

PGTuner: An Efficient Framework for Automatic and Transferable Configuration Tuning of Proximity Graphs

Hao Duan
Shanghai Jiao Tong University
Shang Hai, China
bunny_duanhao@sjtu.edu.cn

Bin Yao*
Shanghai Jiao Tong University
Shang Hai, China
yaobin@cs.sjtu.edu.cn

Yitong Song
Shanghai Jiao Tong University
Shang Hai, China
yitong_song@sjtu.edu.cn

Anqi Liang
Shanghai Jiao Tong University
Shang Hai, China
lianganqi@sjtu.edu.cn

Abstract

Approximate Nearest Neighbor Search (ANNS) plays a crucial role in many key areas. Proximity graphs (PGs) are the leading method for ANNS, offering the best balance between query efficiency and accuracy. However, their performance heavily depends on various construction and query parameters, which are difficult to optimize due to their complex inter-dependencies. Given that users often prioritize specific accuracy levels, efficiently identifying the optimal PG configurations to meet these targets is essential. Although some studies have explored automatic configuration tuning for PGs, they are limited by inefficiencies and suboptimal results. These issues stem from the need to construct numerous PGs for searching and re-tuning from scratch whenever the dataset changes, as well as the failure to capture the complex dependencies between configurations, query performance, and tuning objectives.

To address these challenges, we propose PGTuner, an efficient framework for automatic PG configuration tuning leveraging pre-training knowledge and model transfer techniques. PGTuner improves efficiency through a pre-trained query performance prediction (QPP) model, eliminating the need to build multiple PGs. It also features a deep reinforcement learning-based parameter configuration recommendation (PCR) model to recommend optimal configurations for specific datasets and accuracy targets. Additionally, PGTuner incorporates out-of-distribution detection and deep active learning for efficient tuning in dynamic scenarios and transferring to new datasets. Extensive experiments demonstrate that PGTuner can stably achieve the top-level tuning effect across different datasets while significantly improving tuning efficiency by up to 14.69 \times , with a 14.64 \times boost in dynamic scenarios. The code and data for PGTuner are available online at <https://github.com/hao-duan/PGTuner>.

*represents the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
<https://doi.org/XXXXXXX.XXXXXXX>

Keywords

approximate nearest neighbor search, proximity graph, configuration tuning, model transfer

ACM Reference Format:

Hao Duan, Yitong Song, Bin Yao, and Anqi Liang. 2018. PGTuner: An Efficient Framework for Automatic and Transferable Configuration Tuning of Proximity Graphs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 15 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

In recent years, Approximate Nearest Neighbor Search (ANNS) on high-dimensional vectors has gained widespread attention due to its crucial role in many fields, including information retrieval [22], data mining [35], recommendation systems [33], and retrieval-augmented generation (RAG) [13]. ANNS methods are typically categorized into tree-based [19, 36], hash-based [20, 48], quantization-based [25, 39], and proximity graph (PG) based approaches [4, 24]. Recent works [28, 37, 47] have shown that PG-based methods deliver the best performance, making them the preferred choice in industrial vector databases like Milvus¹ and Weaviate².

PG-based methods organize vectors as nodes in a graph, connecting each node to its approximate nearest neighbors [37]. During index construction, an ANNS algorithm identifies efC candidate neighbors for each node, from which M neighbors are selected for connection using a heuristic pruning algorithm. For query processing, the ANNS algorithm retrieves nodes, stores them in a candidate set of size efS , and then returns the top- k nearest nodes from the set. The query performance is evaluated in terms of accuracy (i.e., query recall) and efficiency (i.e., queries per second, QPS), with higher QPS often resulting in lower recall. The trade-off can be adjusted through both construction (e.g., efC , M) and query parameters (e.g., efS). Generally, higher values for these parameters enable the exploration of more high-quality nodes during query processing, improving recall but reducing QPS. Different parameter configurations may achieve similar recall with varying QPS. In practice, users often have specific recall requirements, making the optimization of PG parameters to meet users' recall preferences while maximizing QPS a critical focus [1, 8, 12, 43].

¹Milvus: <https://milvus.io/>

²Weaviate: <https://github.com/weaviate/weaviate>

Due to the labor-intensive nature of manual tuning and the vast space of parameter combinations, existing research primarily focuses on automating PG configuration tuning. These studies fall into two categories: search-based methods [37] and learning-based methods [27, 43]. The most commonly used search-based method is GridSearch [37], which defines a large set of candidate configurations and tests them to select the best one. However, this approach is time-consuming due to the need to construct and test numerous PGs, especially when dealing with tens of millions of vectors. VDTuner [43] and GMM [27] are two representatives of learning-based methods. VDTuner improves tuning efficiency by using constrained Bayesian optimization to recommend promising configurations for target recalls. However, it remains inefficient, as it requires constructing PGs to evaluate the recommended configurations during tuning and re-tuning from scratch whenever the dataset changes. In contrast, GMM [27] uses a pre-trained meta-model to directly predict the query performance of parameter configurations without constructing PGs, and then selects the best configuration for recommendation. Although GMM enables efficient PG tuning on a given dataset, it often recommends unsatisfactory configurations due to its inability to effectively transfer the pre-trained meta-model to the given dataset. Additionally, both VDTuner and GMM fail to effectively model the hidden dependencies between configurations, query performance, and tuning objectives, resulting in unstable and suboptimal configurations across different target recalls.

Overall, there are three challenges in automatically tuning PG: (1) how to effectively capture the hidden dependencies between configurations, query performance, and tuning objectives to reliably recommend high-performance configurations for various target recalls; (2) how to accurately evaluate PG configurations without actually constructing and testing PGs; (3) how to effectively and efficiently tune across various and dynamic datasets.

To address these challenges simultaneously, we propose PG-Tuner, an efficient framework for automatic and transferable configuration tuning of PGs. For the first challenge, PG-Tuner trains a parameter configuration recommendation (PCR) model using deep reinforcement learning (DRL) [38] to effectively capture the complex dependencies described above, enabling it to consistently recommend high-performance configurations for different target recalls. For the second and third challenges, we design a query performance prediction (QPP) model to avoid constructing PGs and leverage model transfer techniques for generalization to different datasets. Specifically, the PCR model considers generalizable tuning strategies, allowing it to build on small-scale, low-dimensional base datasets and be rapidly transferred to more complex target datasets through fine-tuning. The QPP model enables effective transfer by iteratively retraining with data collected from the target dataset, using deep active learning (DAL) [15]. To further enhance transfer efficiency, PG-Tuner also uses Out-of-Distribution (OOD) detection to assess the similarity between the given dataset and base datasets during each iteration. If similarity is detected, the transfer process can be terminated early, significantly reducing tuning time.

In summary, we make the following contributions:

- We propose PG-Tuner, a transferable auto-tuning framework for PG, which can achieve effective and efficient tuning on any given dataset and target recall.

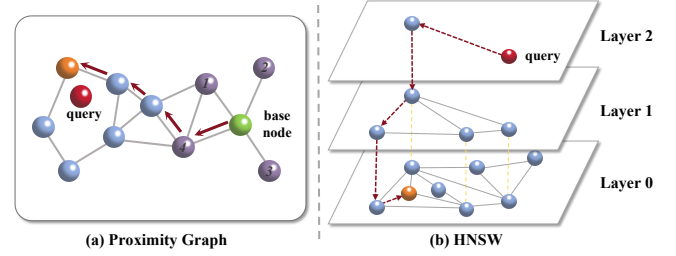


Figure 1: Examples of proximity graph and HNSW index.

- A DRL-based PCR model is introduced to capture the complex inter-dependencies between configurations, query performance, and tuning objectives. It learns a generalizable and high-quality tuning strategy and can stably recommend high-performance configurations across different datasets and target recalls.
- A QPP model for quickly evaluating PG configurations is proposed, together with an OOD detection- and DAL-based model transfer method to ensure effective and efficient transfer of the QPP model to different datasets.
- Extensive experiments on real-world datasets demonstrate that PG-Tuner can stably achieve the top-level tuning effect across different datasets while significantly improving tuning efficiency. Compared with the state-of-the-art baseline, PG-Tuner increases tuning efficiency by up to 14.69×, and also achieves a 14.64× boost in the dynamic scenario where the dataset size scales up.

2 PRELIMINARIES

In this section, we introduce the proximity graph and define the problem of PG configuration tuning.

2.1 Proximity Graph

A proximity graph (PG) is a data structure used for approximate nearest neighbor search. It treats vectors as graph nodes, with edges connecting neighboring nodes based on their distances evaluated by a distance function (e.g., Euclidean distance). As shown in Figure 1 (a), the four nodes (numbered 1–4) connected to the green node are its neighbors. During PG construction, for each inserted node i , the search starts at an initial base node (e.g., the green node) and examines the distances from i to the current base node and its neighbors. The closest node is then selected as the next base node, while dynamically recording efC candidate approximate nearest neighbors (NNs) for i . The search is terminated when no new closer node to i is found. A heuristic pruning algorithm is then applied to select M candidate nodes as the final neighbors of i . During query processing, for a query vector q , the search begins from an initial node and proceeds in the same way while continuously maintaining efS candidate approximate NNs for q . The search stops when no closer node is identified, and the top- k NNs are returned.

The query performance of PG is highly sensitive to its construction and search parameters. In general, increasing efC improves the neighbor precision of each node, which helps identify more true NNs during query processing, thereby enhancing recall. However, this improvement may increase the number of nodes explored, leading to decreasing QPS. In contrast, increasing M expands each

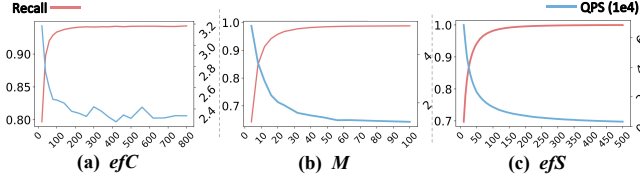


Figure 2: (a), (b) and (c) show the variations of recall and QPS with efC , M and efS on the SIFT1M dataset, respectively. The selected configuration is (500, 16, 50), fixing two of them and varying the third one each time.

node’s neighbors, which enhances recall because more potential true NNs might be explored, but decreases QPS. Similarly, Larger efS allows for exploring more nodes, boosting recall at the cost of decreased QPS. Figure 2 shows that these parameters significantly influence the query performance of PG and there is a trade-off between the recall and QPS.

In this paper, we introduce our tuning framework PGTuner based on HNSW [24], a widely used PG index. HNSW is structured as a hierarchical proximity graph, as shown in Figure 1 (b). Each layer stores a proximity graph with nodes that are a subset of those in the layer below, and the bottom graph contains all the vectors. During the HNSW construction, the highest level l for each inserted node i is determined, and i is inserted into graphs from level l down to the bottom. The insertion process in each level is identical to that described above. For query processing, HNSW starts from a random node on the top level and recursively traverses downward to find a node close to the query vector q in the bottom graph. It then continues to search in the bottom graph from this node and returns the k NNs of q . The query performance of HNSW is also determined by efC , M and efS . Note that, although the PGTuner is introduced based on HNSW, it can be easily extended to other PGs.

