Appendix

A Implementation Strategy Analysis for HQ

As **§1.1** mentions, existing solutions to HQ are designed in a "decomposition-assembly" model, where a HQ is decomposed to two subqueries—that is, the ANNS is based on a vector index and AF is based on an attribute index. There are currently two implementation strategies regarding the execution order of these two subqueries, i.e., Strategies A and B in Fig. 11. By contrast, our solution works by using a new fusion strategy (Strategy C).

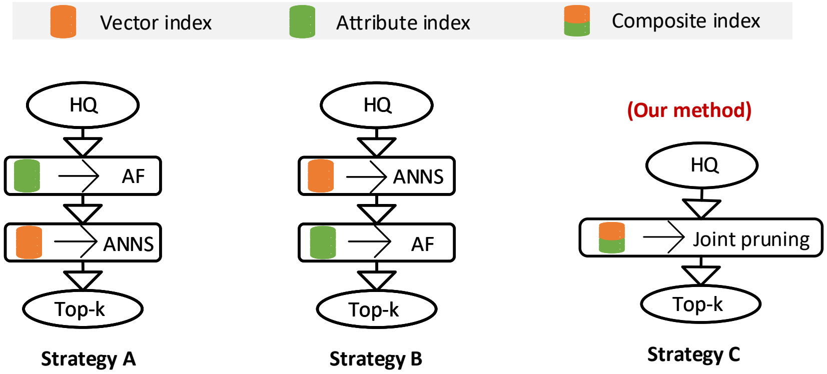


Figure 11: Three different HQ strategies.

Strategy A: "first AF, then ANNS." In general, AF shows higher unit efficiency than ANNS, e.g., in our evaluation, the average overhead time is about $0.02\mu s$ and $0.50\mu s$ for checking whether two attributes are equal and computing the distance of a pair of feature vectors on SIFT1M dataset [1], respectively. Therefore, when AF runs first, ANNS can be performed on the obtained candidate subset rather than the entire object set, which will boost query efficiency to some extent. As a side effect, this strategy suffers from memory and efficiency issues regarding vector index construction for searching similar vectors, as we analyzed in **§1.1** (**L4**). To address this problem, existing efforts [54, 51] encode the feature vectors by PQ [28] for approximating Euclidean distance (Eq. 1). Yet, they yield remarkably low query accuracy, especially compared to ANNS on top of the PG-based vector index [33].

Strategy B: "first ANNS, then AF." This strategy can be classified into two different categories: (1) One is to perform AF after completing ANNS; that is, we only need to check the attributes of the returned top-k objects. (2) The other is to perform AF after each step of ANNS; that is, we need to check the attributes for each object explored during the ANNS. According to our evaluation, (1) is more efficient than (2) in most cases, so that (1) is adopted more widely [32, 29]. One possible explanation is that, current vector indexes are built with feature vectors rather than attributes, prematurely filtering the object that mismatches the attributes in each search step may impair ANNS's performance [52]. Nevertheless, a HQ working off (1) usually requires ANNS to return much more than k candidates for subsequent AF (e.g., it may need to process 300 candidates for obtaining top-10 results), which limits ANNS's efficiency (**§1.1, L3**).

In addition, Strategies A and B must maintain the attribute index and vector index simultaneously (Fig. 11), and separately searching on the two indexes would increase computational overhead (discussed in **§1.1, L1–2**). To overcome these limitations, we design the following novel solution.

Strategy C: "joint AF and ANNS". This strategy carries out ANNS and AF concurrently on a composite index (Fig. 11) that contains both the feature vectors' and attributes' information, which obviously differs from the vector index or attribute index in Strategies A and B. For Strategy C, given a query object and a composite index, we jointly prune unpromising objects with dissimilar vectors and mismatched attributes during query processing, so that we return query results in one step without intermediate candidates. To the best of our knowledge, this is the first work to present a HQ solution in a fused way. With this essential difference, compared to existing solutions we achieve 10× improvement in accuracy and efficiency.

B Intuition for NHQ framework

Rather than answering a HQ by two separate subqueries on different indexes, we propose a NHQ framework, with a well-designed composite index to support both feature vector and attribute constraints. To build a purpose-built index for HQ, it is natural to assemble both feature vector and attribute information in this index (i.e., a composite index). Research suggests that the graph has inherent advantages in fusing complex information [61, 26, 10]. For example, the multimodal knowledge graph embeds various modal data (e.g., image, text) into a graph, thereby benefiting a multitude of downstream tasks [64, 30]. Driven by this, we explore how to integrate various types of information into a graph as our composite index. Recently, the mainstream vector index is implemented based on PG (see Def. 4 in **§3**), which has been proven to offer state-of-the-art performance compared to other index types [18, 63, 62]. However, existing PGs only contain feature vectors' neighborhood relationship while excluding attributes, so they cannot be used directly for building the composite index.

C PG Analysis

In Def. 4, B determines how many neighbors each vertex u_i has. A higher B means more neighbors for u_i, which increases the search space and reduces the efficiency [36]. A lower B may disconnect the PG and affect the accuracy [19]. Different PGs mainly differ in how they implement (2) in Def. 4 (e.g., DPG [33] and NSG [19]). A recent survey [52] analyzes various PG-based index algorithms and their performance, strengths, and pitfalls.

D Composite Index

The ordinary PGs construct the neighborhood relationship between two objects based on their feature vector distance (Eq. 1) alone. By contrast, we take into consideration both the distances of feature vectors and attribute vectors by Eq. 6, thereby generating a new PG and we can take it as the composite index. Algorithm 1 shows how to build a composite index. In our evaluation, B' has an optimal value to achieve a robust query performance. We found that B' is strongly related to the out-degree, and the upper bound of out-degree is always 20 for an optimal B' on most datasets. We can determine this value via a grid search [25].

E Joint pruning

We prune the objects that have both dissimilar feature vectors and mismatched attributes (using the fusion distance, Eq. 6) based on the composite index. This is different from the existing work [54, 51, 32] that prunes feature vector and attribute constraints separately (see Strategies A and B in Fig. 11). Algorithm 2 shows how to do joint pruning.

