compiled using g++ 7.5 with -O3 optimization and used OpenMP for parallel computing using 24 threads for all methods.

6.2 Evaluating Index Construction

In this section, we evaluate the index construction. Specifically, we implemented three methods. (1) CompactGraph corresponds to Algorithm 2, which generates all the compact adjacent lists in the compact ranges as the compact graph index; (2) ARKGraph corresponds to Algorithm 3, which generates all the compact adjacent lists in the compact partial ranges as the ARKGraph index; (3) ARKGraphDelta improves ARKGraph by using delta compression to compress the consecutive compact adjacent lists; Note that, we did not include the brute-force index here as it was too large and too time consuming to build. We vary K and the dataset size n and report the index time and index size.

CompactGraph vs. ARKGraph. We first evaluate the numbers of compact adjacent lists generated by CompactGraph and ARKGraph. Figure 6 shows the results. As we can see from the figure, ARKGraph generated much less number of compact adjacent lists than CompactGraph. For example, on DEEP1B dataset, when $K = 16$ and n increase from 1000 to 4000, ARKGraph generated 73,549 and 494,641 compact adjacent lists correspondingly, while CompactGraph generated 905,898 and 5,216,177 compact adjacent lists, which is $10x$ much than ARKGraph. This is because the compact adjacent lists in consecutive compact partial ranges tend not to change and thus are more effective in grouping. Moreover, the gap between CompactGraph and ARKGraph steadily grew when K increased. For example, on DEEP1B dataset, when $n = 4000$ and K increase from 8 to 64, ARKGraph's compact adjacent list increases by 2.4 times and CompactGraph's compact adjacent list increases by 10.1 times. This is consistent with our complexity analysis, which shows that the number of compact adjacent lists in CompactGraph is K times



(a) DEEP1B ($K = 16, M = 16$) (b) DEEP1B ($n = 4000, M = 16$) (c) BigGraph ($K = 16, M = 16$) (d) BigGraph ($n = 4000, M = 16$)

(e) DEEP1B ($K = 16, M = 16$) (f) DEEP1B ($n = 4000, M = 16$) (g) BigGraph ($K = 16, M = 16$) (h) BigGraph ($n = 4000, M = 16$)

(i) DEEP1B ($K = 16, M = 16$) (j) DEEP1B ($n = 4000, M = 16$) (k) BigGraph ($K = 16, M = 16$) (l) BigGraph ($n = 4000, M = 16$)

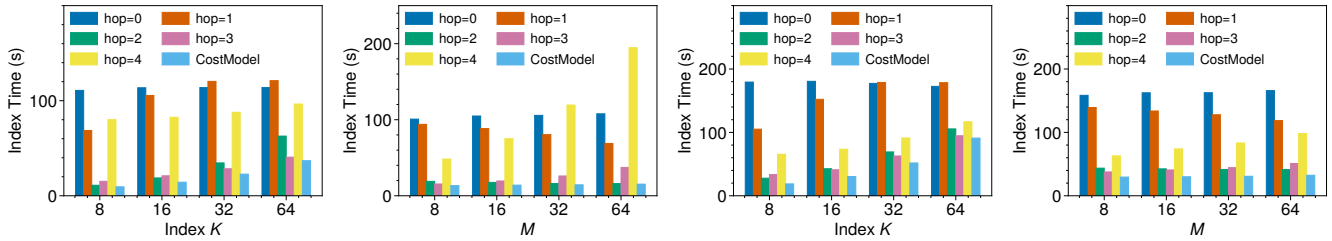
Figure 6: Evaluating index construction of CompactGraph and ARKGraph.

of that in ARKGraph. Next we evaluate the index sizes of CompactGraph and ARKGraph. Figures 6(e, f, g, h) shows the results. We can see similar trends as for the number of compact adjacent lists. This is because the index size is proportional to the number of compact adjacent lists. Moreover, the index size of ARKGraph scaled super linearly with K and n. The growth was faster with K than with n. For example, on DEEP1B dataset, when K increases by 4 times from 16 to 64, the index size increase by 10.9 times. As contrast, when n increases by 4 times from 1000 to 4000 and $K = 16$, the index size only increases by 6.7 times. This is consistent with our analysis, where the index size of ARKGraph on average is $O(K^2 n \log n)$. Finally, we evaluated the index time (excluding the vector sorting time). Figure 6(i, j, k, l) shows the results. As we can see from the figure, the index time of ARKGraph was consistently and significantly smaller than that of CompactGraph by up to 2 orders of magnitudes. For example, on DEEP1B dataset, when the $K = 16, n = 4000$, ARKGraph costs 0.27s to build the index while CompactGraph costs 18.31s. This is because the number of compact partial ranges is much more than the number of compact ranges (K times smaller to be specific).