2.2 PG Configuration Tuning

Consider a PG with n tunable parameters, p_1, p_2, \dots, p_n , where the first n_C parameters are construction parameters and the rest are search parameters. A parameter configuration θ is defined as (p_1, p_2, \dots, p_n) , where $(p_1, p_2, \dots, p_{n_C})$ represents the construction parameter configuration θ_C , while (p_{n_C+1}, \dots, p_n) is the search parameter configuration θ_S . Assuming the value space of the i -th parameter is Θ_i , then the entire configuration space is $\Theta = \Theta_1 \times \Theta_2 \times \dots \times \Theta_n$. Given a dataset containing a base vectors dataset (BD) and a query vectors dataset (QD), and a parameter configuration θ , we first construct a PG on BD based on θ_C , then execute all queries in QD using θ_S to obtain the corresponding query performance. As users often have specific recall requirements, it is essential to find the optimal PG configurations that achieve the desired recalls while maximizing QPS. In this work, we define the PG configuration tuning problem as follows.

Definition 1 (PG Configuration Tuning, PGCT). Given a dataset $X \in \mathbb{R}^d$, a PG with a configuration space Θ , and a target recall rec_t , PGCT aims at finding the optimal configuration $\theta^* \in \Theta$ such that the corresponding recall rec_{θ^*} reaches rec_t and the queries per second

QPS_{θ^*} achieves the maximum. This can be formulated as follows.

$$\begin{aligned} \theta^* &= \arg \max_{\theta \in \Theta} (QPS_{\theta}) \\ \text{s.t. } &rec_{\theta} \geq rec_t \end{aligned} \quad (1)$$

3 OVERVIEW OF PGTUNER

In this section, we propose our automatic PG configuration tuning framework PGTuner. The compositions and working mechanism of PGTuner are presented in Figure 3.

3.1 Composition of PGTuner

PGTuner consists of six core components: (1) Data Collector, which gathers query performance data on a given dataset; (2) Feature Extractor, which extracts the dataset’s features; (3) Query Performance Prediction (QPP) model, which predicts the query performance of configurations; (4) Parameter Configuration Recommendation (PCR) model, which recommends the optimal configuration for the dataset and a target recall; (5) Data Similarity Detector, which detects the similarity between datasets; and (6) Construction Parameter Configuration Selector, which adaptively selects a small set of construction parameter configurations to update the QPP model. These components are discussed in Section 4.

3.2 Working Mechanism

As shown in Figure 3, the working mechanism of PGTuner is divided into two phases: pre-training and online tuning.

3.2.1 Pre-training. The pre-training phase aims to pre-train high-performance QPP and PCR models on a few small-scale and low-dimensional datasets (Referred to as base datasets). PGTuner first uses the Data collector to collect query performance data (referred to as base query performance data) of predetermined configurations on base datasets. Concurrently, PGTuner employs the Feature Extractor to extract representative features from the dataset as its feature vector (①). It then uses the base query performance data and feature vectors to train a high-precision QPP model (②). Finally, PGTuner utilizes the QPP model to train a powerful PCR model (③). The PCR model iteratively recommends configurations for nine default target recalls (ranging from 0.85 to 0.99). The recommended configurations in each iteration are fed into the QPP model to obtain corresponding query performance. The query performance is then fed back to the PCR model to compute rewards, which are used to adjust its tuning strategy and proceed with the next recommendation. Through this training process, the PCR model learns a high-quality and generalizable tuning strategy.

3.2.2 Online Tuning. During the online tuning phase, PGTuner adaptively recommends the optimal parameter configuration on a given new dataset and specified target recall. PGTuner first employs the Feature Extractor to extract the feature vector of the dataset (①). It then uses the Data Similarity Detector to assess the similarity between the new dataset and base datasets based on their feature vectors and the pre-trained QPP model (②). If the result indicates similarity, the pre-trained PCR model is fine-tuned in the same way as during the pre-training phase, where the configuration that reaches the target recall while achieving the maximum QPS during fine-tuning is considered optimal (③). If the result indicates

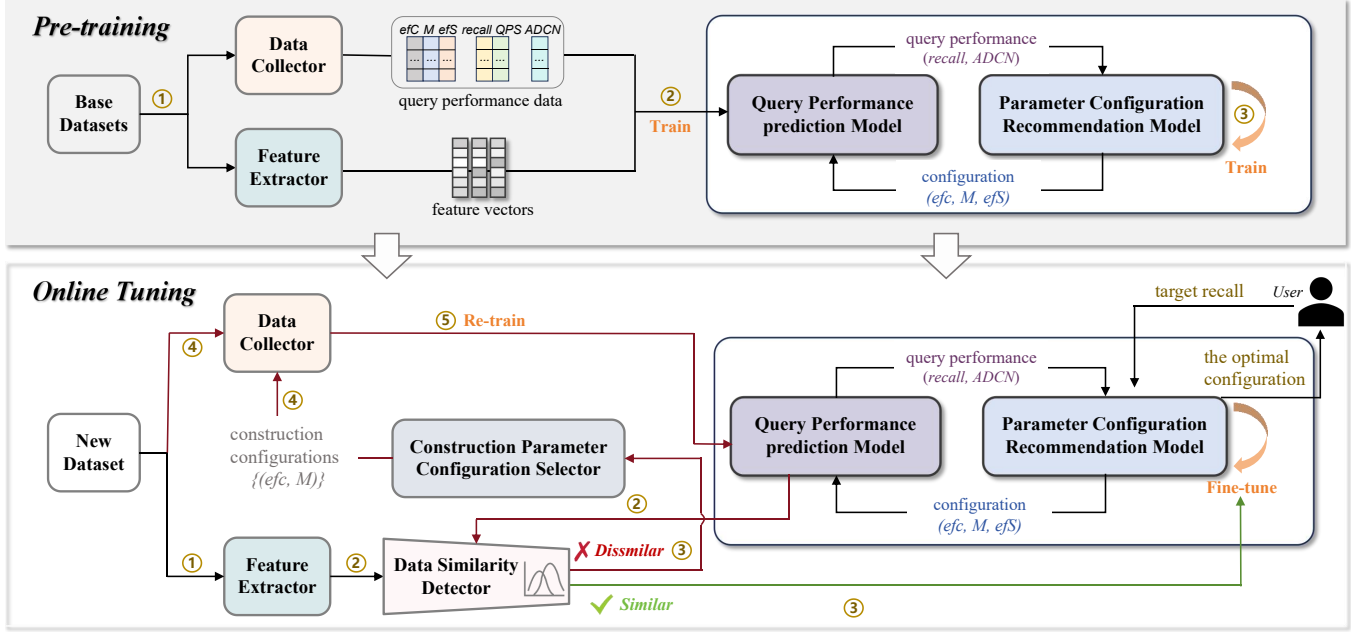


Figure 3: Architecture and Working Mechanism of PGTuner.

dissimilarity, the model transfer process is initiated to adapt the pretrained QPP model to the new dataset by iterative re-training. In each iteration, PGTuner first selects a small number of construction parameter configurations using the Construction Parameter Configuration Selector (③). The Data Collector then collects query performance data for these Construction configurations (④). The collected data is mixed with all base query performance data to re-train the QPP model (⑤). Finally, the updated QPP model is used for a new round of data similarity detection (②). This process is repeated until the detection result is similar or the maximum number of iterations is reached. Ultimately, the pre-trained PCR model is fine-tuned in the same way (③). In addition, once the QPP model is transferred to the new dataset, any changes in the target recall can be quickly addressed by fine-tuning the PCR model.

4 DESIGN AND IMPLEMENTATION OF PGTUNER

In this section, we describe the design of the six core components of PGTuner in detail.

4.1 Data Collector

The Data Collector collects query performance data for training the QPP and PCR models, which includes parameter configurations (e.g., (efC, M, efS)) and their corresponding query performance (e.g., recall and QPS). It takes the construction parameter configuration as input and returns the corresponding query performance data. For each given construction configuration, the Data Collector first constructs the corresponding PG, then iterates through all predefined search parameter configurations to repeatedly execute queries and record the corresponding query performance. During the traversal process, the query process is terminated when

both the values of search parameters and query recall reach preset thresholds, thereby speeding up the data collection process.

Note that QPS cannot be used directly for model training, as it is related to device performance. Training models with QPS data collected from a single device will impair their transferability to other devices. Considering that the time spent on distance computations dominates query processing [14] and is proportional to the number of distance calculations (DCN), we use DCN as a surrogate for QPS in model training because it is independent of device performance. To mitigate the impact of dimensional differences across different QD sizes, we further normalize the DCN by dividing it by the QD size to obtain the average DCN (ADCN). A higher QPS is associated with a lower ADCN. Therefore, maximizing QPS in the PGCT problem corresponds to minimizing ADCN.

4.2 Feature Extractor

Feature extractor extracts representative features of a dataset that are relevant to query performance. There are two important types of characteristics that can describe a dataset, which are general measures and hardness measures [27]. General measures contain basic information about the dataset, such as the dataset cardinality and vector dimensionality [17]. The hardness measure is related to the complexity of the dataset. The widely used hardness measure is the local intrinsic dimension (LID) [17, 24, 37]. A dataset with a larger LID value implies it is harder than others [17], typically requiring longer search time to achieve the same recall.

In this work, for general measures, we extract the cardinality and vector dimensionality of BD, which are related to the query recall and ADCN. Larger cardinality and dimensionality of BD generally increase its difficulty, leading to a decline in recall [37]. Moreover, the cardinality of QD is also likely to influence the query recall.

For instance, when the cardinality increases with the addition of more difficult query vectors, the recall decreases. For hardness measures, in addition to LID, we also designed two other features involving BD and QD. First, we observe that the vectors that are closer to their neighbors are more likely to form a well-connected clique. Therefore, when the vector is one of the k nearest neighbors (NNs) of a query vector, it can be accessed from multiple directions through its neighbors, improving the query efficiency. Additionally, if the difference between the distance from a query vector to its k NNs in BD and the distance to its non- k NNs is large, it is easy to distinguish them accurately during the query, thus enhancing query efficiency and accuracy. Based on these observations, we first calculate the sum of distances of each vector in BD with its k NNs, denoted as DS . We also calculate the ratio between the average distance from a query vector to its k NNs in BD and the average distance to all other non- k NNs, denoted as DR . We extracted the statistics of DS and DR similar to [27], including minimum, mean, maximum, and standard deviation. The extracted dataset features are as follows:

$$DF = (C_B, C_D, D, LID, DS_{min}, DS_{mean}, DS_{max}, DS_{std}, DR_{min}, DR_{mean}, DR_{max}, DR_{std}) \quad (2)$$

where C_B and C_D denote the cardinality of BD and QD, respectively; D represents the dimensionality; LID refers to the LID measure; $DS_{min}, DS_{mean}, DS_{max}$ are the minimum, mean, maximum, and standard deviation of DS , similar to those used in DR . The value of k is set to 10 according to the preliminary experimental results.

4.3 Query Performance Prediction Model

The QPP model is used to predict the query performance of parameter configurations on different datasets. We build a neural network composed of 4-layer MLPs [31] as the QPP model, as MLP has the advantages of strong expressive power and fast calculation speed. The model input is composed of parameter configuration and the dataset's feature vector, and the output is recall and ADCN. The loss function is the mean squared error (MSE) loss.