F Proof for Lemma 1

Proof. We determine the parameters B and B' by a certain upper bound of out-degree R, as shown in Appendix C and D. We use an adjacency list to store the graph index, which contains the vertex ID and R neighbor IDs for each vertex[2]. The index size Λ is:

$$\Lambda = |V| \cdot \varepsilon \cdot (R+1) \quad , \tag{7}$$

where ε is the size per ID. Given a dataset, the composite index and the ordinary PG maintain the same R. Therefore, they have the same index size. □

G Proof for Theorem 1

Proof. Let u be any vertex in the composite index. Suppose there are at least $|N(u)|$ other vertices (denoted by U) that share the same attributes as u. Thus, $\chi(\ell(u), \ell(o)) = 0$ for any $o \in U$. By Eq. 6,

---

[2] We add padding for assignment when $|N(u)| < R$ for any vertex $u \in V$.

$\mathit{\Gamma}(u, o) = \delta(\nu(u), \nu(o))$ with the proposed weight configuration. Hence, $\Omega_{min} \leq \mathit{\Gamma}(u, o) \leq \Omega_{max}$. We choose U as the neighbors of u and induce ①. If we choose other vertices outside U, we have two cases: Case 1: if a vertex v has the same attributes as u, then $\Omega_{min} \leq \mathit{\Gamma}(u, v) \leq \Omega_{max}$; Case 2: otherwise, $\Omega_{min} < \mathit{\Gamma}(u, v) \leq 2 \cdot \Omega_{max}$ for a vertex v. We exclude vertices with $\mathit{\Gamma}(u, v) > \Omega_{max}$ based on the B' bound. Therefore, $\Omega_{min} \leq \mathit{\Gamma}(u, o) \leq \Omega_{max}$.

Let u be any vertex in the ordinary PG. We select $|N(u)|$ neighbors with the smallest $\delta(\nu(u), \nu(o))$ for any $o \in N(u)$. We have two cases: Case 1: if all $o \in N(u)$ have the same attributes as u, then $\Omega_{min} \leq \mathit{\Gamma}(u, o) \leq \Omega_{max}$; Case 2: otherwise, $\Omega_{min} < \mathit{\Gamma}(u, v) \leq 2 \cdot \Omega_{max}$ for a vertex v. The ordinary PG only considers the B bound based on feature vector. Thus, $\Omega_{min} \leq \mathit{\Gamma}(u, o) \leq 2 \cdot \Omega_{max}$.

□

H Proof for Theorem 2

Proof. Let q be the query object. The composite index and the ordinary PG have the same initial R (i.e., P). We consider two cases: Case 1: If R does not change in the search procedure, then the $Recall@k$ is the same; Case 2: If we update R with at least one vertex, then the composite index and the ordinary PG have different updates. The composite index uses the fusion distance to replace a vertex in R with a more similar vertex to q in both feature vector and attributes. The ordinary PG uses only the feature vector distance to replace a vertex in R with a closer vertex to q in feature vector, ignoring the attributes. Therefore, the composite index has a better R and a higher $Recall@k$. □

I Analysis for Current Edge Selection Strategies.

Edge selection is a key step in building a PG [52]. Given an object set $\mathcal{S}$, we obtain the neighbors of each object e in $\mathcal{S}$ based on an edge selection strategy. Different strategies produce noteworthy index structure discrepancies for a PG, thereby impacting search performance on the PG [19, 33].

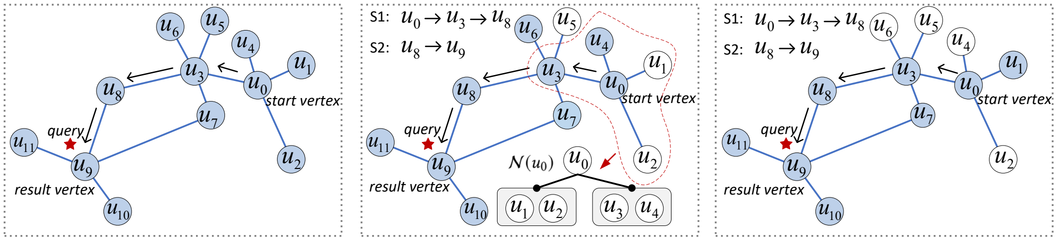
Intuition. In general, existing PGs focus on two factors during edge selection: *distance between two vertices* (D1) and *distribution of all vertices* (D2) [52]. Early PGs such as NSW [35] and KGraph [14] only consider D1 when selecting edges—that is, each vertex is connected with a certain number of its nearest neighbors. [34] argued that only considering D1 would lead to redundant computations and impair search efficiency, as illustrated in Example 5.

Example 5 *Fig. 3(a) illustrates a situation where four nearest neighbors are connected to u_i. We divide u_i's neighborhood into four parts (P1–P4) by a black solid line, and the neighbors located in each part will guide the search to approach the query object q along this orientation. Suppose we are going to expand the search space from u_i. Because most of u_i's neighbors are in the same area (such as u_0–u_2 in P1), we cannot use these neighbors to guide a search toward q that is located in different areas (e.g., P3). Instead, we need extra computation to find a new route to q's area, which adds computational overhead and reduces query efficiency.*

Recent PGs add D2 to edge selection, which diversifies the neighbors' direction while ensuring a similarity of neighbors [33], thereby improving the search performance. A recent experimental study [52] concluded that state-of-the-art PGs connect neighbors in more directions through the edge selection strategy of relative neighborhood graph (RNG).

Definition 6 ***RNG [49].*** *A RNG $G = (V, E)$ w.r.t a given object set $\mathcal{S}$ holds that Def. 4 (PG's definition), and for any two vertices u_i, u_j from V, we have an edge $u_iu_j \in E$, iff $\delta(\nu(u_i), \nu(u_j)) < \delta(\nu(u_i), \nu(u_k))$, or $\delta(\nu(u_i), \nu(u_j)) < \delta(\nu(u_k), \nu(u_j))$, where $\forall\ u_k \in V$ is a vertex that satisfies $u_k \neq u_i \neq u_j$.*

Example 6 *As Fig. 3(b) shows, the yellow lune is formed by two circles' intersection. For example, the small lune is formed by two circles with vertices u_i and u_0 as the centers and distance $\delta(\nu(u_i), \nu(u_0))$ as the radius. We add an edge between u_i and u_0, iff the lune formed by u_i and u_0 does not include any other vertices, which is equivalent to the condition of RNG's edge selection in Def. 6 (refer to [6] for proof). Under this rule, RNG's edge selection prevents u_i's neighbors from congregating in the same area, so that there is more opportunity to link neighbors in different areas (such as u_5 in P4) [33]. However, RNG's edge selection is still problematic, e.g., the vertices in P2*



(a) Greedy search (b) TOGG (c) Our routing

Figure 12: Different routing strategies on PG. The white vertices do not need to, but the blue ones must calculate the distance from the query.

and P3 cannot be connected to u_i in Fig. 3(b) because they violate the above rule. For instance, the lune formed by u_3 and u_i already includes a vertex u_2, so we cannot add an edge between u_3 and u_i. As a result, for the case that query object q in the area of P2 or P3, we still need more computations for finding a new route to q's area.