Evaluating Cost Model for Vector Sorting. Next we evaluate our cost model for efficient vector sorting by distances as discussed in Section 3.3. We implemented the fixed-hop method, which takes an integer x as input and explores all the vectors within x hops. We also implemented our cost model based method CostModel that explores the neighbors dynamically. Figures 7 shows the total index time of ARKGraph equipped with different vector sorting methods. As we can see from the figure our proposed cost model almost always achieved the smallest index time compared with fixed-hop methods with various x. For example, on DEEP1B dataset, when $M = 16, K = 16$, our cost model method costs 14.28s to build the index, and its average hops is 2.49. In contrast, the fixed-2-hops method costs 18.77s and the fixed-3-hops method costs 21.00s. This is because, using a small fixed hop will increase the number of vectors to be scanned later, while using a large fixed hop will incur a long time for exploring the neighbors. Our cost model effectively chooses the number of hops dynamically for every vector (i.e., each vector has a different hop). For example, when $K = 16, M = 16$, the average hops in our cost model method was 2.50.

Evaluating the Impact of M. We observe that the index time of our CostModel method remained roughly the same under different M. For example, as shown in Figures 7(b), on the DEEP1B dataset, when $K = 16$, the index time for $M = 8$ and $M = 16$ were respectively 13.31 seconds and 15.26 seconds. This is because our CostModel method can adaptively choose the number of hops to explore in the MNNG based on the value of M. For instance, on average, our cost model explored 2.82 and 2.12 hops of the MNNG respectively for $M = 8$ and $M = 16$.

Next, we evaluate the impact of M. For this purpose, we vary M in $[8, 16, 32, 64]$ and report the query latency and query accuracy. Figure 8 shows the result. As we can see, the query latency using indexes constructed by different M were roughly the same, while



(a) DEEP1B ($n = 10^5, M = 16$) (b) DEEP1B ($n = 10^5, K = 16$) (c) BigGraph ($n = 10^5, M = 16$) (d) BigGraph ($n = 10^5, K = 16$)

Figure 7: Evaluating the cost model for vector sorting.

the query accuracy increased a bit with the growth of M. For example, on the DEEP1B dataset, when $K = 16$ and $|s| = 100\%$, the query latency for $M = 8$ and $M = 64$ were 11.43ms and 11.26ms respectively, while the query accuracy were respectively 0.974 and 0.995. In a nutshell, the impact of M to the index time, query latency, and query accuracy were small. For simplicity, we set M as K for the rest of experiments.

Evaluating Delta Compression. Next, we evaluate the delta compression. We additionally implemented two alternative ways to organize the deltas as discussed in Section 5. ARKGraphDeltaRaw decompresses the compact adjacent lists forwards, while ARKGraphDeltaST indexes the compact partial ranges and the deltas in a segment tree. Both of them have lower time complexity for query processing while entailing some overheads. Figure 9 shows the index time and index size of the four methods, ARKGraph (without delta compression), ARKGraphDelta, ARKGraphDeltaRaw, and ARKGraphDeltaST. The index time of the four methods was roughly the same, meaning that the delta compression did not incur significant overhead. Moreover, the index time of ARKGraphDelta was a bit less than that of ARKGraph. This is because the delta compression saves data copying time (copying the common part of compact adjacent lists in two consecutive compact partial ranges). As for the index size, we can see, with delta compression, the index size was significantly reduced. For example, on the DEEP1B dataset, when $K = 64$, ARKGraph needs 20GB to store the index and ARKGraphDelta just needs 0.72GB, which is 27.8 times smaller. This is because delta compression avoids storing the same neighbors again and again. Moreover, we observe the gap of the index sizes between ARKGraph and ARKGraphDelta grew linearly with K. For example, on the DEEP1B dataset, when $K = 16$, ARKGraph's index size is 3.6 times larger than ARKGraphDelta, but the gap comes to 27.8 when $K = 64$. This is consistent with our complexity analysis as shown in Table 1. In addition, the index size of ARKGraphDeltaST was much larger than that of ARKGraphDelta. This is because the additional segment tree structure takes a lot of space. The index size of ARKGraphDeltaRaw was roughly half of that of ARKGraphDelta as it does not need to store the deletion list.