4.4 Parameter Configuration Recommendation Model

Configuration tuning can be modeled as a sequential decision-making task. Considering that deep reinforcement learning (DRL) has demonstrated strong capabilities in tackling such tasks, with outstanding performance in database configuration tuning [2, 16, 45], we train a DRL-based PCR model to recommend the optimal parameter configurations. During the tuning process, the tuned PG and the QPP model form the environment, while the PCR model acts as the agent in DRL. The PCR model takes the environment state as input and outputs a parameter configuration. The corresponding query performance predicted by the QPP model causes changes in the environment state, which are used to compute rewards. The agent then updates its tuning strategy based on the state and reward, aiming to recommend better configuration. This process iterates until the model stably converges, indicating that the model has learned the optimal tuning strategy. In this process, the state and reward are crucial as they directly affect the learning speed and performance of the PCR model. Therefore, we provide a

detailed description of the design of the state and reward function. Additionally, we introduce a post-processing operation to optimize the recommended configurations.

4.4.1 State. It is imperative to provide comprehensive and accurate environmental information to enable the agent to make correct decisions. Given that the agent needs to recommend the configuration that achieves the target recall while minimizing ADCN, it needs to know the target recall as well as the recall and ADCN of each recommended configuration. Furthermore, to enable the agent to leverage previous tuning experiences, it should be provided with the current best configuration and corresponding query performance at each iteration. This helps guide the agent toward promising tuning direction and motivates it to continuously recommend better configurations. Additionally, the agent should not know specific dataset features and explicit target recall value, which encourages it to learn a generalizable tuning strategy. Based on these principles, the state is designed as follows:

$$S = (\theta_l, \theta_b, rec_{\theta_l}, ADCN_{\theta_l}, \Delta rec_{l,t}, \Delta rec_{b,t}, \Delta rec_{l,b}, \Delta ADCN_{l,b}) \quad (3)$$

where θ_l and θ_b represent the last and the current best configurations; rec_{θ_l} and $ADCN_{\theta_l}$ denote the recall and ADCN of θ_l ; $\Delta rec_{l,t}$, $\Delta rec_{b,t}$, $\Delta rec_{l,b}$ and $\Delta ADCN_{l,b}$ are the performance changes, which can be calculated as follows:

$$\begin{aligned} \Delta rec_{l,t} &= rec_{\theta_l} - rec_t \\ \Delta rec_{b,t} &= rec_{\theta_b} - rec_t \\ \Delta rec_{l,b} &= rec_{\theta_l} - rec_{\theta_b} \\ \Delta ADCN_{l,b} &= \frac{-ADCN_{\theta_l} + ADCN_{\theta_b}}{ADCN_{\theta_b}} \end{aligned} \quad (4)$$

where rec_t represents the target recall, while rec_{θ_b} and $ADCN_{\theta_b}$ are the recall and ADCN of θ_b . Using $\Delta rec_{b,t}$, $\Delta rec_{l,b}$ and $\Delta ADCN_{l,b}$ instead of rec_{θ_b} and $ADCN_{\theta_b}$ in S can help reduce information redundancy and enhancing the agent's learning efficiency and effectiveness.

The aforementioned design helps the agent fully comprehend the performance changes caused by recommended configurations, thereby enabling it to make better recommendations. Note that both the θ_l and θ_b are the configurations with the minimal values in the first iteration. During the pre-training phase, the agent is trained with nine preset target recalls (from 0.85 to 0.99). In the online tuning phase, the agent can efficiently recommend optimal configurations for target recalls specified by users.

4.4.2 Reward Function. The reward function calculates rewards for each recommended configuration to encourage the agent to optimize its tuning strategy continuously. Due to the tuning objective is to achieve the target recall while minimizing ADCN, we devise the following tuning approach:

- (1) Initially, the agent should progressively recommend configuration with increasing recall until the recall meets the target recall, which is fundamental to achieving the above objective.
- (2) Once the agent has recommended a configuration whose recall reaches the target recall, it should then progressively recommend a configuration with a smaller ADCN in each iteration.

(3) During the tuning process of (2), the agent must ensure that the recall of each recommended configuration should not be lower than the target recall, otherwise the optimization process is meaningless.

On the basis of the reward function proposed by [45], we design a new reward presented below to realize the above tuning idea.

At the current iteration, we first calculate the performance changes caused by the last recommended configuration θ_l according to Equation 4, and we set different computational conditions below:

$$\begin{aligned} \text{cond1} &= \Delta \text{rec}_{l,t} < 0 \text{ and } \Delta \text{rec}_{b,t} < 0 \\ \text{cond2} &= \Delta \text{rec}_{l,t} \geq 0 \text{ and } \Delta \text{rec}_{b,t} < 0 \\ \text{cond3} &= \Delta \text{rec}_{l,t} < 0 \text{ and } \Delta \text{rec}_{b,t} \geq 0 \\ \text{cond4} &= \Delta \text{rec}_{l,t} \geq 0 \text{ and } \Delta \text{rec}_{b,t} \geq 0 \text{ and } \Delta \text{ADCN}_{l,b} \geq 0 \\ \text{cond5} &= \Delta \text{rec}_{l,t} \geq 0 \text{ and } \Delta \text{rec}_{b,t} \geq 0 \text{ and } \Delta \text{ADCN}_{l,b} < 0 \end{aligned} \quad (5)$$

Then the equation for calculating reward r is as follows:

$$r = \begin{cases} -(1 - \Delta \text{rec}_{l,t})^2 + 1 & \text{if } \text{cond1} \\ (1 + \Delta \text{rec}_{l,t})^2 \cdot (1 + \Delta \text{rec}_{l,b}) & \text{if } \text{cond2} \\ -(1 - \Delta \text{rec}_{l,t})^2 \cdot (1 - \Delta \text{rec}_{l,b}) & \text{if } \text{cond3} \\ (1 + \Delta \text{ADCN}_{l,b})^2 - 1 & \text{if } \text{cond4} \\ -(1 - \Delta \text{ADCN}_{l,b})^2 + 1 & \text{if } \text{cond5} \end{cases} \quad (6)$$

As illustrated in Equations 5 and 6, when either rec_{θ_l} or rec_{θ_b} is below rec_t , the reward is calculated using only these three recalls. This incentivizes the agent to recommend or maintain the configuration that achieves rec_t . Once both rec_{θ_l} and rec_{θ_b} reach rec_t , the reward calculation shifts to utilize only ADCN_{θ_l} and ADCN_{θ_b} . This adjustment encourages the agent to continuously recommend configurations with smaller ADCN. Through the above tuning process, the agent can recommend the optimal configuration with the smallest ADCN while achieving the target recall.

4.4.3 Post-processing for the Recommended Configuration. Due to the prediction errors of the QPP model, the query recall of the recommended optimal configuration may either fall below the target or exceed it by a large margin, meaning this configuration does not achieve the optimal performance. To address this issue, we designed a simple but effective post-processing method. For the recommended configuration, we first construct the PG based on the construction configuration, then execute queries with the search configuration to obtain the corresponding query recall. If the recall is below the target, we iteratively increase the values of the search configuration by a fixed step size and replay the queries until the recall reaches the target. At that point, the corresponding configuration is considered the ultimate recommended configuration. Conversely, if the initial recall exceeds the target, we iteratively decrease the values of the search configuration and execute queries. The process is terminated once the recall falls below the target, at which point the search configuration from the last iteration is returned. Then the configuration composed of the construction configuration and the returned search configuration is the ultimate recommended configuration. This process only needs to be executed once to optimize the optimal configuration ultimately recommended by the PCR model, and it can typically be completed within a very short time.

In this work, we utilize the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm [5], which can stably and efficiently

Algorithm 1 Data Similarity Detection

Input input feature vectors of base datasets F_b^{in} , input feature vectors of new dataset F_n^{in} , QPP model ϕ

- 1: For $f_{b_i}^{\text{in}}$ in F_b^{in} , generate output feature vectors by ϕ
 $F_b^{\text{out}} = (\phi(f_{b_1}^{\text{in}}), \phi(f_{b_2}^{\text{in}}), \dots, \phi(f_{b_N}^{\text{in}})) = (f_{b_1}^{\text{out}}, f_{b_2}^{\text{out}}, \dots, f_{b_N}^{\text{out}})$
- 2: For $f_{b_i}^{\text{out}}$ in F_b^{out} , calculate the distance between it and its k -th nearest neighbor
 $\text{Dist} = (d_1, d_2, \dots, d_N)$
- 3: Resort Dist in ascending order and set the 95th percentile as the distance threshold d_{tr}
- 4: For f_n^{in} in F_n^{in} , obtain output feature vectors in the same way
 $F_n^{\text{out}} = (f_{n_1}^{\text{out}}, f_{n_2}^{\text{out}}, \dots, f_{n_M}^{\text{out}})$
- 5: Calculate the average distance \bar{d} between all f_n^{out} in F_n^{out} and their k -th nearest neighbors in F_b^{out}
- 6: Check if $\bar{d} \leq d_{tr}$, return True or False

Output The result of the data similarity detection (True/False)

learn high-quality tuning strategy. The actor and critic networks of the TD3 are both neural networks composed of 4-layer MLPs.

4.5 Data Similarity Detector

The Data Similarity Detector detects the similarity between a new dataset and base datasets. To achieve efficient and accurate detection, we design a detection algorithm based on an Out-of-Distribution detection algorithm proposed in [34], which aims at detecting whether test data falls within the distribution of training data. It computes the distance between the test data's embedding and its k -th nearest neighbor in the training data's embeddings generated by a model. If this distance is below a predefined threshold, the test data can be considered similar to the training data.

However, one issue for performing detection is the lack of explicit test data. To address this problem, the Data Similarity Detector combines the new dataset's feature vector with all candidate parameter configurations to generate corresponding input feature vectors as the test data, denoted as F_n^{in} . It then feeds these feature vectors into the QPP model to generate output feature vectors from the model's penultimate layer, denoted as F_n^{out} , where each feature vector in it is represented as f_n^{out} . Similarly, F_b^{in} represents the input feature vectors of the base training data, and the corresponding output feature vectors are generated in the same way, denoted as F_b^{out} , where each feature vector is represented as f_b^{out} . Note that the base training data is assumed to be accessible during online tuning, which is consistent with related researches on out-of-distribution detection [42] and active learning [15].

Upon obtaining F_b^{out} and F_n^{out} , the Data Similarity Detector conducts similarity detection as shown in Algorithm 1.

- (1) Calculate the distance between each f_b^{out} and its k -th nearest neighbor in F_b^{out} , then sort all these distances in ascending order and set the 95th percentile as the distance threshold (Lines 2–3).
- (2) Calculate the average distance between all f_n^{out} in F_n^{out} and their k -th nearest neighbors in F_b^{out} (Line 5). If this average distance is below the distance threshold, the new dataset can be considered similar to the base datasets; otherwise, it is dissimilar (Line 6).

According to the preliminary experimental results, increasing k causes the threshold d_{tr} to grow more rapidly. An excessively large threshold causes incorrect detections, hindering the effective transfer of the QPP model to the new dataset and further preventing the PCR model from making high-quality recommendations. Therefore, k is set to 1 for the most stringent detection.