J Proof for Theorem 3

Proof. Let G be a graph with vertices and edges, and let $N(u)$ be the set of neighbors of a vertex u in G. Suppose that there exist two neighbors o and v of a vertex u in G, such that the angle between o and v regarding u (denoted by $\sigma(o,u,v)$) is less than $\pi/3$. We consider the triangle $\triangle ouv$, which has three angles: $\sigma(u,o,v)$, $\sigma(u,v,o)$, and $\sigma(o,u,v)$. By the triangle angle sum theorem, the sum of these three angles is $180°$, or π radians. Therefore, the sum of $\sigma(u,o,v)$ and $\sigma(u,v,o)$ is more than $(2\cdot\pi)/3$. We have the following two cases by comparing the edge lengths of uo and uv.

Case 1: If $uv > uo$, then o is added to $N(u)$ before v. When we check if v can be added to $N(u)$, we use the landing zone of u and one of its current neighbors in $N(u)$. The landing zone of u and o, denoted by $L(u,o)$, is the region where a vertex can be added to $N(u)$. Let $U(u,o)$ be the perpendicular bisector of the line segment between u and o, and let $H(o,u)$ be the half plane that contains o and is bounded by $U(u,o)$. Then, we have v is in $H(o,u)$. Since $L(u,o) = H(u,o) \setminus B(u,\delta(\nu(u),\nu(o)))$, where $B(u,\delta(\nu(u),\nu(o)))$ is a ball centered at u with radius $\delta(\nu(u),\nu(o))$, that is, $v \notin L(u,o)$. So, v cannot be added to $N(u)$, which contradicts the assumption ($v \in N(u)$).

Case 2: If $uv < uo$, we can swap v and o and get the same conclusion. □

K Analysis for Routing Strategies

Routing is a key step of searching on a PG [58]. Given a PG and a query object, the process of finding the query results is implemented using a proper routing strategy that determines a routing path from the start vertex to the result vertex [58], e.g., the path indicated by the black arrows in Fig. 12. Obviously, the routing strategy directly affects the efficiency and accuracy of searching on the PG [39].

Intuition. Most existing PGs adopt a greedy search as their routing strategy, which leads to some redundant computations (see Example 7) [4, 31]. Some machine learning optimizations [31] are used to mitigate this problem, which achieve better *Speedup* vs *Recall* trade-off at the expense of more index processing time and memory [52]. A recent study [58] specifies two search stages on which routing has different requirements, i.e., *the stage far from the query object* (S1) and *the stage closer to the query object* (S2). In S1, it claims that we should quickly locate the query object's neighborhood (for efficiency requirement), while in S2, we should focus on comprehensively visiting vertices nearest to the query object (for accuracy requirement). Therefore, it designs a tailored routing strategy for each stage, to form a two-stage routing strategy, called TOGG. However, TOGG attaches a tree-based index (e.g., KD-tree is used) to organize the neighbors of each vertex (see Example 7), which increases the index construction time and memory overhead.

Our routing. As a simple and effective solution, we design a random TOGG. In S1, for each visited vertex u_i, rather than obtaining partial neighbors of u_i for distance calculation by an additional tree (as TOGG did), we randomly select u_i's $\lceil R/h \rceil$ ($1 \le h \le R$, where R is the upper bound of out-degree) neighbors from $N(u_i)$ for distance calculation and quickly approach the query object's neighborhood. While in S2, we evaluate the distance of all neighbors in $N(u_i)$ to the query object to ensure query accuracy, which is the same as the greedy search. Differ from TOGG, our routing can achieve the comparable accuracy but does not introduce additional index management overhead according to our evaluation. We emphasize that this random strategy in S1 does not impair the accuracy as most of vertices do not need to calculate the distance from the query and a small number of possibly missed vertices can be made up in S2.

Example 7 *Fig. 12 shows the routing process (indicated by black arrows) from the start vertex u_0 to the result vertex u_9 working off different routing strategies. When we use greedy search as the routing strategy (Fig 12(a)), neighbors of each visiting vertex are explored fully; e.g., for u_0, the distances from all vertices in $N(u_0)$=$\{u_1, u_2, u_3\}$ to the query object are computed. As a result, the greedy search has 12 distance calculations. In Fig. 12(b), TOGG organizes each vertex's neighbors ($N(u_0)$) in a tree-based index. In S1 ($u_0 \rightarrow u_3 \rightarrow u_8$), TOGG selects the next hop based on each visited vertex's neighbor index. In S2 ($u_8 \rightarrow u_9$), TOGG uses a similar routing to the greedy search. Finally, TOGG only needs nine distance calculations. Instead of an additional tree-based index for each vertex's neighbors, our routing randomly selects $\lceil R/h \rceil$=2 neighbors (when we set R=4 and h=2) for computing the distance to the query object in S1 and performs the greedy search in S2. Similar to TOGG, our strategy requires only eight distance calculations in Fig. 12(c). It is worth noting that our routing avoids some unnecessary calculations and improves query efficiency without extra index processing and memory overhead.*

L Proof for Theorem 4

Proof. Let q be the query object. We start with the same initial result set R (i.e., P) as the current greedy search. In S1, we have two possible cases. Case 1: We can update R in S1 and get a better initial R for S2. Note that our routing in S2 is the same as the current greedy search. Case 2: We cannot update R in S1 and we have the same initial R as the current greedy search in S2.