6.3 Evaluating Range KGraph Query

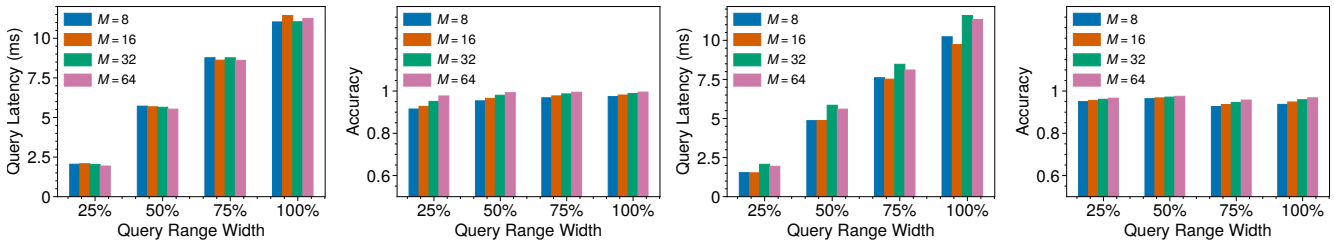
In this section, we evaluate the range KGraph query processing using different indexes. Specifically, we compared ARKGraph (i.e., without delta compression), ARKGraphDelta, ARKGraphDeltaRaw, and ARKGraphDeltaST with a baseline method KGraph. The baseline method NNDescent [16], a popular approximate K-nearest-neighbor graph construction algorithm, constructs a KGraph of vectors in the query range on-the-fly (i.e., it has no index). We vary K, the dataset size n, and the query range width $|s|$, and report the query latency and query accuracy as defined in Definition 1.

Figures 10 show the query latency results. As we can see, all of our methods had a much lower query latency than the baseline method. On DEEP1B dataset, when $K = 16$ and $|s| = 25\%$, ARKGraph needs 1.42ms to restore the KNN while the baseline method needs 1921.69ms, ARKGraph is 1353 times faster. This is because the baseline did not use any index. Among our methods, ARKGraph had the lowest query latency, followed by ARKGraphDelta, while ARKGraphDeltaST had the highest query latency. This is because it does not need to decompress the compact adjacent lists first to restore the KNN. Although the time complexity of ARKGraphDeltaST and ARKGraphDeltaRaw are lower than ARKGraphDelta, they had significant overhead compared with ARKGraphDelta in decompressing. Specifically, ARKGraphDeltaRaw needs to scan from the beginning of the list and maintain the top-K nearest neighbors, while ARKGraphDeltaST needs to query the segment tree. Thus they took longer time than ARKGraphDelta. The difference between ARKGraphDelta and ARKGraph was small. For example, on DEEP1B dataset, when $K = 16$ and $|s| = 100\%$, ARKGraphDelta needs 9.86ms to restore the KNN and ARKGraph needs 6.44ms. Thus the decompressing overhead was affordable, especially the total query latency was very small and the index size after delta compression was significantly reduced (by up to 100 times).

Figure 11 shows the query accuracy results. Note that all of our four methods generate exactly the same KGraph all the time. Thus we only plot one of them, ARKGraphDelta, in the figures. As we can see, the quality of the KGraph produced by our index was comparable to that of the baseline method. Both of them were close 100%. For example on DEEP1B dataset, when $K = 16, n = 10^5$ and $|s| = 75\%$, the restored KGraph accuracy of ARKGraphDelta and baseline were respectively 0.977 and 0.980. This is because our ARKGraph index is a lossless compression of the brute-force index, which consists of a KGraph for every search key range.