4.6 Construction Parameter Configuration Selector

During the online tuning phase, if a new dataset is detected to be dissimilar to the base datasets, it is essential to collect some query performance data on the new dataset to update the pre-trained QPP model, enabling it to predict accurately on both the base and new datasets. This process involves two key challenges: (1) which parameter configurations to select for data collection, and (2) how many configurations are needed. If the selected configurations are not representative, the collected data may not effectively facilitate transferring the pre-trained QPP model to the new dataset. On the other hand, selecting too few configurations results in insufficient data, which may also cause the updated model to fail to predict accurately on the new dataset. Conversely, choosing too many configurations will significantly increase the time required for data collection, thus extending the tuning time.

Considering that obtaining the query performance for given configurations can be viewed as a data labeling task, the DAL technique is well-suited to address this task. DAL aims to achieve strong model performance with fewer labeled samples, which is suitable for adaptively selecting representative construction configurations to enable effective model transfer. Therefore, we borrow from the core-set selection algorithm [32] to solve the above challenges, which iteratively selects data (i.e., core-set) from the unlabeled data with the largest nearest neighbor distance (NND) to the current labeled data. This can effectively ensure that the model updated with the core-set can perform well on the remaining unlabeled data.

In our context, the F_b^{in} of base datasets are the initial labeled data, while F_n^{in} of the new dataset are the initial unlabeled data. Additionally, we consider collecting data in a unit of construction parameter configuration rather than parameter configuration in the model transfer process. This is because compared to parameter configurations, collecting the same amount of data requires fewer construction configurations, which can avoid constructing extensive PGs. Therefore, the objective is to select construction configurations that can effectively minimize the distance between F_n^{in} and F_b^{in} . To this end, we propose a construction parameter configuration selection (CPCS) algorithm, as outlined in Algorithm 2. In each iteration, the CPCS first initializes an empty for selected construction configurations and generates the output feature vectors F_b^{out} corresponding to F_b^{in} (Lines 2–3). It then obtains the output feature vectors for each candidate construction configuration from F_n^{in} and calculates the average NND (ANND) between corresponding output feature vectors and F_b^{out} (Lines 4–7). The CPCS selects the construction configurations with the largest and average ANNDs (Lines 8–10). This selection rationale is that we find Choosing only configurations with the maximum ANND often leads to similar selections, limiting the ability to reduce the distance between unlabeled and labeled data. In contrast, adding a

Algorithm 2 Construction Parameter Configurations Selection

Input N_C candidate construction parameter configurations θ^C , base datasets' features $\{DF_b\}$, new dataset's feature DF_n ; initial input feature vectors: F_b^{in}, F_n^{in} ; QPP model ϕ , base training data \mathcal{T} , selection rounds R

- 1: **for** $i \in \{1, \dots, R\}$ **do**
- 2: Initialize the selected construction parameter configurations $U = \emptyset$, corresponding input feature vectors are F_u^{in}
- 3: Generate output feature vectors F_b^{out} corresponding to F_b^{in}
- 4: **for** $j \in \{1, \dots, N_C\}$ **do**
- 5: Obtain the input features vectors $F_{n_j}^{in}$ of θ_j^C from F_n^{in} and generate corresponding $F_{n_j}^{out}$
- 6: Calculate the average distances (d_j) between all $F_{n_j}^{out}$ in $F_{n_j}^{out}$ and their nearest neighbors in F_b^{out}
- 7: **end for**
- 8: Calculate the the average value μ of all (d_j)
- 9: $j_1 = \arg \max_j (d_j), j_2 = \arg \min_j |d_j - \mu|$
- 10: $U = \{\theta_{j_1}^C, \theta_{j_2}^C\}, F_u^{in} = \{F_{n_{j_1}}^{in}, F_{n_{j_2}}^{in}\}$
- 11: collecting query performance data according to U in Data Collector, and generating new training data \mathcal{T}_u
- 12: $\mathcal{T} = \mathcal{T} \cup \mathcal{T}_u$, retrain the QPP model ϕ using \mathcal{T}
- 13: $\theta^C = \theta^C \setminus U, F_b^{in} = F_b^{in} \cup F_u^{in}, F_n^{in} = F_n^{in} \setminus F_u^{in}$
- 14: flag = **Data Similarity Detection**(F_b^{in}, F_n^{in}, ϕ)
- 15: **if** flag is **True** **then**
- 16: **break**
- 17: **end if**
- 18: **end for**
- 19: $\{DF_b\} = \{DF_b\} \cup \{DF_n\}$
- 20: **return** $\{DF_b\}$ and ϕ

Output the new base datasets' features $\{DF_b\}$ and the updated QPP model ϕ

configuration with average ANND increases diversity and improves effectiveness. Subsequently, the CPCS uses the Data Collector to collect query performance data for the selected construction configurations and retrains the QPP model (Lines 11–13). Finally, it calls the Data Similarity Detector to assess the similarity between the current labeled data and remaining unlabeled data (Lines 14–18). If similar, this process is terminated; Otherwise, the next iteration begins. The process repeats until the detection result indicates similarity or the maximum number of iterations is reached. This design aims to avoid selecting too many construction configurations. Upon completion of this process, the pre-trained QPP model is effectively transferred to the new dataset (Lines 20–21).

5 EVALUATION

In this section, we evaluate the tuning performance of PGTuner on HNSW [24]. In Section 5.1, We compare PGTuner with baselines on six real-world datasets and conduct a successive transfer tuning study across four datasets to demonstrate its transferability. In Section 5.2, we further evaluate PGTuner's tuning ability in two scenarios where datasets are continuously changing. In Section 5.3, We conduct a comprehensive study on the influence of various components of PGTuner and the scale of pre-training data on

Table 1: Evaluated Datasets.

Dataset	Size of BD	Size of QD	Dimension	Category	Server
Deep1M	1000000	10000	96	Base	Server2
SIFT1M	1000000	10000	128	Base	Server2
Paper	2029997	10000	200	Base	Server1
Crawl	1989995	10000	300	Base	Server2
Msong	992272	200	420	Base	Server1
Nytimes	290000	1000	256	New	Server2
Glove	1183514	10000	100	New	Server2
Tiny1M	1000000	1000	384	New	Server2
GIST	1000000	1000	960	New	Server1
Deep10M	10000000	10000	96	New	Server1
SIFT50M	50000000	10000	128	New	Server2

its performance. In Section 5.4, we evaluate PGTuner on another representative PG, NSG [4], to verify its applicability to other PGs. **Experimental Environment.** All experimental data are collected on two servers. Server1 has a 14-core 2.6GHz CPU and 224GB RAM, and Server2 has a 24-core 5.8GHz CPU and 128GB RAM. Both servers are equipped with an NVIDIA 4090 GPU.

Datasets. In the experiments, we use seven real-world datasets: Nytimes³, Glove⁴, Tiny5M⁵, Paper⁶, Crawl⁷, Msong⁸, GIST⁹, Deep10M¹⁰, and SIFT1B¹¹. We respectively extract 1M subsets from Deep10M and from SIFT1B as base datasets. Additionally, a 1M subset from Tiny5M and a 50M subset from SIFT1B are respectively extracted as the new datasets for tuning. Finally, we partition five base datasets and six new datasets. The basic information of these datasets and respective servers used are provided in Table 1.

Baselines. PGTuner is compared with the following methods:

- **GridSearch** [37] constructs PGs according to predefined parameter configurations to search the best one.
- **RandomSearch** uses Latin Hypercube Sampling (LHS) [23] to randomly sample a small number of construction parameter configurations to obtain the optimal parameter configuration.
- **VDTuner** [43] employs the Gaussian process regression and Constrained Bayesian optimization to auto-tune PG.
- **GMM** [27] pre-trains a Random Forest based-meta-model. For a new dataset, GMM collects query performance data with fixed parameter configurations to update the meta-model and then uses it to predict the query performance of all candidate configurations to select the best one.

Settings. We consider 10-nearest neighbor search with Euclidean distance. For HNSW [24], we selected efC , M , and efS as the tuning parameters, which are key parameters of HNSW and have been widely considered in related researches [4, 37, 43]. Furthermore,

according to [24, 37] and preliminary experimental results, we set the ranges of efC , M , and efS as [20, 800], [4, 100], and [10, 5000], with 20, 13, and 94 distinct values, respectively. For PGTuner, the selection rounds R in Algorithm 2 is set to 7 and the according to the experimental results. Moreover, the maximum number of recommendation rounds in the PCR model is set to 250. For RandomSearch, 14 construction configurations are randomly sampled to be consistent with PGTuner. For VDTuner, the number of tuning rounds is set to 50. Additionally, we sample 5 construction configurations and collect corresponding query performance data to initialize its surrogate model. For GMM, it uses the same pre-training data as PGTuner and 14 construction configurations are also sampled to update its meta-model on new datasets.

Metrics. We evaluate the tuning performance of PGTuner and the baselines in terms of tuning effect and tuning efficiency. The tuning effect is measured by the QPS of the recommended configuration, while the tuning efficiency refers to the time taken for tuning (i.e., tuning time). Furthermore, we calculate the QPS improvements of PGTuner over the baselines to quantify the superiority of PGTuner’s tuning effect. The QPS improvement is calculated as follows:

$$\Delta QPS = (QPS_P - QPS_b) / QPS_b \times 100\%$$

Where ΔQPS represents the QPS improvement; QPS_P and QPS_b denote the QPS of PGTuner and a specific baseline, respectively.

5.1 Tuning Performance Evaluation

In this section, we evaluate the tuning performance of PGTuner and baselines on six real-world datasets. For Glove, Tiny1M and GIST datasets, we set 9 target recalls: 0.85, 0.88, 0.9, 0.92, 0.94, 0.95, 0.96, 0.98, and 0.99. For Nytimes dataset, the 0.99 recall is not considered because it is unreachable. For Deep10M and SIFT50M datasets, due to the longer tuning time on these large-scale datasets, we focus on three primary target recalls: 0.9, 0.95, and 0.99.