M Implementation Details for Two NPG Algorithms

We present two NPGs, named NPG_{nsw} and NPG_{kgraph}, based on two mainstream PGs, NSW [35] and KGraph [14], by using our edge selection and routing strategies. To be more precisely, we construct two NPGs through our edge selection and conduct the search on them through our routing.

NPG_{nsw}. NPG_{nsw} is constructed by inserting an object incrementally. Specifically, a newly inserted object $e_i \in \mathcal{S}$ (corresponding to a vertex u_i) is regarded as a query object, so we conduct a greedy search [19] to obtain l vertices closest to u_i as u_i's candidate neighbors $C(u_i)$ from the NPG_{nsw} built on previously inserted objects [35]. Then we apply our edge selection to form u_i's neighbors $N(u_i)$. We repeat these operations for each object in $\mathcal{S}$, to eventually yield a NPG_{nsw} built on $\mathcal{S}$.

NPG_{kgraph}. It is built by iteratively updating neighbors. For each $e_i \in \mathcal{S}$, we randomly generate its l initial candidate neighbors $C(u_i)$. To make the vertices in $C(u_i)$ are closer to u_i, we refine $C(u_i)$ through the neighbors of u_i's neighbors, as *the neighbors of a vertex's neighbors are likely to be neighbors of the vertex* [14]. Then we obtain $N(u_i)$ by using our edge selection in the final $C(u_i)$. Specifically, we execute the following process until the graph quality (see Def. 7) [7] reaches a preset threshold (in our experiment, 0.8 is enough to achieve a good performance): for any $u_j \in C(u_i)$ and $u_k \in C(u_j)$, we add u_k to $C(u_i)$, if $\delta(u_i, u_k) < \delta(u_i, u_t)$, where u_t is the farthest vertex to u_i in $C(u_i)$. After the iteration completes, for each $u_i \in V$, its neighbors $N(u_i)$ are produced from $C(u_i)$ via our edge selection strategy.

Definition 7 ***Graph quality [52].*** *Given a PG $G = (V, E)$, we define the graph quality Q_G of G as the mean ratio of the number of u's neighbors (i.e., $N(u)$) in $M(u)$ over $|M(u)|$ for all $u \in V$ (Eq. 8), where $M(u)$ is the k nearest vertices of u in $(V \setminus \{u\})$.*

$$Q_G = \frac{1}{|V|} \sum_{u \in V} \frac{|N(u) \cap M(u)|}{k} \tag{8}$$

Given the two constructed NPGs, we can apply our routing strategy on them directly to return query results efficiently, starting from a randomly acquired seed vertex.

Remarks. In addition to NSW and KGraph, our edge selection and routing also can be applied to many other PGs, e.g., SPTAG [37]. So, we easily obtain a new PG by improving the basal PG with our edge selection and routing (just like we built NPG_{nsw} and NPG_{kgraph}). Experiment shows that our NPGs yield significant performance improvement, comparing to the basal PGs.

N NPG-Based HQ Methods

We integrate the proposed NPGs (i.e., NPG_{nsw} and NPG_{kgraph}) into our NHQ framework, thereby yielding two practical HQ methods; namely, NHQ-NPG_{nsw} and NHQ-NPG_{kgraph}.

N.1 NPG-Based Composite Index

We modify the build process of NPGs by changing its original Euclidean distance to the fusion distance (Eq. 6). So, NPG_{nsw} and NPG_{kgraph} can be deployed in NHQ to serve as a composite index. We emphasize that our NHQ framework is practically flexible, as it is friendly to existing PGs as well as custom-optimized PGs.

N.2 Joint Pruning Optimization

Driven by our routing strategy, we optimize the joint pruning of NHQ by performing a two-stage search (stages S1 and S2 mentioned in Sec. 4.2) on a composite index. Specifically, in S1, our search only visits $\lceil R/h \rceil$ neighbors of each vertex in the search path for efficiency; thus we modify line 5 of Alg. 2 (joint pruning) to randomly visit each vertex's $\lceil R/h \rceil$ neighbors, and execute the loop in lines 2–8 until it falls into the local optimum (i.e., it reaches the vicinity of the result vertices). In S2, we set $C = C \cup R$, and continue to perform lines 2–8 (without modifying line 5) to visit all neighbors of the vertices in the search path comprehensively. Finally, R is returned as the query results.

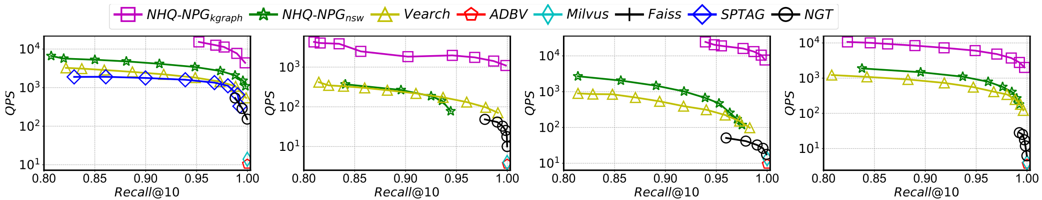
Remarks. (1) In our optimized joint pruning, different stages have specific requirements when searching on a composite index. In S1, it quickly reaches a small area of objects that are similar to the given query object's feature vector and attributes; while in S2, it accurately obtains top-k objects with similar feature vectors and matched attributes. (2) Given the NHQ-NPG_{kgraph} and NHQ-NPG_{nsw} built in Sec. N.1, we can apply the optimized joint pruning directly on them to return HQ results efficiently.

O Evaluation Setting

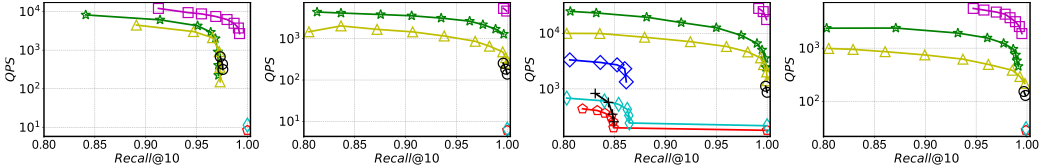
Datasets. In our experimental datasets, the first nine public datasets are composed of high-dimensional feature vectors, which do not originally contain attributes; so, we generate attributes for each object in these datasets following the same method in [51, 56]. For example, we add attributes such as date, location, size to each image on SIFT1M to form an object set having both feature vectors and attributes. Unless otherwise specified, we set the dimension of attributes (m) as 3 by default. *Paper* is an in-house dataset, each object denotes an individual academic paper that consists of a feature vector extracted from the textual content and structured attributes (e.g., affiliation, venue, and topic). We summarize their main characteristics in Tab. 1. LID indicates local intrinsic dimensionality, and a larger LID value implies a "harder" dataset.