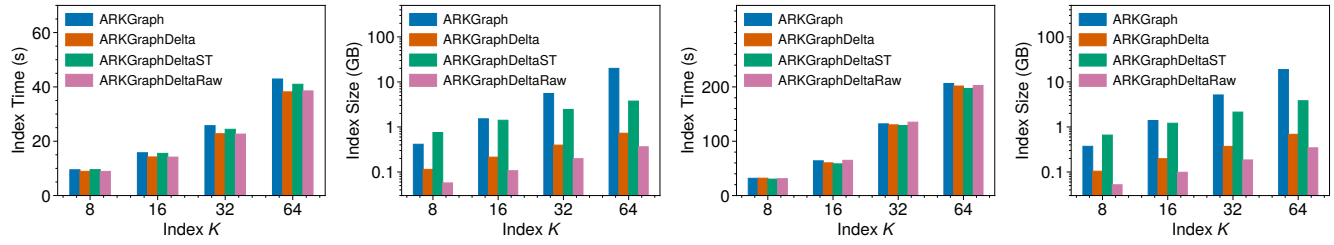
6.4 Scalability

In this section, we evaluate the scalability of our best index ARKGraphDelta and compare it with the baseline method. We vary n the number of vectors in the dataset in $[10^4, 10^5, 10^6]$ and report the index time, index size, query latency, and query accuracy. The results are shown in Figures 12 and 13. As we can see, with the increase in the dataset size, the index time and index size scaled very well. For example, on DEEP1B dataset, when $K = 16$ and n



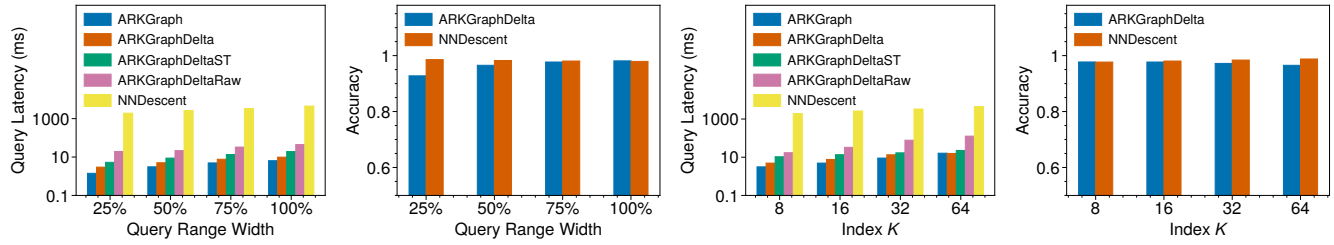
(a) DEEP1B ($n = 10^5, K = 16$) (b) DEEP1B ($n = 10^5, K = 16$) (c) BigGraph ($n = 10^5, K = 16$) (d) BigGraph ($n = 10^5, K = 16$)

Figure 8: Evaluating MNNG M on KGraph query processing.



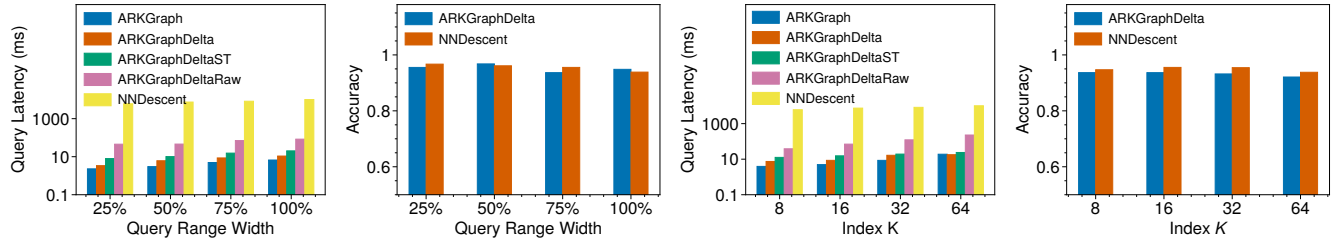
(a) DEEP1B ($n = 10^5, M = 16$) (b) DEEP1B ($n = 10^5, M = 16$) (c) BigGraph ($n = 10^5, M = 16$) (d) BigGraph ($n = 10^5, M = 16$)

Figure 9: Evaluating delta compression methods.