5.1.1 Tuning Effect Comparison. We compare the tuning effect of PGTuner with all baselines. Figure 4 reports the maximal QPS achieved by different methods on different datasets and target recalls. PGTuner achieves the highest QPS in 54% of the dataset-recall combinations. This result demonstrates PGTuner has a stable and superior tuning ability across datasets of different scales and dimensions. Firstly, compared to GridSearch, the most competitive baseline, PGTuner achieves average QPS improvements of 1.58%, -2.2%, 2.26%, 0.59%, 2.76%, and 1.28% on Nytimes, Glove, Tiny1M, GIST, Deep10M, and SIFT50M datasets. It can be seen that PGTuner stably reaches a tuning effect similar to or better than that of GridSearch overall. This is because PGTuner learns a high-quality and generalizable tuning strategy through pre-training, enabling it to recommend better configurations in most cases. Secondly, PGTuner generally outperforms all other baselines. It achieves average QPS improvements of 2.16%-12.91% compared to RandomSearch. It can be observed that RandomSearch produces relatively worse tuning effects on datasets like Nytimes, Tiny1M, GIST, and Deep10M. The reason for its varying performance across datasets is that it samples configurations independently of dataset, preventing it from consistently finding high-performance configurations for each dataset. Furthermore, PGTuner also achieves average QPS improvements

³Nytimes: <https://github.com/erikbern/ann-benchmarks/?tab=readme-ov-file>

⁴Glove: <http://downloads.zjulearning.org.cn/data/glove-100.tar.gz>

⁵Tiny5M: <https://www.cse.cuhk.edu.hk/systems/hash/gqr/datasets.html>

⁶Paper: https://drive.google.com/file/d/1t4b93_1Viudzd5D3I6_9_9Guwm1vmTn/view

⁷Crawl: <http://downloads.zjulearning.org.cn/data/crawl.tar.gz>

⁸Msong: https://drive.google.com/file/d/1UZ0T-nio8i2V8HetAx4-kt_FMK-GphHj/view

⁹GIST: <http://corpus-texmex.irisa.fr/>

¹⁰Deep10M: <http://sites.skoltech.ru/compvision/noimi/>

¹¹SIFT1B: <http://corpus-texmex.irisa.fr/>

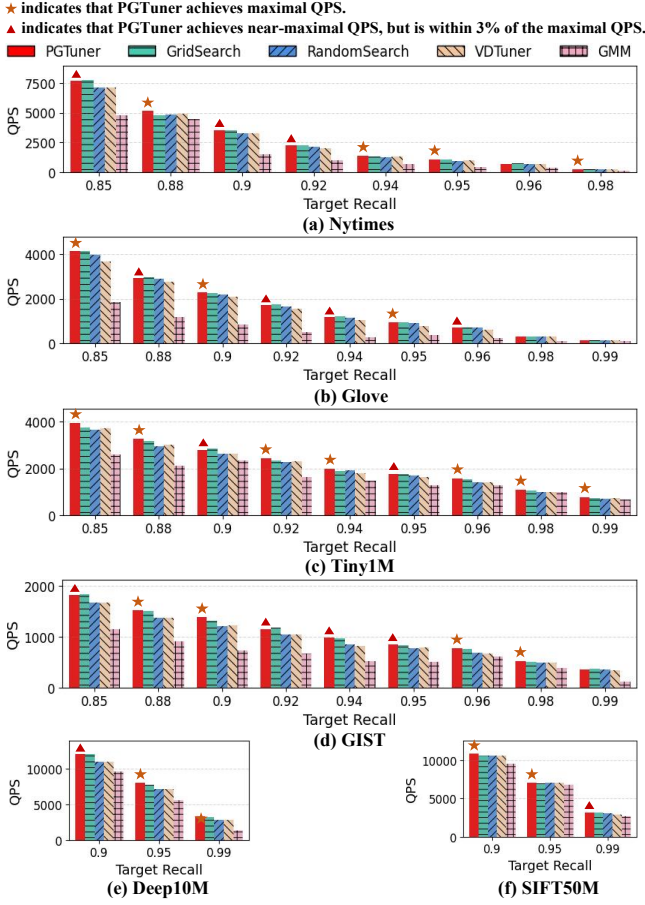


Figure 4: The QPS for all recommended parameter configurations at different target recalls.

of 3.9%-13.73% over VDTuner. The suboptimal tuning effect of VDTuner arises from its inability to accurately capture complex dependencies between configurations, query performance, and tuning objectives, leading to missing promising configurations and unstable results. Similarly, GMM performs poorly on datasets other than SIFT50M as it fails to effectively generalize the meta-model and capture the complex dependencies, resulting in large prediction errors that hinder accurate configuration recommendations.

5.1.2 Tuning Efficiency Comparison. In this section, we compare the tuning efficiency of PGTuner with baselines. Figure 5 displays the tuning time of each method. Firstly, it can be observed that PGTuner’s tuning time is similar across different target recalls on the same dataset because it can parallel tune for multiple target recalls. Secondly, PGTuner is on average 5.11 \times , 7.91 \times , 7.4 \times , 8.16 \times , 14.69 \times and 12.34 \times faster than GridSearch on Nytimes, Glove, Tiny1M, GIST, Deep10M and SIFT50M respectively. This result shows that PGTuner can significantly reduce the tuning time while stably achieving the similar tuning effect as GridSearch, and its advantage generally increases as the size and dimension of the dataset expand. In contrast, PGTuner achieves average speedups of 0.36-1.22 over RandomSearch on these datasets. RandomSearch tunes faster on

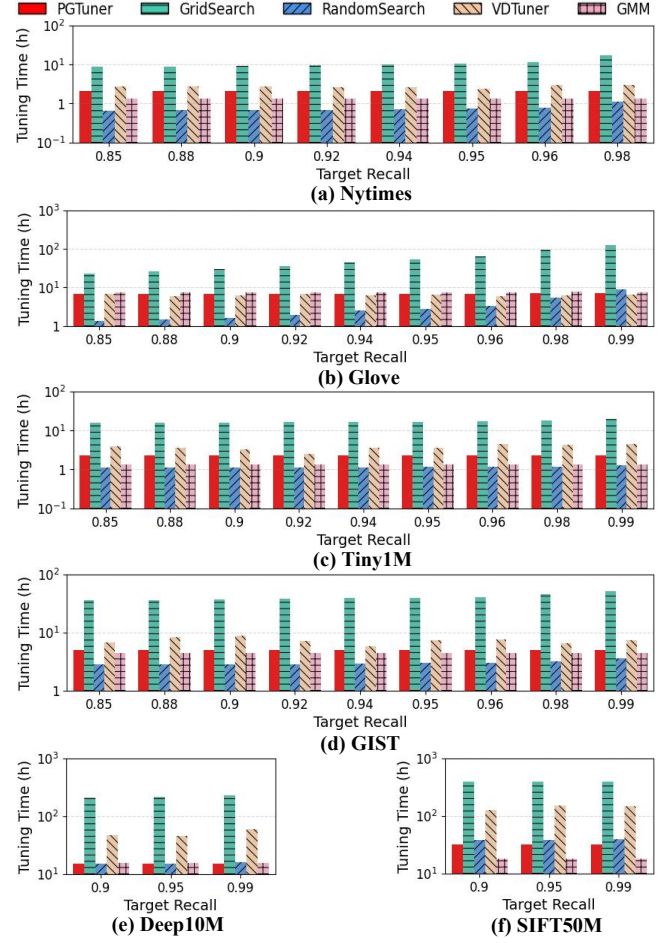


Figure 5: The tuning time at different target recalls.

Nytimes, Glove, Tiny1M, and GIST due to the short data collection time resulting from their small sizes or low dimensions, especially at low target recalls. Although PGTuner is slower on these datasets, its tuning time is still short while attaining a higher-level tuning effect. Compared with VDTuner, PGTuner improves the tuning efficiency by an average of 0.92 \times -4.5 \times . VDTuner is slightly faster on Glove also because of the low dimension and small size of Glove, leading to short construction time of HNSW. The tuning time between PGTuner and GMM is mostly minimally different, as they both collect query performance data of 14 construction configurations during tuning, which dominates the tuning time.

Besides efficiently tuning for a fixed target recall, PGTuner boosts tuning efficiency more significantly as user recall preference rises. For example, when increasing the target recall from 0.95 to 0.99, PGTuner only needs to fine-tune the PCR model to quickly recommend a new optimal configuration. In contrast, GridSearch and RandomSearch must re-execute search on all constructed PGs (assuming constructed PGs are stored) to find new optimal configuration for 0.99 recall. At this time, PGTuner is 16.89 \times -125.17 \times faster than GridSearch and 1.44 \times -10.13 \times faster than RandomSearch on Glove, GIST,

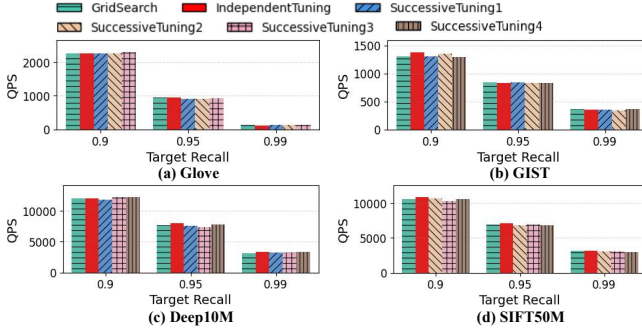


Figure 6: The comparison of QPS for independent and successive tuning at different target recalls.

Deep10M and SIFT50M datasets. These results further demonstrate PGTuner’s superior tuning efficiency and flexibility.

5.1.3 Successive Transfer Tuning Study. In Section 5.1.1, PGTuner tunes independently on four datasets without leveraging the previous tuning knowledge. However, in practice, as a vector database may continuously store new datasets, tuning on each new dataset is often performed based on previous tuning. Therefore, we use PGTuner to perform tuning across four datasets in succession to evaluate its performance for transfer tuning, considering target recalls of 0.9, 0.95, and 0.99. We set up four tuning sequences: (1) SIFT50M, Deep10M, GIST, Glove; (2) Deep10M, GIST, Glove, SIFT50M; (3) GIST, Glove, SIFT50M, Deep10M; and (4) Glove, SIFT50M, Deep10M, GIST. For each dataset, we compare the tuning effect of independent tuning with that of successive tuning in three sequences, where the dataset is not at the first position in the sequence.

Figure 6 shows the tuning effect of independent tuning and successive tuning of different tuning sequences. In the legend, IndependentTuning represents independent tuning, while SuccessiveTuning1 to SuccessiveTuning4 correspond to the four sequences. First, it can be seen that the tuning effect of successive tuning across different tuning sequences reaches or exceeds that of GridSearch. Secondly, successive tuning generally aligns with independent tuning, except for an improvement up to 11.51% at the target recall of 0.99 on Glove. This is because the knowledge gained from previous tuning helps the QPP model to be more effectively transferred to Glove, which differs significantly from the base datasets, enabling the PCR model to accurately recommend high-performance configurations. These results prove the powerful transferability of PGTuner, which can help it maintain stable and superior tuning effect in successive transfer tuning.

5.1.4 Overhead Analysis. In this section, we provide the time overhead of PGTuner during pre-training and online tuning, which are presented in Table 2 and Table 3. The tuning process of PGTuner mainly includes five stages: feature extraction, data collection, QPP model training, PCR model training and post-processing. Since the post-processing time is very short (typically less than one minute) and associated with the specific target recall. Therefore, it is not shown in Table 3 but is included in the tuning time calculation.

Since the data are collected on different servers, we cannot directly calculate the total time overhead. According to data statistics,

Table 2: The time overhead (s) of PGTuner during the pre-training phase.

Dataset	Feature Extraction	Data Collection	QPP Training	PCR Training
Deep1M	38.6	61136.1	561.6	27643.3
SIFT1M	49.7	51475.2		
Paper	213.7	129367.9		
Crawl	242.8	428819.5		
Msong	110.3	49668.4		
Total: 749327.1				

Table 3: The time overhead (s) of PGTuner on each dataset during the online tuning phase.