Compared methods. We compare our HQ methods with six existing ones that have been used in many high-tech companies.

- **ADBV** [54] is a cost-based HQ method proposed by Alibaba. It optimizes IVFPQ [28] for ANNS.
- **Milvus** [38, 51] divides the object set through frequently used attributes, and deploys ADBV [54] on each subset.
- **Vearch** [29, 32] is developed by Jingdong, which implements the HQ working off Strategy B.
- **NGT** [59] is a ANNS library released by Yahoo Japan, which answers a HQ to conduct attribute filtering atop the candidates recalled by NGT (Strategy B).
- **Faiss** [17] is a ANNS library developed by Facebook, which answers a HQ based on IVFPQ [28] and Strategy A.



(a) SIFT1M (b) GIST1M (c) GloVe (d) Crawl



(e) UQ-V (f) Msong (g) Audio (h) Enron

Figure 13: *QPS* vs *Recall* of different HQ methods.

- **SPTAG** [37] is a PG-based ANNS library from Microsoft, which answers HQ on Strategy B.
- **NHQ-NPG_{nsw}** and **NHQ-NPG_{kgraph}** are our HQ methods based on NHQ framework integrating two NPGs.

Metrics. For index build performance, we record the index build time and index size. We evaluate the search efficiency, accuracy and memory overhead to demonstrate search performance. In terms of ANNS, the search efficiency can be measured by *queries per second* (*QPS*) and *Speedup*; *QPS* is the ratio of the number of queries ($\#q$) to the search time (t), i.e., $\#q/t$; *Speedup* is defined as $|\mathcal{S}|/NDC$, where $|\mathcal{S}|$ is the object set's size and is also the total number of distance calculations of the linear scan for a query, and *NDC* is the number of distance calculations of searching on a PG. We use the *Recall* rate to evaluate the search accuracy, which is measured by Eq. 3. As for HQ, *QPS* is more suitable for evaluating search efficiency, because different methods' calculation cost is distinguishing. Unlike ANNS, attribute constraints are added to the *Recall* rate formula in Eq. 3 for HQ processing, i.e., the elements in $R \cap T$ have exactly the same attributes as the query object. In addition, *selectivity* is used to evaluate the scalability for the attribute constraints with different difficulty, it is defined as $1 - |P|/|\mathcal{S}|$ in [54], where $|P|$ is the number of objects that match the given attributes, and $|\mathcal{S}|$ is the total number of objects.

Implementation setup. Most methods' codes are publicly accessible online, otherwise we implement the corresponding methods according to their papers. Given that all the compared approaches have parallel versions and SIMD, prefetching instructions' optimizations in their index construction codes, we build all the indexes in parallel with 64 threads and turn on these time-saving optimizations. However, considering that not all algorithms support the parallelization of a single query, we use a single thread to perform search, which is a mainstream setting in related work [19, 18].

All codes are written in C++, and are compiled by g++ 6.5. All experiments are conducted on a Linux server with an Intel(R) Xeon(R) Gold 6248R CPU at 3.00GHz, and a 755G memory. We report the average results of all indicators by performing three repeated trials.

Parameters. Because doing parameters' adjustment in the entire dataset may cause overfitting [19], we randomly sample a certain percentage of data from the entire dataset to form a validation dataset and search for the optimal values of all the adjustable parameters on each validation dataset.

P HQ performance

Fig. 13 shows that our methods outperform existing methods in terms of *QPS* vs *Recall* trade-off on all datasets. For example, NHQ-NPG_{kgraph} achieves two to three orders of magnitude higher *QPS* than others when *Recall@10* > 0.99. Our methods also have robust and scalable search performance, especially on harder datasets (e.g., GloVe), where other methods (e.g., NGT) struggle. We do not include Faiss and SPTAG in most charts, because their recall is below 0.8 regardless of the hyper-parameters. Moreover, Milvus and ADBV have no *Recall* data between (about) 0.8 and 0.99, as their PQ-based plans have limited *Recall*; they resort to a linear scan to reach a high *Recall* (*Recall@10*

Table 3: Index build time of different HQ methods.

Algorithm	Build Time (s)							
	SIFT1M	GIST1M	GloVe	Crawl	UQ-V	Msong	Audio	Enron
ADBV	1,614	1,812	1,838	2,895	1,598	1,627	478	611
Milvus	2,131	2,526	2,645	3,389	2,116	2,113	607	933
Vearch	108	519	150	406	**21**	47	**1**	43
NGT	27	1,124	806	2,745	31	73	**1**	20
Faiss	1,591	1,721	1,838	2,791	1,598	1,627	478	611
SPTAG	456	2,690	609	1,555	614	1,574	27	276
NHQ-NPG_{nsw}	**17**	121	**83**	406	40	207	2	23
NHQ-NPG_{kgraph}	25	**70**	189	**188**	25	**61**	**1**	**3**

Table 4: Index size of different PG-based methods.

Algorithm	Index Size (MB)							
	SIFT1M	GIST1M	GloVe	Crawl	UQ-V	Msong	Audio	Enron
Vearch	691	3,903	736	2,832	1,141	1,904	50	526
NGT	672	3,939	665	2,688	1,171	1,803	49	530
SPTAG	656	3,830	650	2,611	1,144	1,756	48	512
NHQ-NPG_{nsw}	648	3,745	550	3,050	1,098	1,786	52	529
NHQ-NPG_{kgraph}	**561**	**3,709**	**491**	**2,346**	**1,041**	**1,678**	**42**	**500**

=1). NHQ-NPG_{kgraph} can easily obtain a high *Recall* with high *QPS*, so its *QPS* is too high to show in the figure when *Recall@10* $<$ (about) 0.9 on most datasets.