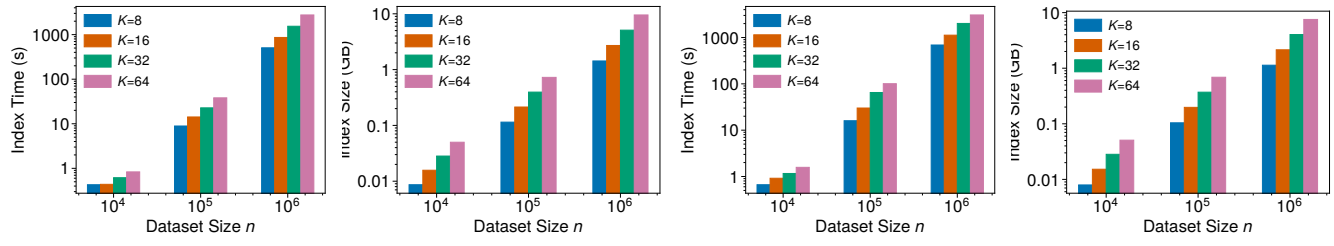
(a) DEEP1B ($n = 10^5, K = 16$) (b) DEEP1B ($n = 10^5, K = 16$) (c) DEEP1B ($n = 10^5, |s| = 75\%$) (d) DEEP1B ($n = 10^5, |s| = 75\%$)

Figure 10: Evaluating range KGraph query processing on DEEP1B.



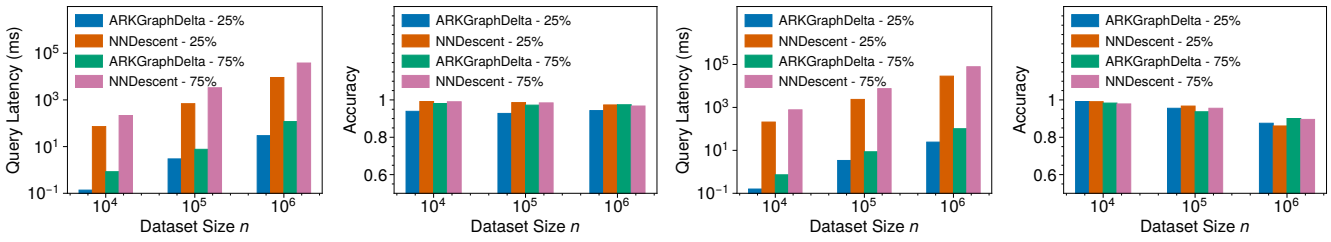
(a) BigGraph ($n = 10^5, K = 16$) (b) BigGraph ($n = 10^5, K = 16$) (c) BigGraph ($n = 10^5, |s| = 75\%$) (d) BigGraph ($n = 10^5, |s| = 75\%$)

Figure 11: Evaluating range KGraph query processing on BigGraph.



(a) DEEP1B ($M = K$) (b) DEEP1B ($M = K$) (c) BigGraph ($M = K$) (d) BigGraph ($M = K$)

Figure 12: Evaluating scalability of ARKGraphDelta index construction.



(a) DEEP1B ($K = 16$) (b) DEEP1B ($K = 16$) (c) BigGraph ($K = 16$) (d) BigGraph ($K = 16$)

Figure 13: Evaluating scalability of range KGraph query processing using ARKGraphDelta.

increases from 10^5 to 10^6, the index time for ARKGraphDelta increases from 14.22s to 864.28s and the index size increases from 0.21GB to 2.71GB. This is consistent with our complexity analysis. Moreover, the query accuracy was close to 100% for different dataset sizes n and different query widths $|s|$. With the increase of the dataset size n or the increase of the query width, the query accuracy almost remained unchanged. For example, on DEEP1B dataset, when $K = 16, |s| = 75\%$, and n increases from 10^5 to 10^6, the corresponding restored KGraph accuracy is 0.972 and 0.974. In the meanwhile, the query latency scaled almost linearly with the dataset size n. For example, under the same conditions as the last example, the corresponding query latency for $n = 10^5$ and $n = 10^6$ is 7.66ms and 11.81ms. The good scalability is attributed to the effectiveness of our index algorithm and query processing algorithm.