Dataset	Feature Extraction	Data Collection	QPP Training	PCR Training	Total
Ntimes	9.9	3303.8	3308.9	865	7487.6
Glove	44.8	20268.5	3459.1	833	24605.4
Tiny1M	168.3	3792.9	3314.6	864	8139.8
GIST	419.3	8896	3359.4	840	13514.7
Deep10M	942.8	32970.2	3298.9	853	38064.9
SIFT50M	10116.7	99718	3339.9	676.9	113851.5

the computation speed of Server2 is approximately 1.47x faster than that of Server1. Therefore, we approximate the data collection time on Server1 to the corresponding time on Server2 in Table 2 and Table 3. It can be seen that the time overhead of pre-training is 749327.1 seconds (about 208.1 hours). Although this overhead is relatively long, it can be quickly amortized in the tuning on new datasets, especially on tens-of-millions-scale datasets. For example, the total overhead of pretraining and tuning on Glove, GIST and Deep10M is 227.8 hours, which is lower than the average tuning time of GridSearch on these three datasets (240.2 hours). This result further strongly demonstrates that PGTuner is highly efficient.

5.2 Tuning Performance Comparison in Dynamic Scenarios

In real-world scenarios, a dataset may change over time, such as the increasing data size of the BD or the variable QD, making the current optimal configuration unable to achieve the target recall. In such cases, it is necessary to re-tune the PG. In this section, we simulate the two dynamic scenarios described above to further evaluate the tuning performance of PGTuner at the target recall of 0.95. We assume that the dataset changes at a low frequency so that all methods can complete the current tuning before any new changes occur in the dataset. Achieving stable real-time tuning in a highly dynamic environment is beyond the scope of this work and will be the focus of future research.

5.2.1 Comparison under the Continuous Increase in BD Size. In this section, we extract subsets with data sizes of 5M, 4M, 3M, and 2M from SIFT50M to simulate the continuous increase in data size of BD. The corresponding datasets are named SIFT5M, SIFT4M, SIFT3M, and SIFT2M. As shown in Figure 7 (a), the optimal configurations recommended by GridSearch and PGTuner for SIFT2M yield recalls

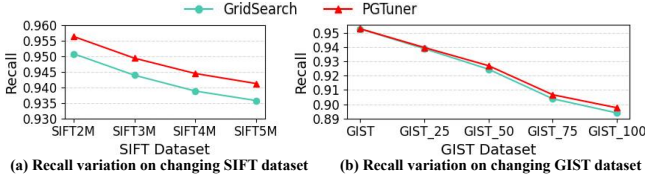


Figure 7: The variations in query recall of the initially tuned configurations under increasing BD and changing QD.

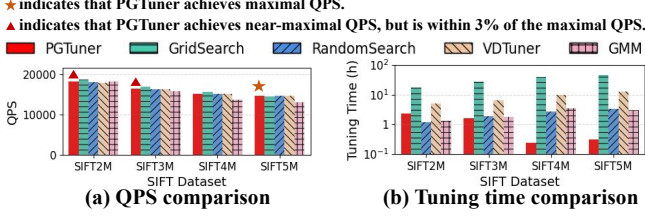


Figure 8: The QPS and Tuning time on SIFT datasets with increasing data size of BD.

lower than 0.95, with a continuous decrease. Therefore, we perform tuning sequentially from SIFT2M to SIFT5M.

Figure 8 (a) shows the maximal QPS achieved by each method on the continuously growing SIFT datasets. It can be observed that PGTuner reaches the near-optimal or optimal tuning effect on all SIFT datasets except SIFT4M. Except for being slightly inferior to GridSearch, the tuning effect of PGTuner is better than that of other baselines. This result shows that PGTuner can maintain excellent tuning capability when the size of BD continuously increases.

Figure 8 (b) displays the tuning time of each method on all datasets. It can be seen that PGTuner achieves the fastest tuning on SIFT3M-SIFT5M, especially on SIFT4M and SIFT5M. On SIFT3M-SIFT5M, PGTuner is $16.77\times$ - $169.73\times$ faster than GridSearch, $1.18\times$ - $11.55\times$ faster than RandomSearch, $4\times$ - $42.22\times$ faster than VDTuner, and $1.15\times$ - 14.64 times faster than GMM, respectively. The significant acceleration of PGTuner can primarily be attributed to the Data Similarity Detector (DSD). When PGTuner performs tuning on the SIFT3M, it first iteratively collects query performance data to update the current QPP model. However, the DSD detects the similarity between SIFT3M and the current base datasets after three iterations. As a result, the QPP model transfer is early terminated, which helps effectively reduce the amount of data collected during model transfer, thereby greatly accelerating the overall tuning process. Furthermore, when starting tuning on SIFT4M and SIFT5M, the DSD identifies the similarities during initial detection. Consequently, the QPP model does not need to be updated, and the time-consuming model transfer process is bypassed. PGTuner can then quickly recommend the optimal configuration by fine-tuning the PCR model, achieving an efficient tuning. This outcome validates the effectiveness of our designed DSD and also indicates that PGTuner could achieve higher tuning efficiency in the dynamic scenario where the BD size continuously increases.

5.2.2 Comparison under the Continuous Changes in QD. In this section, We simulate the continuous variation of QD in the GIST

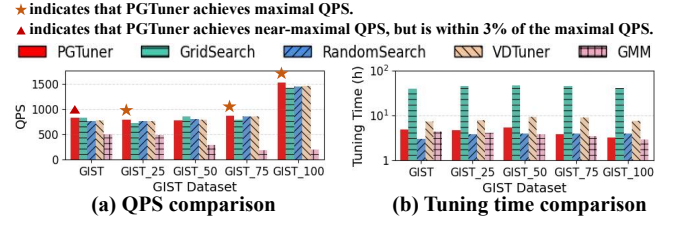


Figure 9: The QPS and Tuning time on GIST datasets with the changes of QD.

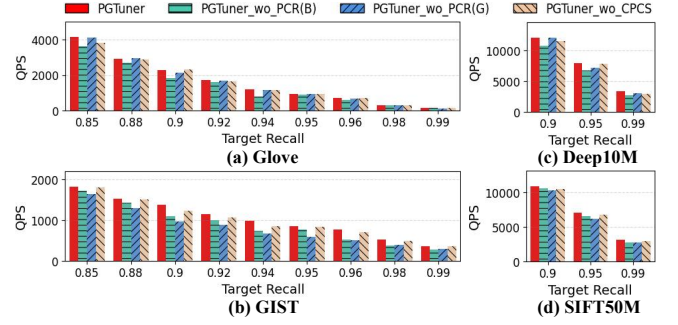


Figure 10: The QPS of PGTuner and its variants.

dataset by adding Gaussian noise at levels of 25%, 50%, 75%, and 100%, and the corresponding datasets are named GIST_25, GIST_50, GIST_75, and GIST_100 respectively. As shown in Figure 7 (b), The optimal configuration tuned on GIST results in recalls below 0.95 after GIST's QD varies, with a rapid decline as the degree of QD change increases. As a result, after tuning on GIST, we sequentially perform tuning from GIST_25 to GIST_100.

The QPS comparison is shown in Figure 9 (a). As can be seen, PGTuner shows the best tuning effect on all datasets except on GIST_50, where it performs relatively worse. This result demonstrates that PGTuner also can stably achieve advanced tuning effect in the dynamic scenario where the QD changes continuously. Figure 9 (b) shows the comparison of tuning times. Although the Data Similarity Detector fails to effectively shorten the tuning time of PGTuner due to the large variation in QD, PGTuner still achieves near-maximum tuning efficiency.

5.3 Analysis of PGTuner

In this section, we first evaluate the effectiveness of the PCR model and Constrction Parameter Configurations Selection (CPCS) algorithm of PGTuner through ablation study. Then, we investigate the impact of two key parameters in the CPCS algorithm on the performance of the updated QPP model. After that, we study the impact of the parameter k in the Feature Extractor on PGTuner's tuning effect. Finally, we conduct an exploration of the relationship between different sizes of pretraining data and PGTuner's tuning performance to determine the best pretraining data size.

5.3.1 Ablation Study. To evaluate the effectiveness of the two key components, we replace one component with a simple method while keeping the other fixed. For the PCR model, we replace it with the

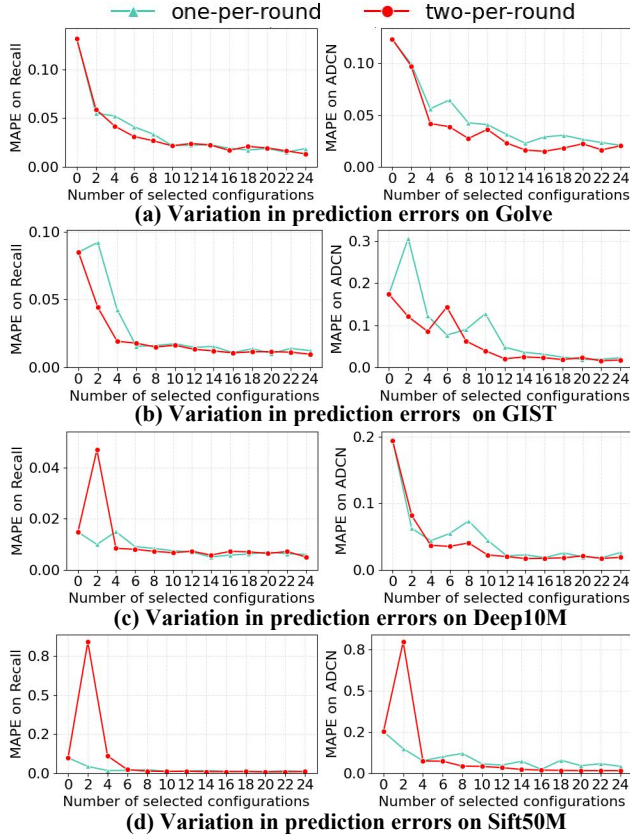


Figure 11: The variations of MAPEs on recall and ADCN with respect to the number of selected configurations under two selection strategies.

Bayesian optimization (BO) and the recommendation method from GMM respectively, and name the corresponding variants as PG-Tuner_wo_PCR(B) and PG-Tuner_wo_PCR(G). The tuning time of PG-Tuner_wo_PCR(B) on each dataset is set the same as PG-Tuner's. For the CPCS, we use random sampling as a replacement, denoted as PG-Tuner_wo_CPCS.

Figure 10 presents the QPS of PG-Tuner and its three variants. PG-Tuner outperforms all variants in most cases and achieves average improvements of up to 24.24%, 33.73% and 10.58%, compared to PG-Tuner_wo_PCR(B), PG-Tuner_wo_PCR(G) and PG-Tuner_wo_CPCS respectively. These results validate the effectiveness of the PCR model and CPCS algorithm, as removing either will significantly harm PG-Tuner's tuning effect. Additionally, PG-Tuner_wo_CPCS generally shows higher QPS than PG-Tuner_wo_PCR(B) and PG-Tuner_wo_PCR(G), indicating that the PCR model makes a greater contribution to achieving the superior tuning ability of PG-Tuner. For PG-Tuner_wo_PCR(B), the BO may be misled to deviate from the global optimum due to the QPP model's prediction errors, making it prone to yield suboptimal and unstable tuning results. In contrast, the DRL-based PCR model learns a generalizable and high-quality tuning strategy through pre-training, enabling PG-Tuner to achieve a more stable and excellent tuning. The comparison of tuning times is not shown because the differences between them are minimal.