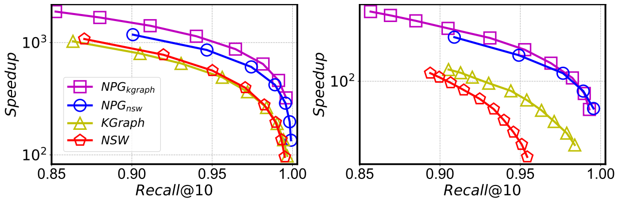
Analysis. Existing methods use "decomposition-assembly" model (Fig. 11 in Appendix A), which separate AF and ANNS, and limit their HQ ability. Ours combine AF and ANNS by fusing feature vectors and attributes into a composite index. By pruning jointly, our methods avoid redundant computation and improve the query efficiency.

Q Index Build Performance

In Tab. 3, our approaches take the least time to complete index building on almost all datasets. Regarding the index size in Tab. 4, NHQ-NPG_{kgraph} is lower than other methods on most datasets. In general, the higher the dataset's LID, the longer it takes to complete index building, and the larger the index size. For PQ-based schemes (such as ADBV), clustering on large-scale high-dimensional vectors results in low index building efficiency; however, these methods' index size are tiny because they compress raw vectors into short codes. By contrast, PG-based methods rely on raw feature vectors, which lead to a larger index size. Nevertheless, our methods show the smallest index size among them, which is because our edge selection eliminates numerous redundant neighbors (Example 4). It is worth noting our methods reach the state-of-the-art HQ performance with the highest index building efficiency, which is the priority of most practical applications [19].

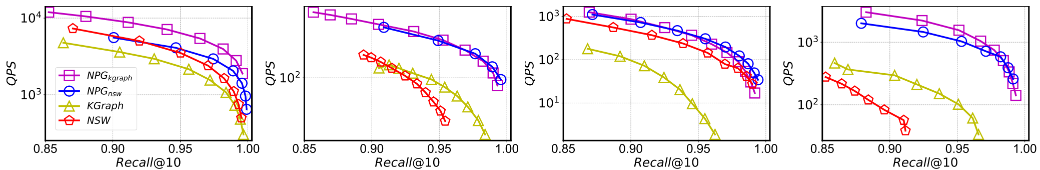
R ANNS performance of NPG

We compare the performance of NPG algorithms and the basal PGs (i.e., KGraph and NSW) to verify the effect of our edge selection and routing strategies. As Fig. 15 shows, NPG_{kgraph} and NPG_{nsw} achieve better accuracy vs efficiency trade-off on all datasets. As the dimension increases, the search efficiency of each PG decreases; for example, from SIFT1M to GIST1M ((a) and (b) in Fig. 15), *QPS* drops by about one order of magnitude. This is because a PG [52] cannot capture the neighborhood relationship of a high-dimensional dataset well, which lowers the search performance (i.e., "curse of dimensionality" [33]). From the results in Appendix T, we see that less distance calculations lead to higher *QPS*. For searching on a PG, most of the search time is spent on calculating the vector distance [63]. NPGs diversify the neighbors' distribution on the basal PGs and use efficient routing strategy, thus improving the search performance.



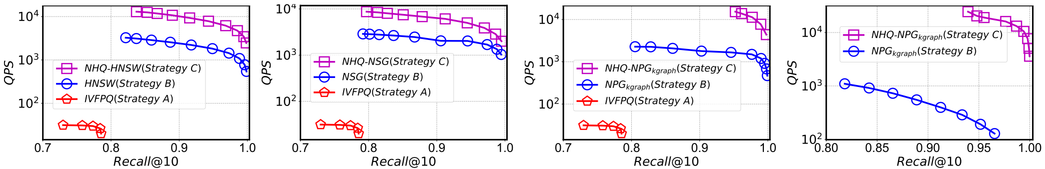
(a) SIFT1M (b) GIST1M

Figure 14: *Speedup* vs *Recall* of different PG-based methods.



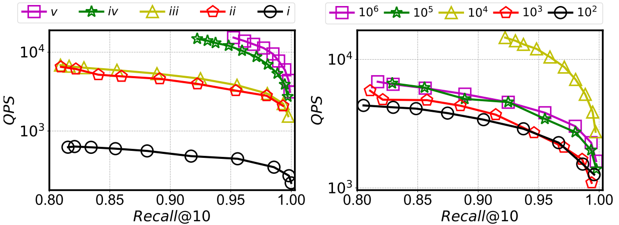
(a) SIFT1M (b) GIST1M (c) GloVe (d) Crawl

Figure 15: ANNS performance of different PG-based methods.



(a) HNSW (SIFT1M) (b) NSG (SIFT1M) (c) NPG_{kgraph} (SIFT1M) (d) NPG_{kgraph} (GloVe)

Figure 16: HQ performance of different strategies.



(a) Weight settings (b) ω_ℓ/ω_ν

Figure 17: Performance under different ω_ν and ω_ℓ. In (a), curve i: $\omega_\nu = \chi/(\delta+\chi)$, $\omega_\ell = \delta/(\delta+\chi)$; curve ii modifies χ by Eq. 9 following curve i; curve iii: $\omega_\nu = \delta_{max}^{-1}$, $\omega_\ell = \chi_{max}^{-1}$; curve iv: $\omega_\ell/\omega_\nu = 10^4$; and curve v: $\omega_\nu = 1$, $\omega_\ell = \delta/\chi_{max}$.

S Validation of NHQ framework.

We test the universality of our NHQ (Strategy C) by applying the HQ model with "decomposition-assembly" to different PGs (including HNSW [36], NSG [19], and our NPG_{kgraph}) based on "first ANNS, then AF" (Strategy B in Appendix A). We also use IVFPQ [28] in "first AF, then ANNS" (Strategy A in Appendix A) following state-of-the-art implementation [54, 51] because Strategy A does not support PG (**L4** in **§1.1**). We combine HNSW and NSG with NHQ to form NHQ-HNSW and NHQ-NSG, respectively. As Fig. 16 shows, NHQ outperforms other strategies on different PGs, and keeps stable *QPS* advantage on different datasets; however, the HQ based on Strategy B decline sharply on dataset with higher dimension (e.g., GloVe). Due to IVFPQ's limitations, the HQ based on Strategy A have low accuracy, e.g., $Recall@10 < 0.8$ on GloVe.