7 RELATED WORK

Approximate K-Nearest-Neighbor Graph Construction. There are a few works on KGraph construction [16, 18, 51, 59]. The most popular one is NNDescent [16]. It starts with a random KGraph and then repeatedly refines each vector's neighbors with its neighbors' neighbors. When the process converges, the KGraph is returned. [16] until converge. Tang et al. [51] improves NNDescent for visualizing large-scale high dimensional vectors. Instead of starting with a random KGraph, it proposes to use random projection trees to construct an initial KGraph. EFANNA proposes to use the KD-tree [11] to find an initial KGraph for NNDescent [18]. Zhang et al. [59] proposes to use locality sensitive hashing for KGraph construction.

Approximate Nearest Neighbor Search (ANNS). Due to the well-known "curse of dimensionality" phenomenal [29], tree-structured indexes for multi-dimensional data such as the KD-tree [11], R-tree [25], TV-tree [36], and Quad-tree [17, 48] would not work in high dimensional space. Locality Sensitive Hashing (LSH) [4, 5, 14, 20, 22, 28, 29, 38, 49, 52, 53], product quantization (PQ) [3, 6, 21, 31, 37, 42], and proximity graphs [19, 40] are the cornerstone of almost all the existing index structures and algorithms for ANNS. LSH uses specific hash functions to hash the vectors such that nearby vectors are more likely to be placed in the same bucket than far away vectors [29]. Product quantization (PQ) compresses the high dimensional vectors to small "codes" and the distance of two vectors can be efficiently estimated with their PQ codes by table lookup [6, 10, 21, 23, 31, 32, 37, 42]. The graph-based methods build a graph as the index and use the best-first search [47] to find the approximate nearest neighbors of a query [7, 19, 27, 30, 33, 39, 40]. The most popular graph index is hierarchical navigable small-world graph (HNSW) index [33], which achieves the state-of-the-art performance [1, 8].

Multi-Modal Approximate Nearest Neighbor Search. The multi-modal ANNS query is referred to as the "hybrid query" in AnalyticDB-V [57], "attribute-filtering query" in Milvus (a.k.a., Zilliz) [56], and "subset search query" in Rii [41]. AnalyticDB-V [57] proposes four query plans for the hybrid query and uses a cost model to choose an optimal one. The four query plans are simple permutations of index scan (on the search key) and two variant product quantization (PQ) scans (on the vector attribute). Milvus [56] is a vector database system. In addition to the four query plans developed in AnalyticDB-V, it also implements a partition-based query plan, which partitions the dataset and uses a cost-model to choose an optimal plan from the four query plans for each partition. Matsui et. al [41] propose reconfigurable inverted index (Rii). It scans all the PQ codes in the query range if the query range is smaller than a threshold; otherwise, it uses the traditional inverted index to filter the PQ codes first and then checks the search keys. All these systems follow two trivial strategies, ANNS-first and range-first. ANNS-first is inefficient when the query range is small, while range-first takes a prolonged time when the query range is large.

8 CONCLUSION

We study the all-range approximate K-nearest-neighbor graph in this paper. Given a set of vectors, each associated with a search key value, we aim to build an index that takes a search key range as the query and produces an approximate K-nearest-neighbor graph of vectors in the query range. We develop a series of novel techniques to reduce the index size. We formally prove our ARKGraph index size is $O(Kn \log n)$ on average where n is the number of vectors. We develop efficient indexing algorithm to directly construct the ARKGraph index. It can process range KGraph queries in almost real-time. Extensive experiments on real-world datasets show that ARKGraph index significantly outperformed the baseline method and achieved small index size, low query latency, and good scalability.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation grants #2152908, #2212629 and a research gift from Adobe.

REFERENCES

[1] [n.d.]. ANN Benchmark. http://ann-benchmarks.com/index.html.

[2] [n.d.]. Facebook BigGraph. https://github.com/facebookresearch/PyTorch-BigGraph.

[3] [n.d.]. FAISS. https://github.com/facebookresearch/faiss.

[4] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *FOCS*. 459–468.

[5] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *NeurIPS*. 1225–1233.

[6] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *PVLDB* 9, 4 (2015), 288–299.

[7] Sunil Arya and David M. Mount. 1993. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *ACM/SIGACT-SIAM*. 271–280.

[8] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020). https://doi.org/10.1016/j.is.2019.02.006

[9] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*. 2055–2063.