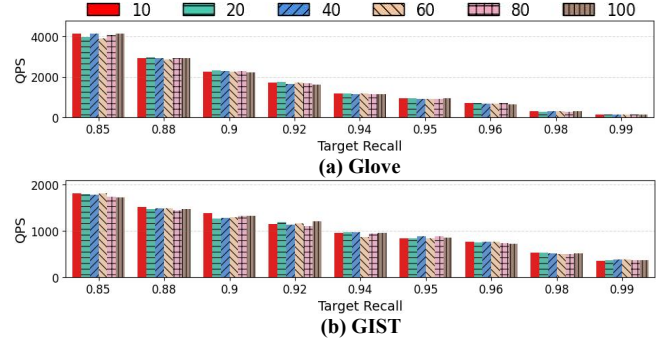


Figure 12: The comparison of tuning effects of PG-Tuner under different values of k .

5.3.2 CPCS Algorithm Study. In this section, we explore the effects of both the number of selection rounds and the number of configurations selected per round in the CPCS algorithm on the QPP model's prediction performance. The construction configurations selected per round can be set to 1 or 2, corresponding to two selection strategies: one-per-round and two-per-round. The one-per-round strategy selects the configuration with the maximum nearest neighbor (NN) distance to the labeled data per round, while the two-per-round strategy selects two configurations with the largest and average NN distances per round. We set the maximum number of selectable configurations to 24.

Figure 11 shows the variations in MAPEs for recall and ADCN as the number of selected configurations increases under two selection strategies. The x-axis value of 0 represents the initial prediction errors of the pre-trained QPP model. It can be seen that both strategies quickly reduce prediction errors to a low level, demonstrating the effectiveness of the CPCS algorithm in model transfer. Additionally, the two-per-round strategy generally leads to faster and larger reductions in MAPEs on both recall and ADCN compared to the one-per-round strategy. Moreover, with the same number of selected configurations, the one-per-round strategy requires twice as many rounds, resulting in double the model retraining time. Therefore, the two-per-round strategy is more effective and efficient.

According to the MAPEs variations under the two-per-round strategy, it can be seen that when the number of selected configurations reaches 14 (i.e., the number of selection rounds is 7), the MAPEs essentially stabilize at or near their minimum values, with minimal change in further rounds. Thus, the number of selection rounds for the two-per-round strategy is set to 7 in consideration of minimizing the tuning time of PG-Tuner.

5.3.3 Feature Extractor Study. In the Feature Extractor, parameter k is used to calculate DS and DR features, which may affect the prediction performance of the QPP model and in turn influences PG-Tuner's tuning effect. We set the values of k as 10, 20, 40, 60, 80, and 100, respectively. We then calculate the DS and DR features with different k values and train corresponding QPP and PCR models. Then tuning tests are conducted on Glove and GIST datasets. The PG-Tuner corresponding to a specific k is denoted as PG-Tuner_ k .

Figures 12 displays the tuning effects of different PG-Tuners. On Glove dataset, PG-Tuner_10 outperforms PG-Tuner_20-PG-Tuner_100

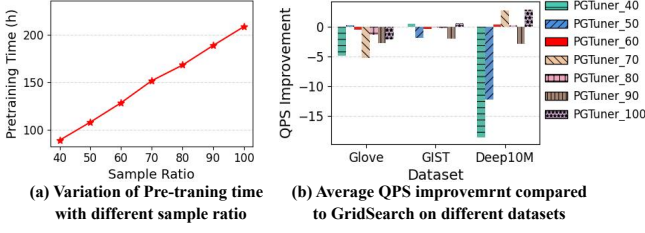


Figure 13: The comparison of pre-training time and tuning effects of PGTuner under pre-training data of different size.

with average QPS improvements of 1.51%, 1.24%, 0.81%, 2.46%, and 2.89% respectively, and the improvements on GIST are 0.44%, 0.12%, 1.61%, 2.29%, and 1.27% respectively. These results suggest that PGTuner_10 possesses the best tuning effect. Meanwhile, the tuning effects of all PGTuners vary little overall, indicating that the tuning effect of PGTuner can remain stable across different k values. Additionally, a larger k typically results in higher time and memory costs to calculate DS and DR . Therefore, setting k to 10 is the best choice for PGTuner.

5.3.4 Size of Pre-training Data Study. The size of pre-training data affects PGTuner’s tuning effect. More pre-training data usually leads to higher-quality pre-trained QPP and PCR models, enhancing PGTuner’s tuning ability. However, this also incurs higher pre-training time. Therefore, we study the impact of the pre-training data size on PGTuner’s tuning effect. The goal is to find the best data size that can maintain the superior tuning effect of PGTuner while minimizing its pre-training time. We employ LHS to sample 40%-90% of configurations at 10% intervals from all construction configurations and retrieve corresponding pre-training data. Subsequently, we execute PGTuner based on each sampled pre-training data and conduct tuning tests on Glove, GIST, and Deep10M datasets.

Figure 13 (a) shows the pre-training times corresponding to the sampled pre-training data at different ratios, which are 89.4, 108.3, 128.5, 151.6, 167.9, 188.4, and 208.1 hours respectively, showing a rapid linear increase with data size growth. Figure 13 (b) illustrates the average QPS improvements of PGTuners over GridSearch based on different ratios. The PGTuner corresponding to the ratio of r is denoted as PGTuner_r. The PGTuner_100 represents the PGTuner executed on the original pre-training data. Firstly, it can be seen that PGTuner_40 and PGTuner_50 exhibit worse tuning effect, especially on Deep10M dataset. In contrast, PGTuner_100 achieves the best tuning effect, which aligns with our analytical expectations. Moreover, both PGTuner_60 and PGTuner_80 demonstrate stable tuning effect on par with GridSearch. Therefore, according to these results, using pre-training data sampled at a 60% ratio can sustain PGTuner’s superior tuning ability while significantly reducing the pre-training time by 38.25%.

5.4 Adaptability

In this section, we evaluate PGTuner on NSG [4]. According to [4, 37] and preliminary experimental results, we consider six key parameters of NSG: K , L , L_{nsg} , R_{nsg} , C , and L_s . The ranges of these parameters are set to [100, 400], [100, 400], [150, 350], [5, 90], [300, 600], and [10, 1500], containing 4, 7, 5, 11, 4, and 52

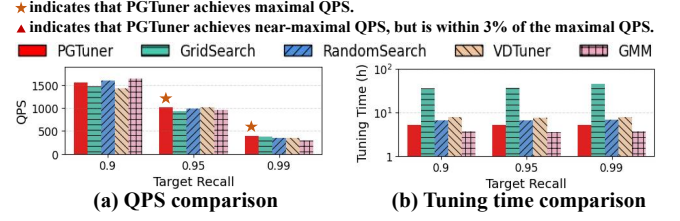


Figure 14: The QPS and Tuning time on GIST dataset.

distinct values, respectively. Specifically, K , L , L_{nsg} , R_{nsg} and C are construction parameters, while L_s is the search parameter. For PGTuner, the parameter R is set to 6, thus selecting up to 192 construction configurations. Similarly, 192 construction configurations are also randomly sampled for both RandomSearch and GMM. For VDTuner, the number of tuning iterations and initialized constructions configuration is set to 200 and 30 respectively. We use Deep1M and Paper as base datasets and GIST for tuning. Due to the long construction time of NSG, we extract subsets containing one-tenth of the data from these datasets, named Deep100K, Paper200K, and GIST100K respectively. The optimal configuration tuned on GIST100K is applied to GIST to measure the corresponding QPS. Three primary target recalls are considered: 0.9, 0.95, and 0.99.

Figure 14 (a) shows the tuning effect of different methods. PGTuner achieves improvements of up to 8.83%, 15.99%, 14.99%, and 32.56% over GridSearch, RandomSearch, VDTuner, and GMM, respectively. Figure 14 (b) shows that PGTuner also achieves higher tuning efficiency though it is $1.4\times$ slower than GMM. However, this difference is acceptable given the significant improvement in the tuning effect. These results demonstrate that PGTuner is highly applicable for configuration tuning on other PGs.

6 RELATED WORK

6.1 Automatic Configuration Tuning for PGs

The current methods for automatic PG configuration tuning can be divided into search-based methods [37, 49] and learning-based methods [27, 43].

Search-based. GridSearch [37] is widely used to search the optimal configurations by constructing numerous PGs, which is extremely costly. IGS-HNSW [49] reduces the number of constructed PG by leveraging the relationship between the average out-degree of HNSW and its query performance. However, IGS-HNSW is only applicable to tuning construction configuration for HNSW.

Learning-based. VDTuner [43] leverages the Constrained Bayesian optimization to iteratively recommend promising PG configurations. However, it is not enough efficient due to the requirement for constructing PGs during the tuning process. GMM [27] pre-trains a random forest-based meta-model to predict the query performance of candidate configurations on new datasets and recommends the best configurations. Although learning-based methods generally achieve faster tuning, they cannot transfer to new datasets effectively and fail to accurately capture the complex dependencies between PG configurations, query performance, and tuning objectives, making it difficult for them to reliably achieve outstanding performance in both tuning effect and efficiency simultaneously.

6.2 Out-of-Distribution Detection

The Out-of-Distribution (OOD) detection aims to detect whether the test data falls within the distribution of the training data. There are mainly three categories of methods for OOD detection [42]. Classification-based methods use the maximum softmax probability to detect classification models [7, 18, 21]. However, they are not suitable for the QPP model which belongs to the regression model. Density-based methods model the data distribution and classify samples with low density as OOD [9, 30, 40]. However, these methods require distributional assumptions and have high computational costs. Distance-based methods compute distances between test and training samples [11, 29, 34]. [11, 29] use Mahalanobis distance for detection, which still requires assumptions about the feature space distribution. In contrast, [34] calculates K nearest neighbor (NN) distance between the test and training samples, which is efficient and matches our work. Thus, we choose it for our data similarity detection.

6.3 Deep Active Learning

Deep active learning aims to achieve strong performance with fewer training samples. It iteratively selects unlabeled samples for labeling based on a core query strategy. Current query strategies are mainly categorized into uncertainty-based, influence-based, and representativeness-based methods [15]. Uncertainty-based methods select samples with the highest uncertainty [3, 10, 41]. However, these methods are often task-specific, limiting their generalizability. Influence-based methods select samples with the greatest impact on the model's performance [6, 26, 46], which are computationally expensive. Representativeness-based methods aim to select samples that effectively cover the feature space [32, 44]. [44] clusters samples and selects the cluster centers, which is very costly for large-scale and high-dimensional vectors. In contrast, [32] selects samples with the largest NN distance to labeled samples, offering higher generality and computational efficiency, and it also aligns well with our work. Therefore, we improve it to enable model transfer.

7 CONCLUSION

In this paper, we propose PGTuner, an automatic and transferable PG configuration tuning framework. PGTuner can effectively and efficiently recommend high-performance configurations that satisfy users' recall requirements on any given dataset. Extensive experiments demonstrate that PGTuner significantly outperforms baselines in tuning performance. Moreover, it also achieves better tuning performance in dynamic scenarios.