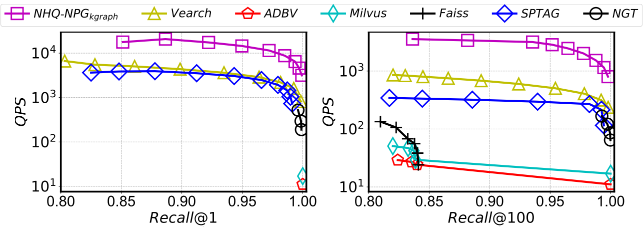
T Speedup Evaluation of PG

Fig. 14 shows the *Speedup* vs *Recall* results on two datasets. We can see that the fewer the number of distance calculations, the larger the *QPS*.

U Weight Analysis

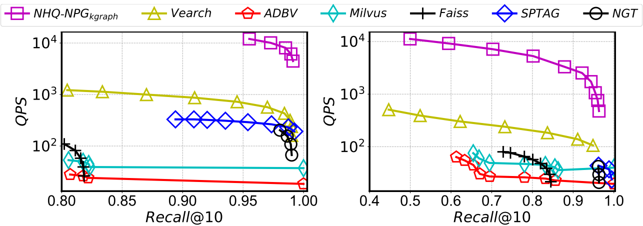
For NHQ, ω_ν and ω_ℓ in Eq. 6 are a pair of parameters that regulate the importance of $\delta(\nu(e_i), \nu(e_j))$ (abbr. as δ) and $\chi(\ell(e_i), \ell(e_j))$ (abbr. as χ) in $\Gamma(e_i, e_j)$ (abbr. as Γ), which impacts the query performance. In the following, based on experimental observations, we evaluate the different settings' effects on performance. We only show the results of NHQ-NPG_{kgraph} on SIFT1M, considering other datasets and methods working off NHQ show similar phenomena.

In general, we treat feature vectors and attributes equally; that is, δ and χ have the same contribution to Γ. We set $\omega_\nu = \chi/(\delta+\chi)$ and $\omega_\ell = \delta/(\delta+\chi)$, so $\Gamma = 2 \cdot \delta \cdot \chi/(\delta+\chi)$ is the harmonic mean of δ and χ. As Fig. 17(a) (**curve i**) shows, this setting does not perform well for HQ. We found that $\chi = 0$ is a common occurrence (two objects possessing the same attributes), and then $\Gamma = 0$ no



(a) SIFT1M ($k = 1$) (b) SIFT1M ($k = 100$)

Figure 18: *QPS* vs *Recall* comparison under different k.



(a) SIFT1M ($m = 6$) (b) SIFT1M ($m = 9$)

Figure 19: Performance under different attribute dimensions.

matter what value δ is, which contradicts the same contribution of the two. To resolve this, we adjust χ as

$$\chi = \begin{cases} 1 & \chi = 0 \\ \chi \cdot c & \chi \neq 0, c > 1 \end{cases}, \tag{9}$$

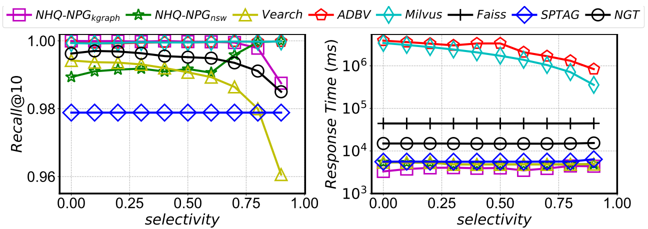
where c is a given constant, and it holds that $c > 1$. This alleviates the defect caused by $\chi = 0$, and it significantly improves the HQ performance (**curve ii**, $c = 100$). Nevertheless, a suitable c is difficult to obtain, so we turn to the other two options to deal with the fact that δ and χ are in different value spaces. One option is to map both δ and χ to $[0, 1]$ by setting $\omega_\nu = \delta_{max}^{-1}$ and $\omega_\ell = \chi_{max}^{-1}$, where δ_{max} and χ_{max} are the max value of δ and χ on a given object set $\mathcal{S}$. We further optimize query performance under this setting (**curve iii**). However, it needs to compute δ_{max} of $\mathcal{S}$ in advance, which is not easy in a real-world scenario because $\mathcal{S}$ is dynamic. The other option is to map χ to δ's value space, i.e., $\omega_\nu = 1$ and $\omega_\ell = \delta/\chi_{max}$, where $\chi_{max} = m$, m is the dimension of attributes, and it can be obtained easily. As curve **v** depicts, this method achieves the best performance in all our settings. Additionally, we also study a naive setting method, where ω_ν and ω_ℓ are fixed as constants. We traverse multiple possible ω_ℓ/ω_ν values by grid search. Fig. 17(b) illustrates that different ω_ℓ/ω_ν leads to a huge difference in HQ performance, and the optimal value falls to 10^4 (**curve iv** in Fig. 17(a)), which makes it close to the median of all δ on $\mathcal{S}$. Thus, we chose the setting of **curve v** for all other experiments.

V Parameter Sensitivity

In the following, we present additional experiments in order to investigate the sensitivity of NHQ in more detail by NHQ-NPG_{kgraph}. We vary three parameters that may affect performance: number of recall results (k), dimension of attributes (m), and *selectivity*.

Number of recall results (k). In other experiments, we set $k = 10$ by default. In this experiment, we evaluate the effect of varying k from 1 (smaller case) to 100 (larger case) on SIFT1M dataset. From the results in Fig. 18, as the number of results increases, the *QPS* of different methods degenerates. Note that the interval between our methods and others enlarges, which shows our methods' superiority when more targets need to be recalled. For existing methods, to improve the *Recall*, we must obtain numerous candidates (intermediate results that satisfy one constraint). For example, when $k = 10$ (Fig. 13(a)), the number of candidates reaches $10\times \sim 1000\times$ larger than k, which undermines the query efficiency. In contrast, our methods directly return the final results without merging.

Dimension of attributes (m). Attribute dimension corresponds to the number of categories of the attribute in a set of attributes. By default, we set $m = 3$ for other experiments. In Fig. 19, we evaluate the performance when m varies for the given dataset. From the results, as the dimension increases, each method's query performance decreases by varying degrees. This is because objects matching



(a) SIFT1M (*Recall*) (b) SIFT1M (Response time)

Figure 20: Performance under different *selectivity*.