[10] Artem Babenko and Victor S. Lempitsky. 2012. The inverted multi-index. In *CVPR*. 3069–3076.

[11] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[12] Oren Boiman, Eli Shechtman, and Michal Irani. 2008. In defense of Nearest-Neighbor based image classification. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*. IEEE Computer Society. https://doi.org/10.1109/CVPR.2008.4587598

[13] Maria R Brito, Edgar L Chávez, Adolfo J Quiroz, and Joseph E Yukich. 1997. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters* 35, 1 (1997), 33–42.

[14] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. 253–262.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[16] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.

[17] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9.

[18] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474.

[20] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*. 541–552.

[21] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2946–2953.

[22] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *PVLDB*. 518–529.

[23] Yunchao Gong and Svetlana Lazebnik. 2011. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*. 817–824.

[24] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *KDD (KDD '16)*. 855–864.

[25] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.

[26] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74.

[27] Ben Harwood and Tom Drummond. 2016. FANNG: Fast Approximate Nearest Neighbour Graphs. In *CVPR*. 5713–5722.

[28] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[29] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[30] Jerzy Jaromczyk and G.T. Toussaint. 1992. Relative neighborhood graphs and their relatives. *Proc. IEEE* 80 (10 1992), 1502 – 1517.

[31] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[32] Yannis Kalantidis and Yannis Avrithis. 2014. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*. 2329–2336.

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

[34] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 32. JMLR.org, 1188–1196. http://proceedings.mlr.press/v32/le14.html

[35] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the 2nd SysML Conference*. Palo Alto, CA, USA.

[36] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. 1994. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB J.* 3, 4 (1994), 517–542.

[37] Yingfan Liu, Hong Cheng, and Jiangtao Cui. 2017. PQBF: I/O-Efficient Approximate Nearest Neighbor Search by Product Quantization. In *CIKM*. 667–676.

[38] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *PVLDB*. 950–961.

[39] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[40] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016). arXiv:1603.09320 http://arxiv.org/abs/1603.09320

[41] Yusuke Matsui, Ryota Hinami, and Shin'ichi Satoh. 2018. Reconfigurable Inverted Index. In *ACM Multimedia Conference on Multimedia Conference*. ACM, 1715–1723. https://doi.org/10.1145/3240508.3240630

[42] Yusuke Matsui, Toshihiko Yamasaki, and Kiyoharu Aizawa. 2015. PQTable: Fast Exact Asymmetric Distance Neighbor Search for Product Quantization Using Hash Tables. In *ICCV*. 1940–1948.

[43] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeurIPS*. 3111–3119.

[44] Jeffrey C Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. 1997. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*. 181–194.

[45] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning Distributed Representations of Graphs. *CoRR* abs/1707.05005 (2017). arXiv:1707.05005 http://arxiv.org/abs/1707.05005

[46] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.

[47] Stuart J. Russell and Peter Norvig. 2003. *Artificial intelligence - a modern approach, 2nd Edition*. Prentice Hall.

[48] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (1984), 187–260.

[49] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB* 8, 1 (2014), 1–12.

[50] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *CVPR*. 1–9.

[51] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. 2016. Visualizing Large-scale and High-dimensional Data. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*. ACM, 287–297. https://doi.org/10.1145/2872427.2883041

[52] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*. 563–576.

[53] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.* 35, 3 (2010), 20:1–20:46.

[54] Laurens van der Maaten. 2014. Accelerating t-SNE using tree-based algorithms. *J. Mach. Learn. Res.* 15, 1 (2014), 3221–3245. https://doi.org/10.5555/2627435.2697068

[55] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[56] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. ACM, 2614–2627. https://doi.org/10.1145/3448016.3457550

[57] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[58] Shuicheng Yan, Dong Xu, Benyu Zhang, HongJiang Zhang, Qiang Yang, and Stephen Lin. 2007. Graph Embedding and Extensions: A General Framework for Dimensionality Reduction. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 1 (2007), 40–51. https://doi.org/10.1109/TPAMI.2007.250598

[59] Yan-Ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-Lin Liu. 2013. Fast kNN graph construction with locality sensitive hashing. In *Machine Learning and Knowledge Discovery in Databases - European Conference*. Springer, 660–674.