References

- [1] Martin Aumüller, Edgar Chavez, and Eric S Tellez. 2023. Overview of the SISAP 2023 Indexing Challenge. In *SISAP*. Springer Nature Switzerland, Cham, 255–264. https://doi.org/10.1007/978-3-031-46994-7_21
- [2] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: an online cloud database hybrid tuning system for personalized requirements. In *SIGMOD*. ACM, New York, NY, USA, 646–659. <https://doi.org/10.1145/3514221.3517882>
- [3] Juan Elenter, Navid NaderiAlizadeh, and Alejandro Ribeiro. 2022. A lagrangian duality approach to active learning. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 37575–37589. https://proceedings.neurips.cc/paper_files/paper/2022/file/f475bdd151d8b5fa01215aeda925e75c-Paper-Conference.pdf
- [4] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Vldb* 12, 5 (2019), 461–474. <https://doi.org/10.14778/3303753.3303754>
- [5] Scott Fujimoto, Herke Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. In *ICML*. PMLR, 1587–1596. <https://proceedings.mlr.press/v80/fujimoto18a.html>
- [6] Denis Gudovskiy, Alec Hodgkinson, Takuya Yamaguchi, and Sotaro Tsukizawa. 2020. Deep active learning for biased datasets via fisher kernel self-supervision. In *CVPR*. 9041–9049. <https://doi.org/10.1109/cvpr42600.2020.00906>
- [7] Dan Hendrycks and Kevin Gimpel. 2016. A baseline for detecting misclassified and out-of-distribution examples in neural networks. <https://doi.org/10.48550/arXiv.1610.02136> arXiv:1610.02136
- [8] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. 2023. Co-design hardware and algorithm for vector search. In *SC*. IEEE, 1–15. <https://doi.org/10.1145/3581784.3607045>
- [9] Ivan Kobyzev, Simon JD Prince, and Marcus A Brubaker. 2020. Normalizing flows: An introduction and review of current methods. *TPAMI* 43, 11 (2020), 3964–3979. <https://doi.org/10.1109/TPAMI.2020.2992934>
- [10] Suraj Kothawade, Saikat Ghosh, Sumit Shekhar, Yu Xiang, and Rishabh Iyer. 2022. Talisman: targeted active learning for object detection with rare classes and slices using submodular mutual information. In *ECCV*. Springer, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-031-19839-7_1
- [11] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 7167–7177. https://proceedings.neurips.cc/paper_files/paper/2018/file/abdeb6f575ac5c6676b747bca8d09cc2-Paper.pdf
- [12] Yejin Lee, Hyunji Choi, Sunhong Min, Hyunseung Lee, Sangwon Beak, Dawoon Jeong, Jae W Lee, and Tae Jun Ham. 2022. Anna: Specialized architecture for approximate nearest neighbor search. In *HPCA*. IEEE, 169–183. <https://doi.org/10.1109/HPCA53966.2022.00021>
- [13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
- [14] Conglong Li, Minjia Zhang, David G Andersen, and Yuxiong He. 2020. Improving approximate nearest neighbor search through learned adaptive early termination. In *SIGMOD*. ACM, New York, NY, USA, 2539–2554. <https://doi.org/10.1145/3318464.3380600>
- [15] Dongyuan Li, Zhen Wang, Yankai Chen, Renhe Jiang, Weiping Ding, and Manabu Okumura. 2024. A Survey on Deep Active Learning: Recent Advances and New Frontiers. *IEEE Transactions on Neural Networks and Learning Systems* 36, 4 (2024), 5879 – 5899. <https://doi.org/10.1109/TNNLS.2024.3396463>
- [16] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Vldb* 12, 12 (2019), 2118–2130. <https://doi.org/10.14778/3352063.3352129>
- [17] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *TKDE* 32, 8 (2019), 1475–1488. <https://doi.org/10.1109/TKDE.2019.2909204>
- [18] Ziqian Lin, Sreya Dutta Roy, and Yixuan Li. 2021. Mood: Multi-level out-of-distribution detection. In *CVPR*. 15313–15323. <https://doi.org/10.1109/cvpr46437.2021.01506>
- [19] Ting Liu, Andrew Moore, Ke Yang, and Alexander Gray. 2004. An investigation of practical approximate nearest neighbor algorithms. In *NeurIPS*. MIT Press, Cambridge, MA, USA, 825–832. https://proceedings.neurips.cc/paper_files/paper/2004/file/1102a326d5f7c9e04fc3c89d0ede88c9-Paper.pdf
- [20] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, Lu Qin, and Xuemin Lin. 2021. EI-LSH: An early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *The VLDB Journal* 30, 2 (2021), 215–235. <https://doi.org/10.1007/s00778-020-00635-4>
- [21] Weitang Liu, Xiaoyun Wang, John Owens, and Yixuan Li. 2020. Energy-based out-of-distribution detection. In *NeurIPS*. 21464–21475. https://proceedings.neurips.cc/paper_files/paper/2020/file/f5496252609c43eb8a3d147ab9b9c006-Paper.pdf
- [22] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. 2007. A survey of content-based image retrieval with high-level semantics. *Pattern recognition* 40, 1 (2007), 262–282. <https://doi.org/10.1016/j.patcog.2006.04.045>
- [23] Wei-Liem Loh. 1996. On Latin hypercube sampling. *The annals of statistics* 24, 5 (1996), 2058–2080. <https://doi.org/10.1214/aos/1069362310>
- [24] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *TPAMI* 42, 4 (2018), 824–836. <https://doi.org/10.1109/TPAMI.2018.2889473>
- [25] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. 2018. A survey of product quantization. *ITE Transactions on Media Technology and Applications* 6, 1 (2018), 2–10. <https://doi.org/10.3169/mta.6.2>
- [26] Mohamad Amin Mohamadi, Wonho Bae, and Danica J Sutherland. 2022. Making look-ahead active learning strategies feasible with neural tangent kernels. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA,

- 12542–12553. https://proceedings.neurips.cc/paper_files/paper/2022/file/5132940b1bcd8a7b28e9695d49d435a-Paper-Conference.pdf
- [27] Rafael Seidi Oyamada, Larissa C Shimomura, Sylvio Barbon Junior, and Daniel S Kaster. 2020. Towards proximity graph auto-configuration: An approach based on meta-learning. In *ADBIS*. Springer, Cham, 93–107. https://doi.org/10.1007/978-3-030-54832-2_9
- [28] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *PACMMOD* 1, 1 (2023), 1–27. <https://doi.org/10.1145/3588908>
- [29] Jie Ren, Stanislav Fort, Jeremiah Liu, Abhijit Guha Roy, Shreyas Padhy, and Balaji Lakshminarayanan. 2021. A simple fix to mahalanobis distance for improving near-ood detection. <https://doi.org/10.48550/arXiv.2106.09022> arXiv:2106.09022
- [30] Jie Ren, Peter J Liu, Emily Fertig, Jasper Snoek, Ryan Poplin, Mark Depristo, Joshua Dillon, and Balaji Lakshminarayanan. 2019. Likelihood ratios for out-of-distribution detection. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 14680–14691. https://proceedings.neurips.cc/paper_files/paper/2019/file/1e79596878b2320cac26dd792a6c51c9-Paper.pdf
- [31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536. <https://doi.org/10.1038/323533a0>
- [32] Ozan Sener and Silvio Savarese. 2017. Active learning for convolutional neural networks: A core-set approach. <https://doi.org/10.48550/arXiv.1708.00489> arXiv:1708.00489
- [33] Jan Suchal and Pavol Návrat. 2010. Full text search engine as scalable k-nearest neighbor recommendation system. In *IFIP International Conference on Artificial Intelligence in Theory and Practice*. Springer, Berlin, Heidelberg, 165–173. https://doi.org/10.1007/978-3-642-15286-3_16
- [34] Yiyu Sun, Yifei Ming, Xiaojin Zhu, and Yixuan Li. 2022. Out-of-distribution detection with deep nearest neighbors. In *ICML*. PMLR, 20827–20840. <https://proceedings.mlr.press/v162/sun22d.html>
- [35] George Valkanas, Theodoros Lappas, and Dimitrios Gunopulos. 2017. Mining competitors from large unstructured datasets. *TKDE* 29, 9 (2017), 1971–1984. <https://doi.org/10.1109/TKDE.2017.2705101>
- [36] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. 2013. Trinary-projection trees for approximate nearest neighbor search. *TPAMI* 36, 2 (2013), 388–403. <https://doi.org/10.1109/TPAMI.2013.125>
- [37] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *VLDB* 14, 11 (2021), 1964–1978. <https://doi.org/10.14778/3476249.3476255>
- [38] Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. 2022. Deep reinforcement learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems* 35, 4 (2022), 5064–5078. <https://doi.org/10.1109/TNNLS.2022.3207346>
- [39] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. 2017. Multiscale quantization for fast similarity search. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 5745–5755. https://proceedings.neurips.cc/paper_files/paper/2017/file/b6617980ce90f637e68c3ebe8b9be745-Paper.pdf
- [40] Zhisheng Xiao, Qing Yan, and Yali Amit. 2020. Likelihood regret: An out-of-distribution detection score for variational auto-encoder. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 20685–20696. https://proceedings.neurips.cc/paper_files/paper/2020/file/eddea82ad2755b24c4e168c5fc2ebd40-Paper.pdf
- [41] Binhui Xie, Longhui Yuan, Shuang Li, Chi Harold Liu, Xinjing Cheng, and Guoren Wang. 2022. Active learning for domain adaptation: An energy-based approach. In *AAAI*. 8708–8716. <https://doi.org/10.1609/aaai.v36i8.20850>
- [42] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. 2024. Generalized out-of-distribution detection: A survey. *TJCV* 132, 12 (2024), 5635–5662. <https://doi.org/10.1007/s11263-024-02117-4>
- [43] Tiannuo Yang, Wen Hu, Wangqi Peng, Yusen Li, Jianguo Li, Gang Wang, and Xiaoguang Liu. 2024. VDTuner: Automated Performance Tuning for Vector Data Management Systems. In *ICDE*. IEEE, 4357–4369. <https://doi.org/10.1109/ICDE60146.2024.00332>
- [44] Yazhou Yang and Marco Loog. 2022. To actively initialize active learning. *Pattern Recognition* 131, C (2022), 108836. <https://doi.org/10.1016/j.patcog.2022.108836>
- [45] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*. Association for Computing Machinery, New York, NY, USA, 415–432. <https://doi.org/10.1145/3299869.3300085>
- [46] Guang Zhao, Edward Dougherty, Byung-Jun Yoon, Francis Alexander, and Xiaoning Qian. 2021. Efficient active learning for Gaussian process classification by error reduction. In *NeurIPS*. Curran Associates Inc., Red Hook, NY, USA, 9734–9746. https://proceedings.neurips.cc/paper_files/paper/2021/file/50d2e70cdf7dd05be85e1b8df3f8ced4-Paper.pdf
- [47] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards efficient index construction and approximate nearest neighbor search in high-dimensional spaces. *VLDB* 16, 8 (2023), 1979–1991. <https://doi.org/10.14778/3594512.3594527>
- [48] Yuxin Zheng, Qi Guo, Anthony KH Tung, and Sai Wu. 2016. LazyLsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*. ACM, New York, NY, USA, 2023–2037. <https://doi.org/10.1145/2882903.2882930>
- [49] Wenyang Zhou, Yuzhi Jiang, Yingfan Liu, Xiaotian Qiao, Hui Zhang, Hui Li, and Jiangtao Cui. 2024. Auto-Tuning the Construction Parameters of Hierarchical Navigable Small World Graphs. <https://ssrn.com/abstract=4925468>

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009