Table 5: Experts output by the academic expert finding engine under different query methods (bold items are correct).

HNSW	**Vearch**	**NHQ-NPG_{kgraph}**
John Collomosse √	**John Collomosse** √	**John Collomosse** √
Mayu Iwata ×	**Rahul Duggal** √	**Rahul Duggal** √
Yanhao Zhang ×	-	**Hosnieh Sattar** √
Takuma Yamaguchi ×	-	**Thanh-Toan Do** √
Rahul Duggal √	-	**Gregory Zelinsky** √

attribute constraints are harder to find in this case. Note that our method still retains the advantage in higher dimension.

Selectivity. A larger *selectivity* means fewer objects match the given attributes constraint, which increases the difficulty in answering a HQ. In Fig. 20, *Recall@10* decreases under a high *selectivity* for most methods. However, NHQ-NPG_{nsw} appears to excel when there is large *selectivity*. One explanation is that, the construction strategy inserts objects incrementally facilitates linking some vertices with the same attributes but farther away from the feature vectors. This improves NHQ-NPG_{nsw}'s ability to get objects with matched attributes, especially such objects are few (high *selectivity*). NHQ-NPG_{kgraph} reaches the highest *Recall* with the least search time for different *selectivity*. In addition, Faiss is not shown in Fig. 20(a) because its *Recall@10* is less than 0.7.

W Use Case Study

We deployed our HQ method NHQ-NPG_{kgraph} into an academic expert finding system [57] to provide HQ processing services for academic papers with attributes on the *Paper* dataset. An important component in such system is the semantically similar papers retrieval that supports attribute filtering. Given a user input HQ with a descriptive text embedded as the feature vector and paper's some attributes as the structured attribute constraints, expert finding aims to extract the Top-k experts from Top-m papers that have the semantically similar vectors to the given query text as well as satisfying the given attribute constraints. Since we extract the experts from the papers returned by HQ, the results of HQ would directly affect the retrieved experts. We implemented two different HQ methods and an ANNS algorithm (i.e., HNSW [36]).

Analysis. Tab. 5 shows the Top-5 experts who have published papers on CVPR for a HQ having the abstract of [12] as the query text and an attribute constraint as CVPR under a given query time. Our method obtains the best results in comparing methods. HNSW only returns the experts who are relevant to the query text without considering the CVPR constraint. Although Vearch adds the CVPR constraints, the experts are insufficient after attribute filtering. However, the query latency will surge when increasing the number of candidate experts.

Table 6: The connectivity of PG using the NPG_{kgraph} method. The row # CONN shows the number of connected components in each graph. The row D shows the average out-degree of each graph.

	SIFT1M (w. attribute)	**SIFT1M (w/o attribute)**	**GIST1M (w. attribute)**	**GIST1M (w/o attribute)**
# CONN	2	2	3	2
D	24	23	19	21

X Connectivity

We evaluate the connectivity of PG on datasets with and without attribute in Tab. 6. The results show that PG has similar connectivity in both cases. We know that KNNG has poor connectivity due to the cluster characteristics of datasets. PG diversifies the neighbor distribution and connects clusters. For datasets with attribute, vectors in different clusters may have the same attributes, which helps connectivity in NHQ.

Y Storage Cost of NHQ and PQ-based methods

Theoretically. NHQ and PQ-based methods have the same attribute storage cost, so we only analyze their feature vector storage cost. PQ-based methods compress high-dimensional vectors into the Cartesian product of multiple sub-codebooks. Let a vector x be split into M sub-vectors u_j, $1 \leq j \leq M$, of dimension $D^* = D/M$, where D is a multiple of M. The sub-vectors are quantized separately using M quantizers with K centroids each. We need to store the $M \times K$ centroids, i.e., $KMD^* = KD$ floating-point values ($4KD$ bytes). Each vector is compressed as the code length $L = M \log_2 K$ ($L/8$ bytes). The storage cost of PQ on N vectors is: $4KD + NL/8$ bytes. NHQ builds a graph index for N vectors with at most R neighbors each. We use the adjacency list (a vertex id with R neighbor ids) to store the index, costing $4(R+1)$ bytes. We also store each raw vector with ZD bytes. The storage cost of NHQ on N vectors is: $ZND + 4(R+1)$ bytes.

On SIFT1M dataset, where $D = 128$, $r = 4$, PQ sets $K = 256$, $M = 32$, costing 32131072 bytes (31MB). NHQ sets $R = 20$, costing 512000084 bytes (488MB).

Note that PQ is often combined with IVF, known as IVFPQ, which adds a coarse step to find probing centroids near the query. This increases the storage cost of PQ-based methods depending on specific optimizations.

Experimentally. We compare the storage cost of NHQ and PQ-based methods on eight datasets in Tab. 7.

Table 7: Storage cost of NHQ (NHQ-NPG_{kgraph} and NHQ-NPG_{nsw}) and PQ-based methods (Faiss, ADBV, and Milvus). The unit of data in the table is MB.

Dataset→	**SIFT1M**	**GIST1M**	**GloVe**	**Crawl**	**UQ-V**	**Msong**	**Audio**	**Enron**
Faiss	**43**	**57**	**50**	**86**	**45**	**48**	**6**	**14**
Milvus	129	441	147	350	177	248	18	108
ADBV	116	346	133	295	151	203	14	78
NHQ-NPG_{nsw}	651	3,748	553	3,053	1,101	1,789	55	532
NHQ-NPG_{kgraph}	568	3,712	494	2,349	1,044	1,681	45	503

We observe that: (1) NHQ costs more than PQ-based methods, consistent with our theoretical analysis; (2) different PQ-based methods have different costs due to extra structures for optimization; (3) the same method has different costs on different datasets due to different optimal parameter configurations.

Notably, PQ-based methods have low accuracy due to compression loss. As shown in Fig. 7, PQ-based techniques sacrifice accuracy (<0.8 on most datasets) for cost saving. In contrast, NHQ achieves close to 1 recall with high efficiency. Therefore, we believe that there is room for improvement in the trade-off between storage and search performance.